Intel® IA-64 Architecture Software Developer's Manual

Volume 3: Instruction Set Reference

January 2000

Order Number: 245319-001

THIS DOCUMENT IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

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® IA-64 processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel’s website at http://developer.intel.com/design/litcentr.

Copyright © Intel Corporation, 2000

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

Contents

1 About this Manual 1-1

1.1 Overview of Volume 1: IA-64 Application Architecture 1-1
1.1.1 Part 1: IA-64 Application Architecture Guide 1-1
1.1.2 Part 2: IA-64 Optimization Guide 1-2
1.2 Overview of Volume 2: IA-64 System Architecture 1-2
1.2.1 Part 1: IA-64 System Architecture Guide 1-2
1.2.2 Part 2: IA-64 System Programmer's Guide 1-3
1.2.3 Appendices 1-4
1.3 Overview of Volume 3: Instruction Set Reference 1-4
1.3.1 Part 1: IA-64 Instruction Set Descriptions 1-4
1.3.2 Part 2: IA-32 Instruction Set Descriptions 1-4
1.4 Overview of Volume 4: Itanium™ Processor Programmer's Guide 1-5
1.5 Terminology 1-5
1.6 Related Documents 1-6

2 IA-64 Instruction Reference 2-1

2.1 Instruction Page Conventions 2-1
2.2 Instruction Descriptions 2-2

3 IA-64 Pseudo-Code Functions 3-1

4 IA-64 Instruction Formats 4-1

4.1 Format Summary 4-3
4.2 A-Unit Instruction Encodings 4-9
4.2.1 Integer ALU 4-9
4.2.2 Integer Compare 4-12
4.2.3 Multimedia 4-17
4.3 I-Unit Instruction Encodings 4-20
4.3.1 Multimedia and Variable Shifts 4-20
4.3.2 Integer Shifts 4-26
4.3.3 Test Bit 4-28
4.3.4 Miscellaneous I-Unit Instructions 4-29
4.3.5 GR/BR Moves 4-31
4.3.6 GR/Predicate/IP Moves 4-32
4.3.7 GR/AR Moves (I-Unit) 4-32
4.3.8 Sign/Zero Extend/Compute Zero Index 4-33
4.4 M-Unit Instruction Encodings 4-34
4.4.1 Loads and Stores 4-34
4.4.2 Line Prefetch 4-49
4.4.3 Semaphores 4-51
4.4.4 Set/Get FR 4-52
4.4.5 Speculation and Advanced Load Checks 4-53
4.4.6 Cache/Synchronization/RSE/ALAT 4-54
4.4.7 GR/AR Moves (M-Unit) 4-55
4.4.8 GR/CR Moves 4-56
4.4.9 Miscellaneous M-Unit Instructions 4-57
4.4.10 System/Memory Management 4-58

4.5 B-Unit Instruction Encodings 4-63
4.5.1 Branches 4-63
4.5.2 Branch Predict and Nop 4-68
4.5.3 Miscellaneous B-Unit Instructions 4-70
4.6 F-Unit Instruction Encodings 4-71
4.6.1 Arithmetic 4-73
4.6.2 Parallel Floating-point Select....4-74
4.6.3 Compare and Classify 4-74
4.6.4 Approximation 4-76
4.6.5 Minimum/Maximum and Parallel Compare 4-77
4.6.6 Merge and Logical....4-78
4.6.7 Conversion 4-79
4.6.8 Status Field Manipulation....4-79
4.6.9 Miscellaneous F-Unit Instructions 4-80
4.7 X-Unit Instruction Encodings 4-81
4.7.1 Miscellaneous X-Unit Instructions 4-81
4.7.2 Move Long Immediate$_{64}$....4-82
4.7.3 Long Branches 4-82
4.8 Immediate Formation....4-84

5 Base IA-32 Instruction Reference 5-1

5.1 IA-64 Additional Faults 5-1
5.2 Interpreting the IA-32 Instruction Reference Pages 5-2
5.2.1 IA-32 Instruction Format....5-2
5.2.2 Operation....5-6
5.2.3 Flags Affected 5-8
5.2.4 FPU Flags Affected 5-8
5.2.5 Protected Mode Exceptions 5-9
5.2.6 Real-address Mode Exceptions 5-9
5.2.7 Virtual-8086 Mode Exceptions 5-10
5.2.8 Floating-point Exceptions....5-10
5.3 IA-32 Base Instruction Reference....5-10

6 IA-32 MMX™ Technology Instruction Reference 6-1

7 IA-32 Streaming SIMD Extension Instruction Reference 7-1

7.1 IA-32 Streaming SIMD Extension Instructions 7-1
7.2 About the Intel Architecture Streaming SIMD Extensions 7-1
7.3 Single Instruction Multiple Data....7-2
7.4 New Data Types....7-2
7.5 Streaming SIMD Extension Registers 7-3
7.6 Extended Instruction Set 7-3
7.6.1 Instruction Group Review....7-4
7.7 IEEE Compliance 7-11
7.7.1 Real Number System 7-12
7.7.2 Operating on NaNs....7-17
7.8 Data Formats....7-18
7.8.1 Memory Data Formats....7-18
7.8.2 Streaming SIMD Extension Register Data Formats 7-18
7.9 Instruction Formats....7-20
7.10 Instruction Prefixes....7-20
7.11 Reserved Behavior and Software Compatibility 7-21

7.12 Notations 7-22
7.13 SIMD Integer Instruction Set Extensions 7-98
7.14 Cacheability Control Instructions 7-112

Figures

2-1 Add Pointer 2-4
2-2 Stack Frame 2-5
2-3 Operation of br.ctop and br.cexit 2-12
2-4 Operation of br.wtop and br.wexit 2-13
2-5 Deposit Example 2-37
2-6 Extract Example 2-40
2-7 Floating-point Merge Negative Sign Operation 2-63
2-8 Floating-point Merge Sign Operation 2-63
2-9 Floating-point Merge Sign and Exponent Operation 2-63
2-10 Floating-point Mix Left 2-66
2-11 Floating-point Mix Right 2-66
2-12 Floating-point Mix Left-Right 2-66
2-13 Floating-point Pack 2-77
2-14 Floating-point Merge Negative Sign Operation 2-90
2-15 Floating-point Merge Sign Operation 2-90
2-16 Floating-point Merge Sign and Exponent Operation 2-91
2-17 Floating-point Swap 2-113
2-18 Floating-point Swap Negate Left or Right 2-113
2-19 Floating-point Sign Extend Left 2-115
2-20 Floating-point Sign Extend Right 2-115
2-21 Function of getf.exp 2-119
2-22 Function of getf.sig 2-119
2-23 Mix Example 2-142
2-24 Mux1 Operation (8-bit elements) 2-158
2-25 Mux2 Examples (16-bit elements) 2-159
2-26 Pack Operation 2-163
2-27 Parallel Add Examples 2-165
2-28 Parallel Average Example 2-168
2-29 Parallel Average with Round Away from Zero Example 2-169
2-30 Parallel Average Subtract Example 2-171
2-31 Parallel Compare Example 2-173
2-32 Parallel Maximum Example 2-175
2-33 Parallel Minimum Example 2-176
2-34 Parallel Multiply Operation 2-177
2-35 Parallel Multiply and Shift Right Operation 2-178
2-36 Parallel Sum of Absolute Difference Example 2-183
2-37 Parallel Shift Left Example 2-184
2-38 Parallel Subtract Example 2-189
2-39 Function of setf.exp 2-204
2-40 Function of setf.sig 2-204
2-41 Shift Left and Add Pointer 2-208
2-42 Shift Right Pair 2-210
2-43 Unpack Operation 2-230
4-1 Bundle Format 4-1
5-1 Bit Offset for BIT[EAX,21] 5-8

5-2 Memory Bit Indexing 5-8
5-3 Version Information in Registers EAX 5-69
6-1 Operation of the MOVD Instruction 6-3
6-2 Operation of the MOVQ Instruction 6-5
6-3 Operation of the PACKSSDW Instruction 6-7
6-4 Operation of the PACKUSWB Instruction 6-10
6-5 Operation of the PADDW Instruction 6-12
6-6 Operation of the PADDSW Instruction 6-15
6-7 Operation of the PADDUSB Instruction 6-18
6-8 Operation of the PAND Instruction 6-21
6-9 Operation of the PANDN Instruction 6-23
6-10 Operation of the PCMPEQW Instruction 6-25
6-11 Operation of the PCMPGTW Instruction 6-28
6-12 Operation of the PMADDWD Instruction 6-31
6-13 Operation of the PMULHW Instruction 6-33
6-14 Operation of the PMULLW Instruction 6-35
6-15 Operation of the POR Instruction. 6-37
6-16 Operation of the PSLLW Instruction 6-39
6-17 Operation of the PSRAW Instruction 6-42
6-18 Operation of the PSRLW Instruction 6-45
6-19 Operation of the PSUBW Instruction 6-48
6-20 Operation of the PSUBSW Instruction 6-51
6-21 Operation of the PSUBUSB Instruction 6-54
6-22 High-order Unpacking and Interleaving of Bytes with the PUNPCKHBW Instruction 6-57
6-23 Low-order Unpacking and Interleaving of Bytes with the PUNPCKLBW Instruction 6-60
6-24 Operation of the PXOR Instruction 6-63
7-1 Packed Single-FP Data Type 7-2
7-2 Streaming SIMD Extension Register Set 7-3
7-3 Packed Operation 7-4
7-4 Scalar Operation 7-4
7-5 Packed Shuffle Operation 7-6
7-6 Unpack High Operation 7-7
7-7 Unpack Low Operation 7-7
7-8 Binary Real Number System 7-12
7-9 Binary Floating-point Format 7-13
7-10 Real Numbers and NaNs 7-15
7-11 Four Packed FP Data in Memory (at address 1000H) 7-18

Tables

2-1 Instruction Page Description 2-1
2-2 Instruction Page Font Conventions 2-1
2-3 Register File Notation 2-2
2-4 C Syntax Differences 2-2
2-5 Branch Types 2-9
2-6 Branch Whether Hint 2-13
2-7 Sequential Prefetch Hint 2-13
2-8 Branch Cache Deallocation Hint 2-13
2-9 Long Branch Types 2-18
2-10 IP-relative Branch Predict Whether Hint 2-20
2-11 Indirect Branch Predict Whether Hint 2-20

2-12 Importance Hint 2-20
2-13 ALAT Clear Completer....2-23
2-14 Comparison Types....2-26
2-15 64-bit Comparison Relations for Normal and unc Compares 2-27
2-16 64-bit Comparison Relations for Parallel Compares....2-27
2-17 Immediate Range for 32-bit Compares....2-29
2-18 Memory Compare and Exchange Size 2-32
2-19 Compare and Exchange Semaphore Types....2-32
2-20 Result Ranges for czx....2-35
2-21 Specified *pc* Mnemonic Values....2-42
2-22 *sf* Mnemonic Values 2-42
2-23 Floating-point Class Relations 2-49
2-24 Floating-point Classes 2-49
2-25 Floating-point Comparison Types....2-52
2-26 Floating-point Comparison Relations....2-52
2-27 Fetch and Add Semaphore Types 2-58
2-28 Floating-point Parallel Comparison Results....2-82
2-29 Floating-point Parallel Comparison Relations....2-82
2-30 *sz* Completers....2-126
2-31 Load Types 2-126
2-32 Load Hints....2-127
2-33 *fsz* Completers....2-130
2-34 FP Load Types 2-130
2-35 *lftype* Mnemonic Values....2-137
2-36 *lfhint* Mnemonic Values....2-138
2-37 Move to BR Whether Hints 2-145
2-38 Indirect Register File Mnemonics 2-150
2-39 Mux Permutations for 8-bit Elements....2-158
2-40 Pack Saturation Limits 2-163
2-41 Parallel Add Saturation Completers....2-165
2-42 Parallel Add Saturation Limits....2-165
2-43 Pcmp Relations....2-173
2-44 PMPYSHR Shift Options 2-178
2-45 Parallel Subtract Saturation Completers....2-189
2-46 Parallel Subtract Saturation Limits....2-189
2-47 Store Types 2-213
2-48 Store Hints 2-213
2-49 *xsz* Mnemonic Values....2-219
2-50 Test Bit Relations for Normal and unc tbits 2-222
2-51 Test Bit Relations for Parallel tbits....2-222
2-52 Test NaT Relations for Normal and unc tnats....2-225
2-53 Test NaT Relations for Parallel tnats 2-225
2-54 Memory Exchange Size....2-231
3-1 Pseudo-Code Functions 3-1
4-1 Relationship between Instruction Type and Execution Unit Type....4-1
4-2 Template Field Encoding and Instruction Slot Mapping 4-2
4-3 Major Opcode Assignments....4-3
4-4 Instruction Format Summary 4-4
4-5 Instruction Field Color Key....4-7
4-6 Instruction Field Names 4-8
4-7 Special Instruction Notations 4-9

4-8 Integer ALU 2-bit+1-bit Opcode Extensions 4-10
4-9 Integer ALU 4-bit+2-bit Opcode Extensions 4-10
4-10 Integer Compare Opcode Extensions 4-12
4-11 Integer Compare Immediate Opcode Extensions 4-13
4-12 Multimedia ALU 2-bit+1-bit Opcode Extensions 4-17
4-13 Multimedia ALU Size 1 4-bit+2-bit Opcode Extensions 4-17
4-14 Multimedia ALU Size 2 4-bit+2-bit Opcode Extensions 4-18
4-15 Multimedia ALU Size 4 4-bit+2-bit Opcode Extensions 4-18
4-16 Multimedia and Variable Shift 1-bit Opcode Extensions 4-20
4-17 Multimedia Opcode 7 Size 1 2-bit Opcode Extensions 4-21
4-18 Multimedia Opcode 7 Size 2 2-bit Opcode Extensions 4-21
4-19 Multimedia Opcode 7 Size 4 2-bit Opcode Extensions 4-22
4-20 Variable Shift Opcode 7 2-bit Opcode Extensions 4-22
4-21 Integer Shift/Test Bit/Test NaT 2-bit Opcode Extensions 4-26
4-22 Deposit Opcode Extensions 4-26
4-23 Test Bit Opcode Extensions 4-28
4-24 Misc I-Unit 3-bit Opcode Extensions 4-29
4-25 Misc I-Unit 6-bit Opcode Extensions 4-30
4-26 Move to BR Whether Hint Completer 4-31
4-27 Integer Load/Store/Semaphore/Get FR 1-bit Opcode Extensions 4-34
4-28 Floating-point Load/Store/Load Pair/Set FR 1-bit Opcode Extensions 4-34
4-29 Integer Load/Store Opcode Extensions 4-35
4-30 Integer Load +Reg Opcode Extensions 4-35
4-31 Integer Load/Store +Imm Opcode Extensions 4-36
4-32 Semaphore/Get FR Opcode Extensions 4-36
4-33 Floating-point Load/Store/Lfetch Opcode Extensions 4-37
4-34 Floating-point Load/Lfetch +Reg Opcode Extensions 4-37
4-35 Floating-point Load/Store/Lfetch +Imm Opcode Extensions 4-38
4-36 Floating-point Load Pair/Set FR Opcode Extensions 4-38
4-37 Floating-point Load Pair +Imm Opcode Extensions 4-39
4-38 Load Hint Completer 4-39
4-39 Store Hint Completer 4-39
4-40 Line Prefetch Hint Completer 4-49
4-41 Opcode 0 System/Memory Management 3-bit Opcode Extensions 4-58
4-42 Opcode 0 System/Memory Management 4-bit+2-bit Opcode Extensions 4-59
4-43 Opcode 1 System/Memory Management 3-bit Opcode Extensions 4-59
4-44 Opcode 1 System/Memory Management 6-bit Opcode Extensions 4-60
4-45 IP-relative Branch Types 4-63
4-46 Indirect/Miscellaneous Branch Opcode Extensions 4-64
4-47 Indirect Branch Types 4-64
4-48 Indirect Return Branch Types 4-65
4-49 Sequential Prefetch Hint Completer 4-65
4-50 Branch Whether Hint Completer 4-65
4-51 Indirect Call Whether Hint Completer 4-66
4-52 Branch Cache Deallocation Hint Completer 4-66
4-53 Indirect Predict/Nop Opcode Extensions 4-68
4-54 Branch Importance Hint Completer 4-69
4-55 IP-relative Predict Whether Hint Completer 4-69
4-56 Indirect Predict Whether Hint Completer 4-69
4-57 Miscellaneous Floating-point 1-bit Opcode Extensions 4-71
4-58 Opcode 0 Miscellaneous Floating-point 6-bit Opcode Extensions 4-71

4-59 Opcode 1 Miscellaneous Floating-point 6-bit Opcode Extensions 4-72
4-60 Reciprocal Approximation 1-bit Opcode Extensions....4-72
4-61 Floating-point Status Field Completer 4-72
4-62 Floating-point Arithmetic 1-bit Opcode Extensions....4-73
4-63 Fixed-point Multiply Add and Select Opcode Extensions 4-73
4-64 Floating-point Compare Opcode Extensions....4-74
4-65 Floating-point Class 1-bit Opcode Extensions....4-75
4-66 Misc X-Unit 3-bit Opcode Extensions 4-81
4-67 Misc X-Unit 6-bit Opcode Extensions 4-81
4-68 Move Long 1-bit Opcode Extensions....4-82
4-69 Long Branch Types....4-83
4-70 Immediate Formation....4-84
5-1 Register Encodings Associated with the +rb, +rw, and +rd Nomenclature....5-3
5-2 Exception Mnemonics, Names, and Vector Numbers 5-9
5-3 Floating-point Exception Mnemonics and Names 5-10
5-4 Information Returned by CPUID Instruction 5-68
5-5 Feature Flags Returned in EDX Register 5-69
5-6 FPATAN Zeros and NaNs 5-138
5-7 FPREM Zeros and NaNs....5-140
5-8 FPREM1 Zeros and NaNs....5-143
5-9 FSUB Zeros and NaNs....5-172
5-10 FSUBR Zeros and NaNs 5-175
5-11 FYL2X Zeros and NaNs....5-188
5-12 FYL2XP1 Zeros and NaNs 5-190
5-13 IDIV Operands....5-193
5-14 INT Cases....5-207
5-15 LAR Descriptor Validity....5-242
5-16 LEA Address and Operand Sizes....5-247
5-17 Repeat Conditions 5-327
7-1 Real Number Notation 7-13
7-2 Denormalization Process....7-16
7-3 Results of Operations with NAN Operands 7-18
7-4 Precision and Range of Streaming SIMD Extension Datatype....7-19
7-5 Real Number and NaN Encodings 7-19
7-6 Streaming SIMD Extension Instruction Behavior with Prefixes 7-20
7-7 SIMD Integer Instructions – Behavior with Prefixes....7-21
7-8 Cacheability Control Instruction Behavior with Prefixes 7-21
7-9 Key to Streaming SIMD Extension Naming Convention....7-22

Part I: IA-64 Instruction Set Descriptions

About this Manual 1

The IA-64 architecture is a unique combination of innovative features such as explicit parallelism, predication, speculation and more. The architecture is designed to be highly scalable to fill the ever increasing performance requirements of various server and workstation market segments. The IA-64 architecture features a revolutionary 64-bit instruction set architecture (ISA) which applies a new processor architecture technology called EPIC, or Explicitly Parallel Instruction Computing. A key feature of the IA-64 architecture is IA-32 instruction set compatibility.

The *Intel IA-64 Architecture Software Developer's Manual* provides a comprehensive description of the programming environment, resources, and instruction set visible to both the application and system programmer. In addition, it also describes how programmers can take advantage of IA-64 features to help them optimize code. This manual replaces the *IA-64 Application Developer's Architecture Guide* (Order Number 245188) which contains a subset of the information presented in this four-volume set.

1.1 Overview of *Volume 1: IA-64 Application Architecture*

This volume defines the IA-64 application architecture, including application level resources, programming environment, and the IA-32 application interface. This volume also describes optimization techniques used to generate high performance software.

1.1.1 Part 1: IA-64 Application Architecture Guide

Chapter 1, "About this Manual" provides an overview of all volumes in the *Intel IA-64 Architecture Software Developer's Manual*.

Chapter 2, "Introduction to the IA-64 Processor Architecture" provides an overview of the IA-64 architecture system environments.

Chapter 3, "IA-64 Execution Environment" describes the IA-64 register set used by applications and the memory organization models.

Chapter 4, "IA-64 Application Programming Model" gives an overview of the behavior of IA-64 application instructions (grouped into related functions).

Chapter 5, "IA-64 Floating-point Programming Model" describes the IA-64 floating-point architecture (including integer multiply).

Chapter 6, "IA-32 Application Execution Model in an IA-64 System Environment" describes the operation of IA-32 instructions within the IA-64 System Environment from the perspective of an application programmer.

1.1.2 Part 2: IA-64 Optimization Guide

Chapter 7, "About the IA-64 Optimization Guide" gives an overview of the IA-64 optimization guide.

Chapter 8, "Introduction to IA-64 Programming" provides an overview of the IA-64 application programming environment.

Chapter 9, "Memory Reference" discusses features and optimizations related to control and data speculation.

Chapter 10, "Predication, Control Flow, and Instruction Stream" describes optimization features related to predication, control flow, and branch hints.

Chapter 11, "Software Pipelining and Loop Support" provides a detailed discussion on optimizing loops through use of software pipelining.

Chapter 12, "Floating-point Applications" discusses current performance limitations in floating-point applications and IA-64 features that address these limitations.

1.2 Overview of *Volume 2: IA-64 System Architecture*

This volume defines the IA-64 system architecture, including system level resources and programming state, interrupt model, and processor firmware interface. This volume also provides a useful system programmer's guide for writing high performance system software.

1.2.1 Part 1: IA-64 System Architecture Guide

Chapter 1, "About this Manual" provides an overview of all volumes in the *Intel IA-64 Architecture Software Developer's Manual*.

Chapter 2, "IA-64 System Environment" introduces the environment designed to support execution of IA-64 operating systems running IA-32 or IA-64 applications.

Chapter 3, "IA-64 System State and Programming Model" describes the IA-64 architectural state which is visible only to an operating system.

Chapter 4, "IA-64 Addressing and Protection" defines the resources available to the operating system for virtual to physical address translation, virtual aliasing, physical addressing, and memory ordering.

Chapter 5, "IA-64 Interruptions" describes all interruptions that can be generated by an IA-64 processor.

Chapter 6, "IA-64 Register Stack Engine" describes the IA-64 architectural mechanism which automatically saves and restores the stacked subset (GR32 – GR 127) of the general register file.

Chapter 7, "IA-64 Debugging and Performance Monitoring" is an overview of the performance monitoring and debugging resources that are available in the IA-64 architecture.

Chapter 8, “IA-64 Interruption Vector Descriptions” lists all IA-64 interruption vectors.

Chapter 9, “IA-32 Interruption Vector Descriptions” lists IA-32 exceptions, interrupts and intercepts that can occur during IA-32 instruction set execution in the IA-64 System Environment.

Chapter 10, “IA-64 Operating System Interaction Model with IA-32 Applications” defines the operation of IA-32 instructions within the IA-64 System Environment from the perspective of an IA-64 operating system.

Chapter 11, “IA-64 Processor Abstraction Layer” describes the firmware layer which abstracts IA-64 processor implementation-dependent features.

1.2.2 Part 2: IA-64 System Programmer’s Guide

Chapter 12, “About the IA-64 System Programmer’s Guide” gives an introduction to the second section of the system architecture guide.

Chapter 13, “MP Coherence and Synchronization” describes IA-64 multi-processing synchronization primitives and the IA-64 memory ordering model.

Chapter 14, “Interruptions and Serialization” describes how the processor serializes execution around interruptions and what state is preserved and made available to low-level system code when interruptions are taken.

Chapter 15, “Context Management” describes how operating systems need to preserve IA-64 register contents and state. This chapter also describes IA-64 system architecture mechanisms that allow an operating system to reduce the number of registers that need to be spilled/filled on interruptions, system calls, and context switches.

Chapter 16, “Memory Management” introduces various IA-64 memory management strategies.

Chapter 17, “Runtime Support for Control and Data Speculation” describes the operating system support that is required for control and data speculation.

Chapter 18, “Instruction Emulation and Other Fault Handlers” describes a variety of instruction emulation handlers that IA-64 operating system are expected to support.

Chapter 19, “Floating-point System Software” discusses how IA-64 processors handle floating-point numeric exceptions and how the IA-64 software stack provides complete IEEE-754 compliance.

Chapter 20, “IA-32 Application Support” describes the support an IA-64 operating system needs to provide to host IA-32 applications.

Chapter 21, “External Interrupt Architecture” describes the IA-64 external interrupt architecture with a focus on how external asynchronous interrupt handling can be controlled by software.

Chapter 22, “I/O Architecture” describes the IA-64 I/O architecture with a focus on platform issues and support for the existing IA-32 I/O port space platform infrastructure.

Chapter 23, “Performance Monitoring Support” describes the IA-64 performance monitor architecture with a focus on what kind of operating system support is needed from IA-64 operating systems.

Chapter 24, “Firmware Overview” introduces the IA-64 firmware model, and how various firmware layers (PAL, SAL, EFI) work together to enable processor and system initialization, and operating system boot.

1.2.3 Appendices

Appendix , “IA-64 Resource and Dependency Semantics” summarizes the dependency rules that are applicable when generating code for IA-64 processors.

Appendix , “Code Examples” provides OS boot flow sample code.

1.3 Overview of *Volume 3: Instruction Set Reference*

This volume is a comprehensive reference to the IA-64 and IA-32 instruction sets, including instruction format/encoding.

1.3.1 Part 1: IA-64 Instruction Set Descriptions

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel IA-64 Architecture Software Developer’s Manual*.

Chapter 2, “IA-64 Instruction Reference” provides a detailed description of all IA-64 instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 3, “IA-64 Pseudo-Code Functions” provides a table of pseudo-code functions which are used to define the behavior of the IA-64 instructions.

Chapter 4, “IA-64 Instruction Formats” describes the encoding and instruction format instructions.

1.3.2 Part 2: IA-32 Instruction Set Descriptions

Chapter 5, “Base IA-32 Instruction Reference” provides a detailed description of all base IA-32 instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 6, “IA-32 MMX™ Technology Instruction Reference” provides a detailed description of all IA-32 MMX™ technology instructions designed to increase performance of multimedia intensive applications. Organized in alphabetical order by assembly language mnemonic.

Chapter 7, “IA-32 Streaming SIMD Extension Instruction Reference” provides a detailed description of all IA-32 Streaming SIMD Extension instructions designed to increase performance of multimedia intensive applications, and is organized in alphabetical order by assembly language mnemonic.

1.4 Overview of Volume 4: *Itanium™ Processor Programmer's Guide*

This volume describes model-specific architectural features incorporated into the Intel® Itanium™ processor, the first IA-64 processor.

Chapter 1, "About this Manual" provides an overview of four volumes in the *Intel IA-64 Architecture Software Developer's Manual.*

Chapter 2, "Register Stack Engine Support" summarizes Register Stack Engine (RSE) support provided by the Itanium processor.

Chapter 3, "Virtual Memory Management Support" details size of physical and virtual address, region register ID, and protection key register implemented on the Itanium processor.

Chapter 4, "Processor Specific Write Coalescing (WC) Behavior" describes the behavior of write coalesce (also known as Write Combine) on the Itanium processor.

Chapter 5, "Model Specific Instruction Implementation" describes model specific behavior of IA-64 instructions on the Itanium processor.

Chapter 6, "Processor Performance Monitoring" defines the performance monitoring features which are specific to the Itanium processor. This chapter outlines the targeted performance monitor usage models and describes the Itanium processor specific performance monitoring state.

Chapter 7, "Performance Monitor Events" summarizes the Itanium processor events and describes how to compute commonly used performance metrics for Itanium processor events.

Chapter 8, "Model Specific Behavior for IA-32 Instruction Execution" describes some of the key differences between an Itanium processor executing IA-32 instructions and the Pentium III processor.

1.5 Terminology

The following definitions are for terms related to the IA-64 architecture and will be used throughout this document:

Instruction Set Architecture (ISA) – Defines application and system level resources. These resources include instructions and registers.

IA-64 Architecture – The new ISA with 64-bit instruction capabilities, new performance-enhancing features, and support for the IA-32 instruction set.

IA-32 Architecture – The 32-bit and 16-bit Intel Architecture as described in the *Intel Architecture Software Developer's Manual*.

IA-64 Processor – An Intel 64-bit processor that implements both the IA-64 and the IA-32 instruction sets.

IA-64 System Environment – The IA-64 operating system privileged environment that supports the execution of both IA-64 and IA-32 code.

IA-32 System Environment – The operating system privileged environment and resources as defined by the *Intel Architecture Software Developer's Manual*. Resources include virtual paging, control registers, debugging, performance monitoring, machine checks, and the set of privileged instructions.

IA-64 Firmware – The Processor Abstraction Layer (PAL) and System Abstraction Layer (SAL).

Processor Abstraction Layer (PAL) – The IA-64 firmware layer which abstracts IA-64 processor features that are implementation dependent.

System Abstraction Layer (SAL) – The IA-64 firmware layer which abstracts IA-64 system features that are implementation dependent.

1.6 Related Documents

The following documents contain additional material related to the *Intel® IA-64 Architecture Software Developer's Manual*:

- ***Intel Architecture Software Developer's Manual*** – This set of manuals describes the Intel 32-bit architecture. They are readily available from the Intel Literature Department by calling 1-800-548-4725 and requesting Order Numbers 243190, 243191and 243192, or can be downloaded at http://developer.intel.com/design/litcentr.
- ***IA-64 Software Conventions and Runtime Architecture Guide –*** This document defines general information necessary to compile, link, and execute a program on an IA-64 operating system. It can be downloaded at http://developer.intel.com/design/ia64.
- ***IA-64 System Abstraction Layer Specification*** – This document specifies requirements to develop platform firmware for IA-64 processor systems.
- ***Extensible Firmware Interface Specification*** – This document defines a new model for the interface between operating systems and platform firmware. It can be downloaded at http://developer.intel.com/technology/efi.

IA-64 Instruction Reference 2

This chapter describes the function of each IA-64 instruction. The pages of this chapter are sorted alphabetically by assembly language mnemonic.

2.1 Instruction Page Conventions

The instruction pages are divided into multiple sections as listed in Table 2-1. The first three sections are present on all instruction pages. The last three sections are present only when necessary. Table 2-2 lists the font conventions which are used by the instruction pages.

Table 2-1. Instruction Page Description

Section Name	Contents
Format	Assembly language syntax, instruction type and encoding format
Description	Instruction function in English
Operation	Instruction function in C code
FP Exceptions	IEEE floating-point traps
Interruptions	Prioritized list of interruptions that may be caused by the instruction
Serialization	Serializing behavior or serialization requirements

Table 2-2. Instruction Page Font Conventions

Font	Interpretation
regular	(Format section) Required characters in an assembly language mnemonic
italic	(Format section) Assembly language field name that must be filled with one of a range of legal values listed in the Description section
`code`	(Operation section) C code specifying instruction behavior
`code_italic`	(Operation section) Assembly language field name corresponding to a *italic* field listed in the Format section

In the Format section, register addresses are specified using the assembly mnemonic field names given in the third column of Table 2-3. For instructions that are predicated, the Description section assumes that the qualifying predicate is true (except for instructions that modify architectural state when their qualifying predicate is false). The test of the qualifying predicate is included in the Operation section (when applicable).

In the Operation section, registers are addressed using the notation `reg[`*`addr`*`].field`. The register file being accessed is specified by `reg`, and has a value chosen from the second column of Table 2-3. The *`addr`* field specifies a register address as an assembly language field name or a register mnemonic. For the general, floating-point, and predicate register files which undergo register renaming, *`addr`* is the register address prior to renaming and the renaming is not shown. The `field` option specifies a named bit field within the register. If `field` is absent, then all fields

of the register are accessed. The only exception is when referencing the data field of the general registers (64-bits not including the NaT bit) where the notation `GR[addr]` is used. The syntactical differences between the code found in the Operation section and ANSI C is listed in Table 2-4.

Table 2-3. Register File Notation

Register File	C Notation	Assembly Mnemonic	Indirect Access
Application registers	AR	ar	
Branch registers	BR	b	
Control registers	CR	cr	
CPU identification registers	CPUID	cpuid	Y
Data breakpoint registers	DBR	dbr	Y
Instruction breakpoint registers	IBR	ibr	Y
Data TLB translation cache	DTC	n/a	
Data TLB translation registers	DTR	dtr	Y
Floating-point registers	FR	f	
General registers	GR	r	
Instruction TLB translation cache	ITC	n/a	
Instruction TLB translation registers	ITR	itr	Y
Protection key registers	PKR	pkr	Y
Performance monitor configuration registers	PMC	pmc	Y
Performance monitor data registers	PMD	pmd	Y
Predicate registers	PR	p	
Region registers	RR	rr	Y

Table 2-4. C Syntax Differences

Syntax	Function
{msb:lsb}, {bit}	Bit field specifier. When appended to a variable, denotes a bit field extending from the most significant bit specified by “msb” to the least significant bit specified by “lsb” including bits “msb” and “lsb”. If “msb” and “lsb” are equal then a single bit is accessed. The second form denotes a single bit.
u>, u>=, u<, u<=	Unsigned inequality relations. Variables on either side of the operator are treated as unsigned.
u>>, u>>=	Unsigned right shift. Zeroes are shifted into the most significant bit position.
u+	Unsigned addition. Operands are treated as unsigned, and zero-extended.
u*	Unsigned multiplication. Operands are treated as unsigned.

2.2 Instruction Descriptions

The remainder of this chapter provides a description of each of the IA-64 instructions.

Add

Format:

(*qp*) add $r_1 = r_2, r_3$	register_form	A1
(*qp*) add $r_1 = r_2, r_3, 1$	plus1_form, register_form	A1
(*qp*) add $r_1 = imm, r_3$	pseudo-op	
(*qp*) adds $r_1 = imm_{14}, r_3$	imm14_form	A4
(*qp*) addl $r_1 = imm_{22}, r_3$	imm22_form	A5

Description: The two source operands (and an optional constant 1) are added and the result placed in GR r_1. In the register form the first operand is GR r_2; in the imm_14 form the first operand is taken from the sign-extended imm_{14} encoding field; in the imm22_form the first operand is taken from the sign-extended imm_{22} encoding field. In the imm22_form, GR r_3 can specify only GRs 0, 1, 2 and 3.

The plus1_form is available only in the register_form (although the equivalent effect in the immediate forms can be achieved by adjusting the immediate).

The immediate-form pseudo-op chooses the imm14_form or imm22_form based upon the size of the immediate operand and the value of r_3.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (register_form)                              // register form
        tmp_src = GR[r2];
    else if (imm14_form)                            // 14-bit immediate form
        tmp_src = sign_ext(imm14, 14);
    else                                            // 22-bit immediate form
        tmp_src = sign_ext(imm22, 22);

    tmp_nat = (register_form ? GR[r2].nat : 0);

    if (plus1_form)
        GR[r1] = tmp_src + GR[r3] + 1;
    else
        GR[r1] = tmp_src + GR[r3];

    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

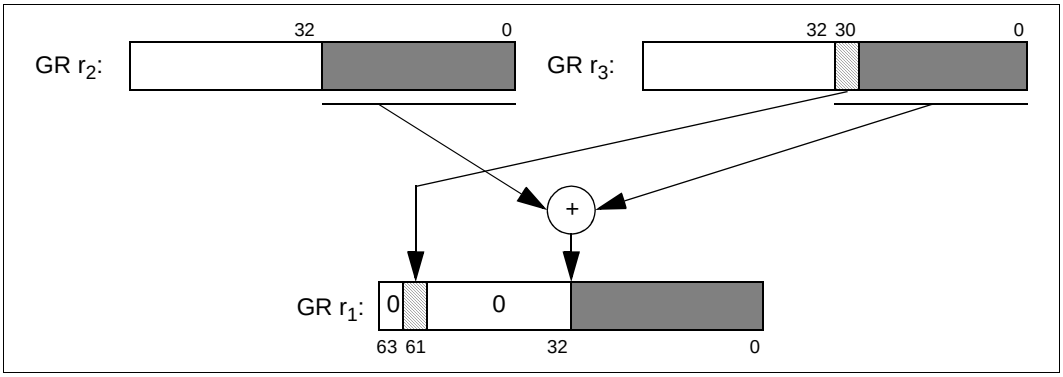
Add Pointer

Format:

(*qp*) addp4 $r_1 = r_2, r_3$	register_form	A1
(*qp*) addp4 $r_1 = imm_{14}, r_3$	imm14_form	A4

Description: The two source operands are added. The upper 32 bits of the result are forced to zero, and then bits {31:30} of GR r_3 are copied to bits {62:61} of the result. This result is placed in GR r_1. In the register_form the first operand is GR r_2; in the imm14_form the first operand is taken from the sign-extended imm_{14} encoding field.

Figure 2-1. Add Pointer



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm14, 14));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    tmp_res = tmp_src + GR[r3];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[r3]{31:30};
    GR[r1] = tmp_res;
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

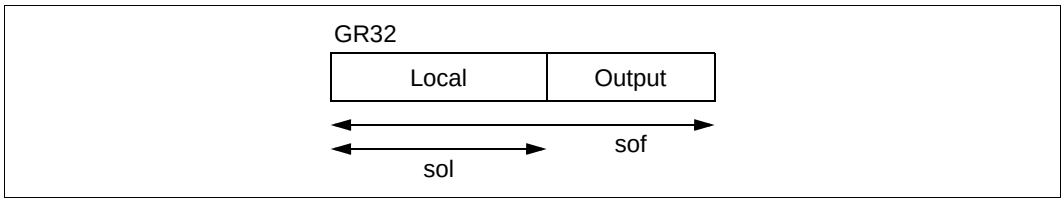
Allocate Stack Frame

Format: alloc r_1 = ar.pfs, *i*, *l*, *o*, *r* M34

Description: A new stack frame is allocated on the general register stack, and the Previous Function State register (PFS) is copied to GR r_1. The change of frame size is immediate. The write of GR r_1 and subsequent instructions in the same instruction group use the new frame. This instruction cannot be predicated.

The four parameters, *i* (size of inputs), *l* (size of locals), *o* (size of outputs), and *r* (size of rotating) specify the sizes of the regions of the stack frame.

Figure 2-2. Stack Frame



The size of the frame (sof) is determined by *i* + *l* + *o*. Note that this instruction may grow or shrink the size of the current register stack frame. The size of the local region (sol) is given by *i* + *l*. There is no real distinction between inputs and locals. They are given as separate operands in the instruction only as a hint to the assembler about how the local registers are to be used.

The rotating registers must fit within the stack frame and be a multiple of 8 in number. If this instruction attempts to change the size of CFM.sor, and the register rename base registers (CFM.rrb.gr, CFM.rrb.fr, CFM.rrb.pr) are not all zero, then the instruction will cause a Reserved Register/Field fault.

Although the assembler does not allow illegal combinations of operands for `alloc`, illegal combinations can be encoded in the instruction. Attempting to allocate a stack frame larger than 96 registers, or with the rotating region larger than the stack frame, or with the size of locals larger than the stack frame, will cause an Illegal Operation fault. An `alloc` instruction must be the first instruction in an instruction group. Otherwise, the results are undefined.

If insufficient registers are available to allocate the desired frame `alloc` will stall the processor until enough dirty registers are written to the backing store. Such mandatory RSE stores may cause the data related faults listed below.

Operation:

```
// tmp_sof, tmp_sol, tmp_sor are the fields encoded in the instruction
tmp_sof = i + l + o;
tmp_sol = i + l;
tmp_sor = r u>> 3;
check_target_register_sof(r1, tmp_sof);
if (tmp_sof u> 96 || r u> tmp_sof || tmp_sol u> tmp_sof)
   illegal_operation_fault();
if (tmp_sor != CFM.sor &&
             (CFM.rrb.gr != 0 || CFM.rrb.fr != 0 || CFM.rrb.pr != 0))
   reserved_register_field_fault();

alat_frame_update(0, tmp_sof - CFM.sof);
rse_new_frame(CFM.sof, tmp_sof);// Make room for new registers; Mandatory
                                // RSE stores can raise faults listed below.
CFM.sof = tmp_sof;
CFM.sol = tmp_sol;
CFM.sor = tmp_sor;

GR[r1] = AR[PFS];
GR[r1].nat = 0;
```

Interruptions:

Illegal Operation fault
Reserved Register/Field fault
Unimplemented Data Address fault
VHPT Data fault
Data Nested TLB fault
Data TLB fault
Alternate Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault

Logical And

Format:

(*qp*) and $r_1 = r_2, r_3$	register_form	A1
(*qp*) and $r_1 = imm_8, r_3$	imm8_form	A3

Description: The two source operands are logically ANDed and the result placed in GR r_1. In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the imm_8 encoding field.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src & GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

And Complement

Format:

(*qp*) andcm $r_1 = r_2, r_3$	register_form	A1
(*qp*) andcm $r_1 = imm_8, r_3$	imm8_form	A3

Description: The first source operand is logically ANDed with the 1's complement of the second source operand and the result placed in GR r_1. In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the imm_8 encoding field.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src & ~GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Branch

Format:

(*qp*) br.*btype.bwh.ph.dh* $target_{25}$	ip_relative_form	B1
(*qp*) br.*btype.bwh.ph.dh* $b_1 = target_{25}$	call_form, ip_relative_form	B3
br.*btype.bwh.ph.dh* $target_{25}$	counted_form, ip_relative_form	B2
br.*ph.dh* $target_{25}$	pseudo-op	
(*qp*) br.*btype.bwh.ph.dh* b_2	indirect_form	B4
(*qp*) br.*btype.bwh.ph.dh* $b_1 = b_2$	call_form, indirect_form	B5
br.*ph.dh* b_2	pseudo-op	

Description: A branch condition is evaluated, and either a branch is taken, or execution continues with the next sequential instruction. The execution of a branch logically follows the execution of all previous non-branch instructions in the same instruction group. On a taken branch, execution begins at slot 0.

Branches can be either IP-relative, or indirect. For IP-relative branches, the $target_{25}$ operand, in assembly, specifies a label to branch to. This is encoded in the branch instruction as a signed immediate displacement (imm_{21}) between the target bundle and the bundle containing this instruction ($imm_{21} = target_{25} - \text{IP} >> 4$). For indirect branches, the target address is taken from BR b_2.

Table 2-5. Branch Types

btype	Function	Branch Condition	Target Address
cond or *none*	Conditional branch	Qualifying predicate	IP-rel or Indirect
call	Conditional procedure call	Qualifying predicate	IP-rel or Indirect
ret	Conditional procedure return	Qualifying predicate	Indirect
ia	Invoke IA-32 instruction set	Unconditional	Indirect
cloop	Counted loop branch	Loop count	IP-rel
ctop, cexit	Mod-scheduled counted loop	Loop count and epilog count	IP-rel
wtop, wexit	Mod-scheduled while loop	Qualifying predicate and epilog count	IP-rel

There are two pseudo-ops for unconditional branches. These are encoded like a conditional branch (*btype* = cond), with the *qp* field specifying PR 0, and with the *bwh* hint of sptk.

The branch type determines how the branch condition is calculated and whether the branch has other effects (such as writing a link register). For the basic branch types, the branch condition is simply the value of the specified predicate register. These basic branch types are:

- **cond:** If the qualifying predicate is 1, the branch is taken. Otherwise it is not taken.
- **call:** If the qualifying predicate is 1, the branch is taken and several other actions occur:
  - The current values of the Current Frame Marker (CFM), the EC application register and the current privilege level are saved in the Previous Function State application register.
  - The caller's stack frame is effectively saved and the callee is provided with a frame containing only the caller's output region.
  - The rotation rename base registers in the CFM are reset to 0.
  - A return link value is placed in BR b_1.

- **return:** If the qualifying predicate is 1, the branch is taken and the following occurs:
  - CFM, EC, and the current privilege level are restored from PFS. (The privilege level is restored only if this does not increase privilege.)
  - The caller's stack frame is restored.
  - If the return lowers the privilege, and PSR.lp is 1, then a Lower-Privilege Transfer trap is taken.
- **ia:** The branch is taken unconditionally, if it is not intercepted by the OS. The effect of the branch is to invoke the IA-32 instruction set (by setting PSR.is to 1) and begin processing IA-32 instructions at the virtual linear target address contained in BR b_2\{31:0\}. If the qualifying predicate is not PR 0, an Illegal Operation fault is raised. If instruction set transitions are disabled (PSR.di is 1), then a Disabled Instruction Set Transition fault is raised.

  The IA-32 target effective address is calculated relative to the current code segment, i.e. EIP\{31:0\} = BR b_2\{31:0\} – CSD.base. The IA-32 instruction set can be entered at any privilege level, provided PSR.di is 0. If PSR.dfh is 1, a Disabled FP Register fault is raised on the target IA-32 instruction. No register bank switch nor change in privilege level occurs during the instruction set transition.

  Software must ensure the code segment descriptor (CSD) and selector (CS) are loaded before issuing the branch. If the target EIP value exceeds the code segment limit or has a code segment privilege violation, an IA-32_Exception(GPFault) is raised on the target IA-32 instruction. For entry into 16-bit IA-32 code, if BR b_2 is not within 64K-bytes of CSD.base a GPFault is raised on the target instruction. EFLAG.rf is unmodified until the successful completion of the first IA-32 instruction. PSR.da, PSR.id, PSR.ia, PSR.dd, and PSR.ed are cleared to zero after `br.ia` completes execution and before the first IA-32 instruction begins execution. EFLAG.rf is not cleared until the target IA-32 instruction successfully completes.

  Software must issue a `mf` instruction before the branch if memory ordering is required between IA-32 processor consistent and IA-64 unordered memory references. The processor does not ensure IA-64-instruction-set-generated writes into the instruction stream are seen by subsequent IA-32 instruction fetches. `br.ia` does not perform an instruction serialization operation. The processor does ensure that prior writes (even in the same instruction group) to GRs and FRs are observed by the first IA-32 instruction. Writes to ARs within the same instruction group as `br.ia` are not allowed, since `br.ia` may implicitly reads all ARs. If an illegal RAW dependency is present between an AR write and `br.ia`, the first IA-32 instruction fetch and execution may or may not see the updated AR value.

  IA-32 instruction set execution leaves the contents of the ALAT undefined. Software can not rely on ALAT values being preserved across an instruction set transition. All registers left in the current register stack frame are undefined across an instruction set transition. On entry to IA-32 code, existing entries in the ALAT are ignored. If the register stack contains any dirty registers, an Illegal Operation fault is raised on the `br.ia` instruction. The current register stack frame is forced to zero. To flush the register file of dirty registers, the `flushrs` instruction must be issued in an instruction group preceding the `br.ia` instruction. To enhance the performance of the instruction set transition, software can start the IA-64 register stack flush in parallel with starting the IA-32 instruction set by 1) ensuring `flushrs` is exactly one instruction group before the `br.ia`, and 2) `br.ia` is in the first B-slot. `br.ia` should always be executed in the first B-slot with a hint of "static-taken" (default), otherwise processor performance will be degraded.

  If a `br.ia` causes any IA-64 traps (e.g. Single Step trap, Taken Branch trap, or Unimplemented Instruction Address trap), IIP will contain the original 64-bit target IP. (The value will not have been zero extended from 32 bits.)

Another branch type is provided for simple counted loops. This branch type uses the Loop Count application register (LC) to determine the branch condition, and does not use a qualifying predicate:

- **cloop:** If the LC register is not equal to zero, it is decremented and the branch is taken.

In addition to these simple branch types, there are four types which are used for accelerating modulo-scheduled loops (and refer to Volume 1). Two of these are for counted loops (which use the LC register), and two for while loops (which use the qualifying predicate). These loop types use register rotation to provide register renaming, and they use predication to turn off instructions that correspond to empty pipeline stages.

The Epilog Count application register (EC) is used to count epilog stages and, for some while loops, a portion of the prolog stages. In the epilog phase, EC is decremented each time around and, for most loops, when EC is one, the pipeline has been drained, and the loop is exited. For certain types of optimized, unrolled software-pipelined loops, the target of a `br.cexit` or `br.wexit` is set to the next sequential bundle. In this case, the pipeline may not be fully drained when EC is one, and continues to drain while EC is zero.

For these modulo-scheduled loop types, the calculation of whether the branch is taken or not depends on the kernel branch condition (LC for counted types, and the qualifying predicate for while types) and on the epilog condition (whether EC is greater than one or not).

These branch types are of two categories: top and exit. The top types (ctop and wtop) are used when the loop decision is located at the bottom of the loop body and therefore a taken branch will continue the loop while a fall through branch will exit the loop. The exit types (cexit and wexit) are used when the loop decision is located somewhere other than the bottom of the loop and therefore a fall though branch will continue the loop and a taken branch will exit the loop. The exit types are also used at intermediate points in an unrolled pipelined loop. (For more details, refer to Volume 1).

The modulo-scheduled loop types are:

- **ctop** and **cexit:** These branch types behave identically, except in the determination of whether to branch or not. For `br.ctop`, the branch is taken if either LC is non-zero or EC is greater than one. For `br.cexit`, the opposite is true. It is not taken if either LC is non-zero or EC is greater than one and is taken otherwise.

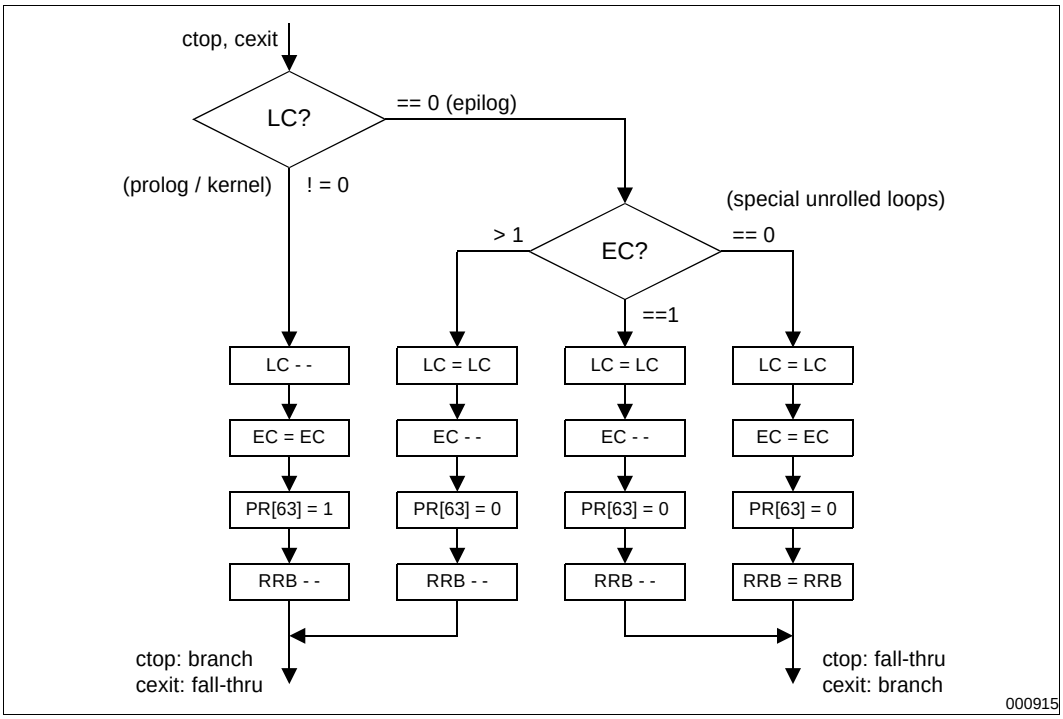
  These branch types also use LC and EC to control register rotation and predicate initialization. During the prolog and kernel phase, when LC is non-zero, LC counts down. When `br.ctop` or `br.cexit` is executed with LC equal to zero, the epilog phase is entered, and EC counts down. When `br.ctop` or `br.cexit` is executed with LC equal to zero and EC equal to one, a final decrement of EC and a final register rotation are done. If LC and EC are equal to zero, register rotation stops. These other effects are the same for the two branch types, and are described in Figure 2-3.

  wtop and **wexit:** These branch types behave identically, except in the determination of whether to branch or not. For `br.wtop`, the branch is taken if either the qualifying predicate is one or EC is greater than one. For `br.wexit`, the opposite is true. It is not taken if either the qualifying predicate is one or EC is greater than one, and is taken otherwise.

  These branch types also use the qualifying predicate and EC to control register rotation and predicate initialization. During the prolog phase, the qualifying predicate is either zero or one, depending upon the scheme used to program the loop. During the kernel phase, the qualifying predicate is one. During the epilog phase, the qualifying predicate is zero, and EC counts down. When `br.wtop` or `br.wexit` is executed with the qualifying predicate equal to zero and EC equal to one, a final decrement of EC and a final register rotation are done. If the qualifying

predicate and EC are zero, register rotation stops. These other effects are the same for the two branch types, and are described in Figure 2-4.

Figure 2-3. Operation of br.ctop and br.cexit

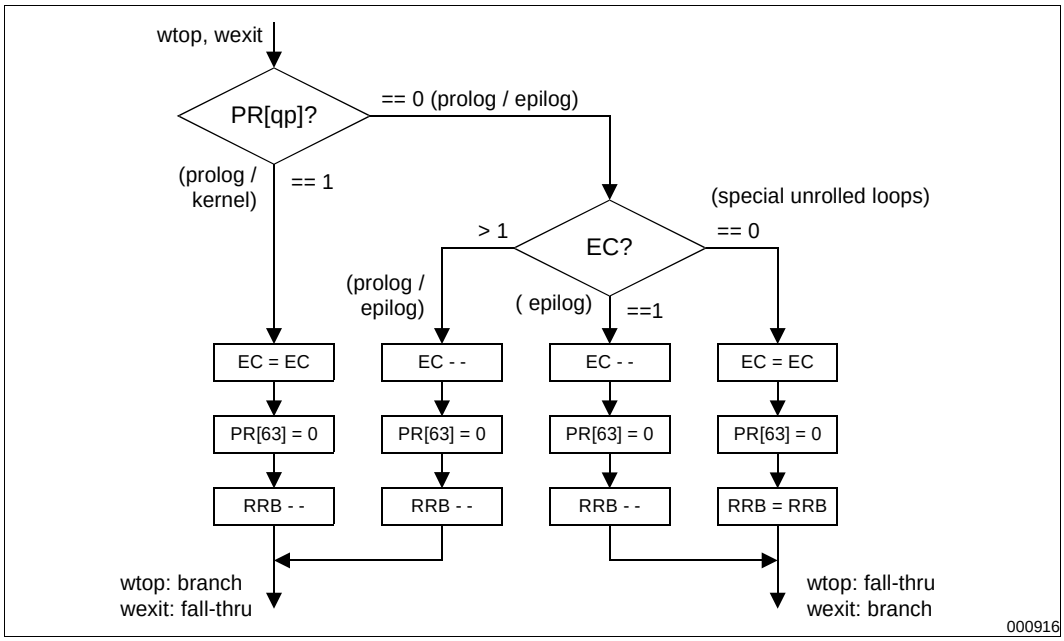


The loop-type branches (`br.cloop`, `br.ctop`, `br.cexit`, `br.wtop`, and `br.wexit`) are only allowed in instruction slot 2 within a bundle. Executing such an instruction in either slot 0 or 1 will cause an Illegal Operation fault, whether the branch would have been taken or not.

Read after Write (RAW) and Write after Read (WAR) dependency requirements are slightly different for branch instructions. Changes to BRs, PRs, and PFS by non-branch instructions are visible to a subsequent branch instruction in the same instruction group (i.e. a limited RAW is allowed for these resources). This allows for a low-latency compare-branch sequence, for example. The normal RAW requirements apply to the LC and EC application registers, and the RRBs.

Within an instruction group, a WAR dependency on PR 63 is not allowed if both the reading and writing instructions are branches. For example, a `br.wtop` or `br.wexit` may not use PR[63] as its qualifying predicate and PR[63] cannot be the qualifying predicate for any branch preceding a `br.wtop` or `br.wexit` in the same instruction group.

Figure 2-4. Operation of br.wtop and br.wexit



For dependency purposes, the loop-type branches effectively always write their associated resources, whether they are taken or not. The cloop type effectively always writes LC. When LC is 0, a cloop branch leaves it unchanged, but hardware may implement this as a re-write of LC with the same value. Similarly, `br.ctop` and `br.cexit` effectively always write LC, EC, the RRBs, and PR[63]. `br.wtop` and `br.wexit` effectively always write EC, the RRBs, and PR[63].

Values for various branch hint completers are shown in the following tables. Whether Prediction Strategy hints are shown in Table 2-6. Sequential Prefetch hints are shown in Table 2-7. Branch Cache Deallocation hints are shown in Table 2-8. See "Branch Prediction Hints" in Volume 1.

Table 2-6. Branch Whether Hint

bwh Completer	Branch Whether Hint
spnt	Static Not-Taken
sptk	Static Taken
dpnt	Dynamic Not-Taken
dptk	Dynamic Taken

Table 2-7. Sequential Prefetch Hint

ph Completer	Sequential Prefetch Hint
few or *none*	Few lines
many	Many lines

Table 2-8. Branch Cache Deallocation Hint

dh Completer	Branch Cache Deallocation Hint
none	Don't deallocate
clr	Deallocate branch information

Operation:

```
if (ip_relative_form)                        // determine branch target
    tmp_IP = IP + sign_ext((imm21 << 4), 25);
else // indirect_form
    tmp_IP = BR[b2];

if (btype != 'ia')                           // for IA-64 branches,
    tmp_IP = tmp_IP & ~0xf;                  //  ignore bottom 4 bits of target

lower_priv_transition = 0;

switch (btype) {
    case 'cond':                             // simple conditional branch
        tmp_taken = PR[qp];
        break;

    case 'call':                             // call saves a return link
        tmp_taken = PR[qp];
        if (tmp_taken) {
            BR[b1] = IP + 16;

            AR[PFS].pfm = CFM;               // ... and saves the stack frame
            AR[PFS].pec = AR[EC];
            AR[PFS].ppl = PSR.cpl;

            alat_frame_update(CFM.sol, 0);
            rse_preserve_frame(CFM.sol);
            CFM.sof -= CFM.sol;              // new frame size is size of outs
            CFM.sol = 0;
            CFM.sor = 0;
            CFM.rrb.gr = 0;
            CFM.rrb.fr = 0;
            CFM.rrb.pr = 0;
        }
        break;

    case 'ret':                              // return restores stack frame
        tmp_taken = PR[qp];
        if (tmp_taken) {
            // tmp_growth indicates the amount to move logical TOP *up*:
            // tmp_growth = sizeof(previous out) - sizeof(current frame)
            // a negative amount indicates a shrinking stack
            tmp_growth = (AR[PFS].pfm.sof - AR[PFS].pfm.sol) - CFM.sof;
            alat_frame_update(-AR[PFS].pfm.sol, 0);
            rse_fatal = rse_restore_frame(AR[PFS].pfm.sol, tmp_growth,
CFM.sof);
            if (rse_fatal) {                 // See Section 6.4 in Volume 2.
                CFM.sof = 0;
                CFM.sol = 0;
                CFM.sor = 0;
                CFM.rrb.gr = 0;
                CFM.rrb.fr = 0;
                CFM.rrb.pr = 0;
            } else // normal branch return
                CFM = AR[PFS].pfm;

            rse_enable_current_frame_load();
            AR[EC] = AR[PFS].pec;
            if (PSR.cpl u< AR[PFS].ppl) {  // ... and restores privilege
                PSR.cpl = AR[PFS].ppl;
                lower_priv_transition = 1;
            }
        }
```

```
    break;

case 'ia':                              // switch to IA mode
    tmp_taken = 1;
    if (qp != 0)
        illegal_operation_fault();
    if (AR[BSPSTORE] != AR[BSP])
        illegal_operation_fault();
    if (PSR.di)
        disabled_instruction_set_transition_fault();
    PSR.is = 1;                         // set IA-32 Instruction Set Mode
    CFM.sof = 0;                        //force current stack frame
    CFM.sol = 0;                        //to zero
    CFM.sor = 0;
    CFM.rrb.gr = 0;
    CFM.rrb.fr = 0;
    CFM.rrb.pr = 0;
    rse_invalidate_non_current_regs();
//compute effective instruction pointer
    EIP{31:0} = tmp_IP{31:0} - AR[CSD].Base;

// Note the register stack is disabled during IA-32 instruction
// set execution
    break;

case 'cloop':                           // simple counted loop
    if (slot != 2)
        illegal_operation_fault();
    tmp_taken = (AR[LC] != 0);
    if (AR[LC] != 0)
        AR[LC]--;
    break;

case 'ctop':
case 'cexit':                           // SW pipelined counted loop
    if (slot != 2)
        illegal_operation_fault();
    if (btype == 'ctop') tmp_taken =  ((AR[LC] != 0) || (AR[EC] u> 1));
    if (btype == 'cexit')tmp_taken = !((AR[LC] != 0) || (AR[EC] u> 1));
    if (AR[LC] != 0) {
        AR[LC]--;
        AR[EC] = AR[EC];
        PR[63] = 1;
        rotate_regs();
    } else if (AR[EC] != 0) {
        AR[LC] = AR[LC];
        AR[EC]--;
        PR[63] = 0;
        rotate_regs();
    } else {
        AR[LC] = AR[LC];
        AR[EC] = AR[EC];
        PR[63] = 0;
        CFM.rrb.gr = CFM.rrb.gr;
        CFM.rrb.fr = CFM.rrb.fr;
        CFM.rrb.pr = CFM.rrb.pr;
    }
    break;

case 'wtop':
case 'wexit':                           // SW pipelined while loop
    if (slot != 2)
```

```
            illegal_operation_fault();
        if (btype == 'wtop') tmp_taken =  (PR[qp] || (AR[EC] u> 1));
        if (btype == 'wexit')tmp_taken = !(PR[qp] || (AR[EC] u> 1));
        if (PR[qp]) {
            AR[EC] = AR[EC];
            PR[63] = 0;
            rotate_regs();
        } else if (AR[EC] != 0) {
            AR[EC]--;
            PR[63] = 0;
            rotate_regs();
        } else {
            AR[EC] = AR[EC];
            PR[63] = 0;
            CFM.rrb.gr = CFM.rrb.gr;
            CFM.rrb.fr = CFM.rrb.fr;
            CFM.rrb.pr = CFM.rrb.pr;
        }
        break;
}
if (tmp_taken) {
    taken_branch = 1;
    IP = tmp_IP;                          // set the new value for IP
    if ((PSR.it && unimplemented_virtual_address(tmp_IP))
         || (!PSR.it && unimplemented_physical_address(tmp_IP)))
        unimplemented_instruction_address_trap(lower_priv_transition,
                                               tmp_IP);
    if (lower_priv_transition && PSR.lp)
        lower_privilege_transfer_trap();
    if (PSR.tb)
        taken_branch_trap();
}
```

Interruptions: Illegal Operation fault
Disabled Instruction Set Transition fault
Unimplemented Instruction Address trap
Lower-Privilege Transfer trap
Taken Branch trap

Additional Faults on IA-32 target instructions:
IA-32_Exception(GPFault)
Disabled FP Reg Fault if PSR.dfh is 1

Break

Format:

(*qp*) break *imm*$_{21}$	pseudo-op	
(*qp*) break.i *imm*$_{21}$	i_unit_form	I19
(*qp*) break.b *imm*$_{21}$	b_unit_form	B9
(*qp*) break.m *imm*$_{21}$	m_unit_form	M37
(*qp*) break.f *imm*$_{21}$	f_unit_form	F15
(*qp*) break.x *imm*$_{62}$	x_unit_form	X1

Description: A Break Instruction fault is taken. For the i_unit_form, f_unit_form and m_unit_form, the value specified by *imm*$_{21}$ is zero-extended and placed in the Interruption Immediate control register (IIM).

For the b_unit_form, *imm*$_{21}$ is ignored and the value zero is placed in the Interruption Immediate control register (IIM).

For the x_unit_form, the lower 21 bits of the value specified by *imm*$_{62}$ is zero-extended and placed in the Interruption Immediate control register (IIM). The L slot of the bundle contains the upper 41 bits of *imm*$_{62}$.

A `break.i` instruction may be encoded in an MLI-template bundle, in which case the L slot of the bundle is ignored.

This instruction has five forms, each of which can be executed only on a particular execution unit type. The pseudo-op can be used if the unit type to execute on is unimportant.

Operation:

```
if (PR[qp]) {
    if (b_unit_form)
        immediate = 0;
    else if (x_unit_form)
        immediate = zero_ext(imm62, 21);
    else // i_unit_form || m_unit_form || f_unit_form
        immediate = zero_ext(imm21, 21);

    break_instruction_fault(immediate);
}
```

Interruptions: Break Instruction fault

Branch Long

Format:

(*qp*) brl.*btype.bwh.ph.dh* $target_{64}$			X3
(*qp*) brl.*btype.bwh.ph.dh* $b_1 = target_{64}$		call_form	X4
brl.*ph.dh* $target_{64}$		pseudo-op	

Description: A branch condition is evaluated, and either a branch is taken, or execution continues with the next sequential instruction. The execution of a branch logically follows the execution of all previous non-branch instructions in the same instruction group. On a taken branch, execution begins at slot 0.

Long branches are always IP-relative. The $target_{64}$ operand, in assembly, specifies a label to branch to. This is encoded in the long branch instruction as an immediate displacement (imm_{60}) between the target bundle and the bundle containing this instruction ($imm_{60} = target_{64} - \text{IP} >> 4$). The L slot of the bundle contains 39 bits of imm_{60}.

Table 2-9. Long Branch Types

btype	Function	Branch Condition	Target Address
cond or *none*	Conditional branch	Qualifying predicate	IP-relative
call	Conditional procedure call	Qualifying predicate	IP-relative

There is a pseudo-op for long unconditional branches, encoded like a conditional branch (*btype* = cond), with the *qp* field specifying PR 0, and with the *bwh* hint of sptk.

The branch type determines how the branch condition is calculated and whether the branch has other effects (such as writing a link register). For all long branch types, the branch condition is simply the value of the specified predicate register:

- **cond:** If the qualifying predicate is 1, the branch is taken. Otherwise it is not taken.
- **call:** If the qualifying predicate is 1, the branch is taken and several other actions occur:
  - The current values of the Current Frame Marker (CFM), the EC application register and the current privilege level are saved in the Previous Function State application register.
  - The caller's stack frame is effectively saved and the callee is provided with a frame containing only the caller's output region.
  - The rotation rename base registers in the CFM are reset to 0.
  - A return link value is placed in BR b_1.

Read after Write (RAW) and Write after Read (WAR) dependency requirements for long branch instructions are slightly different than for other instructions but are the same as for branch instructions. See page 2-13 for details.

This instruction must be immediately followed by a stop; otherwise its behavior is undefined.

Values for various branch hint completers are the same as for branch instructions. Whether Prediction Strategy hints are shown in Table 2-6, Sequential Prefetch hints are shown in Table 2-7, and Branch Cache Deallocation hints are shown in Table 2-8. See "Branch Prediction Hints" in Volume 1.

Warning: This instruction is not implemented on the Intel Itanium processor, which takes an Illegal Operation fault whenever a long branch instruction is encountered, regardless of whether the branch is taken or not. To support the Intel Itanium processor, the operating system is required to provide an Illegal Operation fault handler which emulates taken and not-taken long branches. Presence of this instruction is indicated by a 1 in the lb bit of CPUID register 4. See "Processor Identification Registers" on p. 3-11 in Volume 1.

Operation:

```
    tmp_IP = IP + (imm60 << 4);                 // determine branch target
if (!followed_by_stop())
    undefined_behavior();
if (!long_branch_implemented())
    illegal_operation_fault();

switch (btype) {
    case 'cond':                                // simple conditional branch
        tmp_taken = PR[qp];
        break;

    case 'call':                                // call saves a return link
        tmp_taken = PR[qp];
        if (tmp_taken) {
            BR[b1] = IP + 16;

            AR[PFS].pfm = CFM;                  // ... and saves the stack frame
            AR[PFS].pec = AR[EC];
            AR[PFS].ppl = PSR.cpl;

            alat_frame_update(CFM.sol, 0);
            rse_preserve_frame(CFM.sol);
            CFM.sof -= CFM.sol;                 // new frame size is size of outs
            CFM.sol = 0;
            CFM.sor = 0;
            CFM.rrb.gr = 0;
            CFM.rrb.fr = 0;
            CFM.rrb.pr = 0;
        }
        break;
}
if (tmp_taken) {
    taken_branch = 1;
    IP = tmp_IP;                                // set the new value for IP
    if ((PSR.it && unimplemented_virtual_address(tmp_IP))
         || (!PSR.it && unimplemented_physical_address(tmp_IP)))
        unimplemented_instruction_address_trap(0,tmp_IP);
    if (PSR.tb)
        taken_branch_trap();
}
```

Interruptions: Illegal Operation fault
Taken Branch trap
Unimplemented Instruction Address trap

Branch Predict

Format:

brp.*ipwh*.*ih* $target_{25}$, tag_{13}	ip_relative_form	B6
brp.*indwh*.*ih* b_2, tag_{13}	indirect_form	B7
brp.ret.*indwh*.*ih* b_2, tag_{13}	return_form, indirect_form	B7

Description: This instruction can be used to provide to hardware early information about a future branch. It has no effect on architectural machine state, and operates as a `nop` instruction except for its performance effects.

The tag_{13} operand, in assembly, specifies the address of the branch instruction to which this prediction information applies. This is encoded in the branch predict instruction as a signed immediate displacement ($timm_9$) between the bundle containing the presaged branch and the bundle containing this instruction ($timm_9 = tag_{13} - \text{IP} >> 4$).

The $target_{25}$ operand, in assembly, specifies the label that the presaged branch will have as its target. This is encoded in the branch predict instruction exactly as in branch instructions, with a signed immediate displacement (imm_{21}) between the target bundle and the bundle containing this instruction ($imm_{21} = target_{25} - \text{IP} >> 4$). The indirect_form can be used to presage an indirect branch. In the indirect_form, the target of the presaged branch is given by BR b_2.

The return_form is used to indicate that the presaged branch will be a return.

Other hints can be given about the presaged branch. Values for various hint completers are shown in the following tables. For more details, refer to Volume 1.

The *ipwh* and *indwh* completers provide information about how best the branch condition should be predicted, when the branch is reached.

Table 2-10. IP-relative Branch Predict Whether Hint

ipwh Completer	IP-relative Branch Predict Whether Hint
sptk	Presaged branch should be predicted Static Taken
loop	Presaged branch will be `br.cloop`, `br.ctop`, or `br.wtop`
exit	Presaged branch will be `br.cexit` or `br.wexit`
dptk	Presaged branch should be predicted Dynamically

Table 2-11. Indirect Branch Predict Whether Hint

indwh Completer	Indirect Branch Predict Whether Hint
sptk	Presaged branch should be predicted Static Taken
dptk	Presaged branch should be predicted Dynamically

The *ih* completer can be used to mark a small number of very important branches (e.g. an inner loop branch). This can signal to hardware to use faster, smaller prediction structures for this information.

Table 2-12. Importance Hint

ih Completer	Branch Predict Importance Hint
none	Less important
imp	More important

Operation:

```
tmp_tag = IP + sign_ext((timm9 << 4), 13);
if (ip_relative_form) {
    tmp_target = IP + sign_ext((imm21 << 4), 25);
    tmp_wh = ipwh;
} else { // indirect_form
    tmp_target = BR[b2];
    tmp_wh = indwh;
}
branch_predict(tmp_wh, ih, return_form, tmp_target, tmp_tag);
```

Interruptions: None

Bank Switch

Format:

bsw.0	zero_form	B8
bsw.1	one_form	B8

Description: This instruction switches to the specified register bank. The zero_form specifies Bank 0 for GR16 to GR31. The one_form specifies Bank 1 for GR16 to GR31. After the bank switch the previous register bank is no longer accessible but does retain its current state. If the new and old register banks are the same, `bsw` is effectively a `nop`, although there may be a performance degradation.

A `bsw` instruction must be the last instruction in an instruction group. Otherwise, an Illegal Operation fault is taken. Instructions in the same instruction group that access GR16 to GR31 reference the previous register bank. Subsequent instruction groups reference the new register bank.

This instruction is privileged.

This instruction cannot be predicated.

Operation:

```
if (!followed_by_stop())
    illegal_operation_fault();

if (PSR.cpl != 0)
    privileged_operation_fault(0);

if (zero_form)
    PSR.bn = 0;
else // one_form
    PSR.bn = 1;
```

Interruptions: Illegal Operation fault Privileged Operation fault

Serialization: This instruction does not require any additional instruction or data serialization operation. The bank switch occurs synchronously with its execution.

Speculation Check

Format:	(*qp*) chk.s r_2, $target_{25}$	pseudo-op	
	(*qp*) chk.s.i r_2, $target_{25}$	control_form, i_unit_form, gr_form	I20
	(*qp*) chk.s.m r_2, $target_{25}$	control_form, m_unit_form, gr_form	M20
	(*qp*) chk.s f_2, $target_{25}$	control_form, fr_form	M21
	(*qp*) chk.a.*aclr* r_1, $target_{25}$	data_form, gr_form	M22
	(*qp*) chk.a.*aclr* f_1, $target_{25}$	data_form, fr_form	M23

Description: The result of a control- or data-speculative calculation is checked for success or failure. If the check fails, a branch to $target_{25}$ is taken.

In the control_form, success is determined by a NaT indication for the source register. If the NaT bit corresponding to GR r_2 is 1 (in the gr_form), or FR f_2 contains a NaTVal (in the fr_form), the check fails.

In the data_form, success is determined by the ALAT. The ALAT is queried using the general register specifier r_1 (in the gr_form), or the floating-point register specifier f_1 (in the fr_form). If no ALAT entry matches, the check fails. An implementation may optionally cause the check to fail independent of whether an ALAT entry matches. A `chk.a` with general register specifier r0 or floating-point register specifiers f0 or f1 always fails.

The $target_{25}$ operand, in assembly, specifies a label to branch to. This is encoded in the instruction as a signed immediate displacement (imm_{21}) between the target bundle and the bundle containing this instruction ($imm_{21} = target_{25} - \text{IP} >> 4$).

The branching behavior of this instruction can be optionally unimplemented. If the instruction would have branched, and the branching behavior is not implemented, then a Speculative Operation fault is taken and the value specified by imm_{21} is zero-extended and placed in the Interruption Immediate control register (IIM). The fault handler emulates the branch by sign-extending the IIM value, adding it to IIP and returning.

The control_form of this instruction for checking general registers can be encoded on either an I-unit or an M-unit. The pseudo-op can be used if the unit type to execute on is unimportant.

For the data_form, if an ALAT entry matches, the matching ALAT entry can be optionally invalidated, based on the value of the *aclr* completer (See Table 2-13).

Table 2-13. ALAT Clear Completer

aclr Completer	Effect on ALAT
clr	Invalidate matching ALAT entry
nc	Don't invalidate

Note that if the *clr* value of the *aclr* completer is used and the check succeeds, the matching ALAT entry is invalidated. However, if the check fails (which may happen even if there is a matching ALAT entry), any matching ALAT entry may optionally be invalidated, but this is not required. Recovery code for data speculation, therefore, cannot rely on the absence of a matching ALAT entry.

Operation:

```
if (PR[qp]) {
    if (control_form) {
        if (fr_form && (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0)))
            disabled_fp_register_fault(tmp_isrcode, 0);
        check_type = gr_form ? CHKS_GENERAL : CHKS_FLOAT;
        fail = (gr_form && GR[r2].nat) || (fr_form && FR[f2] == NATVAL);
    } else {                                            // data_form
        if (gr_form) {
            reg_type   = GENERAL;
            check_type = CHKA_GENERAL;
            alat_index = r1;
            always_fail = (alat_index == 0);
        } else {                                        // fr_form
            reg_type   = FLOAT;
            check_type = CHKA_FLOAT;
            alat_index = f1;
            always_fail = ((alat_index == 0) || (alat_index == 1));
        }
        fail = (always_fail || (!alat_cmp(reg_type, alat_index)));
    }
    if (fail) {
        if (check_branch_implemented(check_type)) {
            taken_branch = 1;
            IP = IP + sign_ext((imm21 << 4), 25);
            if ((PSR.it && unimplemented_virtual_address(IP))
                || (!PSR.it && unimplemented_physical_address(IP)))
                unimplemented_instruction_address_trap(0, IP);
            if (PSR.tb)
                taken_branch_trap();
        } else
            speculation_fault(check_type, zero_ext(imm21, 21));
    } else if (data_form && (aclr == 'clr'))
        alat_inval_single_entry(reg_type, alat_index);
}
```

Interruptions: Disabled Floating-point Register fault
Speculative Operation fault
Unimplemented Instruction Address trap
Taken Branch trap

Clear RRB

Format:

clrrrb	all_form	B8
clrrrb.pr	pred_form	B8

Description: In the all_form, the register rename base registers (CFM.rrb.gr, CFM.rrb.fr, and CFM.rrb.pr) are cleared. In the pred_form, the single register rename base register for the predicates (CFM.rrb.pr) is cleared.

This instruction must be the last instruction in an instruction group, or an Illegal Operation fault is taken.

This instruction cannot be predicated.

Operation:

```
if (!followed_by_stop())
    illegal_operation_fault();

if (all_form) {
    CFM.rrb.gr = 0;
    CFM.rrb.fr = 0;
    CFM.rrb.pr = 0;
} else { // pred_form
    CFM.rrb.pr = 0;
}
```

Interruptions: Illegal Operation fault

Compare

Format:

(*qp*) cmp.*crel.ctype* $p_1, p_2 = r_2, r_3$	register_form	A6
(*qp*) cmp.*crel.ctype* $p_1, p_2 = imm_8, r_3$	imm8_form	A8
(*qp*) cmp.*crel.ctype* p_1, p_2 = r0, r_3	parallel_inequality_form	A7
(*qp*) cmp.*crel.ctype* $p_1, p_2 = r_3$, r0	pseudo-op	

Description: The two source operands are compared for one of ten relations specified by *crel*. This produces a boolean result which is 1 if the comparison condition is true, and 0 otherwise. This result is written to the two predicate register destinations, p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*.

The compare types describe how the predicate targets are updated based on the result of the comparison. The normal type simply writes the compare result to one target, and the complement to the other. The parallel types update the targets only for a particular comparison result. This allows multiple simultaneous OR-type or multiple simultaneous AND-type compares to target the same predicate register.

The unc type is special in that it first initializes both predicate targets to 0, *independent of the qualifying predicate*. It then operates the same as the normal type. The behavior of the compare types is described in Table 2-14. A blank entry indicates the predicate target is left unchanged.

Table 2-14. Comparison Types

ctype	pseudo-op of	PR[*qp*]==0		PR[*qp*]==1					
				result==0, No Source NaTs		result==1, No Source NaTs		One or More Source NaTs	
		PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]
none				0	1	1	0	0	0
unc		0	0	0	1	1	0	0	0
or						1	1		
and				0	0			0	0
or.andcm						1	0		
orcm	or			1	1				
andcm	and					0	0	0	0
and.orcm	or.andcm			0	1				

In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the sign-extended imm_8 encoding field; and in the parallel_inequality_form the first operand must be GR 0. The parallel_inequality_form is only used when the compare type is one of the parallel types, and the relation is an inequality (>, >=, <, <=). See below.

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is 1, or if the compare type is unc.

Of the ten relations, not all are directly implemented in hardware. Some are actually pseudo-ops. For these, the assembler simply switches the source operand specifiers and/or switches the predicate target specifiers and uses an implemented relation. For some of the pseudo-op compares in the imm8_form, the assembler subtracts 1 from the immediate value, making the allowed immediate range slightly different. Of the six parallel compare types, three of the types are actually

pseudo-ops. The assembler simply uses the negative relation with an implemented type. The implemented relations and how the pseudo-ops map onto them are shown in Table 2-15 (for normal and unc type compares), and Table 2-16 (for parallel type compares).

Table 2-15. 64-bit Comparison Relations for Normal and unc Compares

crel	Compare Relation (*a* rel *b*)	Register Form is a Pseudo-op of	Immediate Form is a Pseudo-op of	Immediate Range
eq	*a* == *b*			-128 .. 127
ne	*a* != *b*	eq $p_1 \leftrightarrow p_2$	eq $p_1 \leftrightarrow p_2$	-128 .. 127
lt	*a* < *b* signed			-128 .. 127
le	*a* <= *b*	lt a ↔ b $p_1 \leftrightarrow p_2$	lt a-1	-127 .. 128
gt	*a* > *b*	lt a ↔ b	lt a-1 $p_1 \leftrightarrow p_2$	-127 .. 128
ge	*a* >= *b*	lt $p_1 \leftrightarrow p_2$	lt $p_1 \leftrightarrow p_2$	-128 .. 127
ltu	*a* < *b* unsigned			0 .. 127, 2^{64}-128 .. 2^{64}-1
leu	*a* <= *b*	ltu a ↔ b $p_1 \leftrightarrow p_2$	ltu a-1	1 .. 128, 2^{64}-127 .. 2^{64}
gtu	*a* > *b*	ltu a ↔ b	ltu a-1 $p_1 \leftrightarrow p_2$	1 .. 128, 2^{64}-127 .. 2^{64}
geu	*a* >= *b*	ltu $p_1 \leftrightarrow p_2$	ltu $p_1 \leftrightarrow p_2$	0 .. 127, 2^{64}-128 .. 2^{64}-1

Table 2-16. 64-bit Comparison Relations for Parallel Compares

crel	Compare Relation (*a* rel *b*)	Register Form is a Pseudo-op of	Immediate Range
eq	*a* == *b*		-128 .. 127
ne	*a* != *b*		-128 .. 127
lt	0 < *b* signed		no immediate forms
lt	*a* < 0	gt a ↔ b	
le	0 <= *b*		
le	*a* <= 0	ge a ↔ b	
gt	0 > *b*		
gt	*a* > 0	lt a ↔ b	
ge	0 >= *b*		
ge	*a* >= 0	le a ↔ b	

The parallel compare types can be used only with a restricted set of relations and operands. They can be used with equal and not-equal comparisons between two registers or between a register and an immediate, or they can be used with inequality comparisons between a register and GR 0. Unsigned relations are not provided, since they are not of much use when one of the operands is zero. For the parallel inequality comparisons, hardware only directly implements the ones where the first operand (GR r_2) is GR 0. Comparisons where the second operand is GR 0 are pseudo-ops for which the assembler switches the register specifiers and uses the opposite relation.

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    tmp_nat = (register_form ? GR[r2].nat : 0) || GR[r3].nat;
    if (register_form)
        tmp_src = GR[r2];
    else if (imm8_form)
        tmp_src = sign_ext(imm8, 8);
    else // parallel_inequality_form
        tmp_src = 0;

    if      (crel == 'eq')  tmp_rel = tmp_src == GR[r3];
    else if (crel == 'ne')  tmp_rel = tmp_src != GR[r3];
    else if (crel == 'lt')  tmp_rel = lesser_signed(tmp_src,  GR[r3]);
    else if (crel == 'le')  tmp_rel = lesser_equal_signed(tmp_src, GR[r3]);
    else if (crel == 'gt')  tmp_rel = greater_signed(tmp_src,  GR[r3]);
    else if (crel == 'ge')  tmp_rel = greater_equal_signed(tmp_src, GR[r3]);
    else if (crel == 'ltu') tmp_rel = lesser(tmp_src, GR[r3]);
    else if (crel == 'leu') tmp_rel = lesser_equal(tmp_src, GR[r3]);
    else if (crel == 'gtu') tmp_rel = greater(tmp_src, GR[r3]);
    else                    tmp_rel = greater_equal(tmp_src, GR[r3]);//'geu'

    switch (ctype) {
        case 'and':                                     // and-type compare
            if (tmp_nat || !tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                                      // or-type compare
            if (!tmp_nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                                // or.andcm-type compare
            if (!tmp_nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case 'unc':                                     // unc-type compare
        default:                                        // normal compare
            if (tmp_nat) {
                PR[p1] = 0;
                PR[p2] = 0;
            } else {
                PR[p1] = tmp_rel;
                PR[p2] = !tmp_rel;
            }
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Interruptions: Illegal Operation fault

Compare Word

Format:

(*qp*) cmp4.*crel*.*ctype* $p_1, p_2 = r_2, r_3$	register_form	A6
(*qp*) cmp4.*crel*.*ctype* $p_1, p_2 = imm_8, r_3$	imm8_form	A8
(*qp*) cmp4.*crel*.*ctype* $p_1, p_2 =$ r0, r_3	parallel_inequality_form	A7
(*qp*) cmp4.*crel*.*ctype* $p_1, p_2 = r_3$, r0	pseudo-op	

Description: The least significant 32 bits from each of two source operands are compared for one of ten relations specified by *crel*. This produces a boolean result which is 1 if the comparison condition is true, and 0 otherwise. This result is written to the two predicate register destinations, p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-14 on page 2-26.

In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the sign-extended imm_8 encoding field; and in the parallel_inequality_form the first operand must be GR 0. The parallel_inequality_form is only used when the compare type is one of the parallel types, and the relation is an inequality (>, >=, <, <=). See the Compare instruction and Table 2-16 on page 2-27.

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is 1, or if the compare type is unc.

Of the ten relations, not all are directly implemented in hardware. Some are actually pseudo-ops. See the Compare instruction and Table 2-15 and Table 2-16 on page 2-27. The range for immediates is given below.

Table 2-17. Immediate Range for 32-bit Compares

crel	Compare Relation (*a* rel *b*)		Immediate Range
eq	*a* == *b*		-128 .. 127
ne	*a* != *b*		-128 .. 127
lt	*a* < *b*	signed	-128 .. 127
le	*a* <= *b*		-127 .. 128
gt	*a* > *b*		-127 .. 128
ge	*a* >= *b*		-128 .. 127
ltu	*a* < *b*	unsigned	0 .. 127, 2^{32}-128 .. 2^{32}-1
leu	*a* <= *b*		1 .. 128, 2^{32}-127 .. 2^{32}
gtu	*a* > *b*		1 .. 128, 2^{32}-127 .. 2^{32}
geu	*a* >= *b*		0 .. 127, 2^{32}-128 .. 2^{32}-1

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    tmp_nat = (register_form ? GR[r2].nat : 0) || GR[r3].nat;

    if (register_form)
        tmp_src = GR[r2];
    else if (imm8_form)
        tmp_src = sign_ext(imm8, 8);
    else // parallel_inequality_form
        tmp_src = 0;

    if      (crel == 'eq')  tmp_rel = tmp_src{31:0} == GR[r3]{31:0};
    else if (crel == 'ne')  tmp_rel = tmp_src{31:0} != GR[r3]{31:0};
    else if (crel == 'lt')
        tmp_rel = lesser_signed(sign_ext(tmp_src, 32),
                                sign_ext(GR[r3], 32));
    else if (crel == 'le')
        tmp_rel = lesser_equal_signed(sign_ext(tmp_src, 32),
                                sign_ext(GR[r3], 32));
    else if (crel == 'gt')
        tmp_rel = greater_signed(sign_ext(tmp_src, 32),
                                sign_ext(GR[r3], 32));
    else if (crel == 'ge')
        tmp_rel = greater_equal_signed(sign_ext(tmp_src, 32),
                                sign_ext(GR[r3], 32));
    else if (crel == 'ltu')
        tmp_rel = lesser(zero_ext(tmp_src, 32),
                                zero_ext(GR[r3], 32));
    else if (crel == 'leu')
        tmp_rel = lesser_equal(zero_ext(tmp_src, 32),
                                zero_ext(GR[r3], 32));
    else if (crel == 'gtu')
        tmp_rel = greater(zero_ext(tmp_src, 32),
                                zero_ext(GR[r3], 32));
    else        // 'geu'
        tmp_rel = greater_equal(zero_ext(tmp_src, 32),
                                zero_ext(GR[r3], 32));

    switch (ctype) {
        case 'and':                                     // and-type compare
            if (tmp_nat || !tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                                      // or-type compare
            if (!tmp_nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                                // or.andcm-type compare
            if (!tmp_nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case 'unc':                                     // unc-type compare
        default:                                        // normal compare
            if (tmp_nat) {
```

```
                PR[p1] = 0;
                PR[p2] = 0;
            } else {
                PR[p1] = tmp_rel;
                PR[p2] = !tmp_rel;
            }
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Interruptions: Illegal Operation fault

Compare And Exchange

Format: (*qp*) cmpxchg*sz.sem.ldhint* r_1 = [r_3], r_2, ar.ccv M16

Description: A value consisting of *sz* bytes is read from memory starting at the address specified by the value in GR r_3. The value is zero extended and compared with the contents of the `cmpxchg` Compare Value application register (AR[CCV]). If the two are equal, then the least significant *sz* bytes of the value in GR r_2 are written to memory starting at the address specified by the value in GR r_3. The zero-extended value read from memory is placed in GR r_1 and the NaT bit corresponding to GR r_1 is cleared.

The values of the *sz* completer are given in Table 2-18. The *sem* completer specifies the type of semaphore operation. These operations are described in Table 2-19. See Volume 1 and Volume 2 for details on memory ordering.

Table 2-18. Memory Compare and Exchange Size

sz Completer	Bytes Accessed
1	1
2	2
4	4
8	8

Table 2-19. Compare and Exchange Semaphore Types

sem Completer	Ordering Semantics	Semaphore Operation
acq	Acquire	The memory read/write is made visible prior to all subsequent data memory accesses.
rel	Release	The memory read/write is made visible after all previous data memory accesses.

If the address specified by the value in GR r_3 is not naturally aligned to the size of the value being accessed in memory, an Unaligned Data Reference fault is taken independent of the state of the User Mask alignment checking bit, UM.ac (PSR.ac in the Processor Status Register).

The memory read and write are guaranteed to be atomic.

Both read and write access privileges for the referenced page are required. The write access privilege check is performed whether or not the memory write is performed.

This instruction is only supported to cacheable pages with write-back write policy. Accesses to NaTPages cause a Data NaT Page Consumption fault. Accesses to pages with other memory attributes cause an Unsupported Data Reference fault.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 2-32 on page 2-127. Locality hints do not affect program functionality and may be ignored by the implementation.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(SEMAPHORE);

    paddr = tlb_translate(GR[r3], sz, SEMAPHORE, PSR.cpl, &mattr,
                          &tmp_unused);

    if (!ma_supports_semaphores(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

    if (sem == 'acq')
        val = mem_xchg_cond(AR[CCV], GR[r2], paddr, sz, UM.be, mattr,
                            ACQUIRE, ldhint);
    else // 'rel'
        val = mem_xchg_cond(AR[CCV], GR[r2], paddr, sz, UM.be, mattr,
                            RELEASE, ldhint);
    val = zero_ext(val, sz * 8);

    if (AR[CCV] == val)
        alat_inval_multiple_entries(paddr, sz);

    GR[r1] = val;
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

Cover Stack Frame

Format: cover B8

Description: A new stack frame of zero size is allocated which does not include any registers from the previous frame (as though all output registers in the previous frame had been locals). The register rename base registers are reset. If interruption collection is disabled (PSR.ic is zero), then the old value of the Current Frame Marker (CFM) is copied to the Interruption Function State register (IFS), and IFS.v is set to one.

A `cover` instruction must be the last instruction in an instruction group. Otherwise, an Illegal Operation fault is taken.

This instruction cannot be predicated.

Operation:

```
if (!followed_by_stop())
    illegal_operation_fault();

alat_frame_update(CFM.sof, 0);
rse_preserve_frame(CFM.sof);
if (PSR.ic == 0) {
    CR[IFS].ifm = CFM;
    CR[IFS].v = 1;
}

CFM.sof = 0;
CFM.sol = 0;
CFM.sor = 0;
CFM.rrb.gr = 0;
CFM.rrb.fr = 0;
CFM.rrb.pr = 0;
```

Interruptions: Illegal Operation fault

Compute Zero Index

Format:

(*qp*) czx1.l $r_1 = r_3$	one_byte_form, left_form	I29
(*qp*) czx1.r $r_1 = r_3$	one_byte_form, right_form	I29
(*qp*) czx2.l $r_1 = r_3$	two_byte_form, left_form	I29
(*qp*) czx2.r $r_1 = r_3$	two_byte_form, right_form	I29

Description: GR r_3 is scanned for a zero element. The element is either an 8-bit aligned byte (one_byte_form) or a 16-bit aligned pair of bytes (two_byte_form). The index of the first zero element is placed in GR r_1. If there are no zero elements in GR r_3, a default value is placed in GR r_1. Table 2-20 gives the possible result values. In the left_form, the source is scanned from most significant element to least significant element, and in the right_form it is scanned from least significant element to most significant element.

Table 2-20. Result Ranges for czx

Size	Element Width	Range of Result if Zero Element Found	Default Result if No Zero Element Found
1	8 bit	0-7	8
2	16 bit	0-3	4

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {
        if (left_form) {            // scan from most significant down
            if      ((GR[r3] & 0xff00000000000000) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x00ff000000000000) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x0000ff0000000000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0x000000ff00000000) == 0) GR[r1] = 3;
            else if ((GR[r3] & 0x00000000ff000000) == 0) GR[r1] = 4;
            else if ((GR[r3] & 0x0000000000ff0000) == 0) GR[r1] = 5;
            else if ((GR[r3] & 0x000000000000ff00) == 0) GR[r1] = 6;
            else if ((GR[r3] & 0x00000000000000ff) == 0) GR[r1] = 7;
            else GR[r1] = 8;
        } else { // right_form      scan from least significant up
            if      ((GR[r3] & 0x00000000000000ff) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x000000000000ff00) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x0000000000ff0000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0x00000000ff000000) == 0) GR[r1] = 3;
            else if ((GR[r3] & 0x000000ff00000000) == 0) GR[r1] = 4;
            else if ((GR[r3] & 0x0000ff0000000000) == 0) GR[r1] = 5;
            else if ((GR[r3] & 0x00ff000000000000) == 0) GR[r1] = 6;
            else if ((GR[r3] & 0xff00000000000000) == 0) GR[r1] = 7;
            else GR[r1] = 8;
        }
    } else { // two_byte_form
        if (left_form) {            // scan from most significant down
            if      ((GR[r3] & 0xffff000000000000) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x0000ffff00000000) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x00000000ffff0000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0x000000000000ffff) == 0) GR[r1] = 3;
            else GR[r1] = 4;
        } else { // right_form      scan from least significant up
            if      ((GR[r3] & 0x000000000000ffff) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x00000000ffff0000) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x0000ffff00000000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0xffff000000000000) == 0) GR[r1] = 3;
```

```
            else GR[r1] = 4;
        }
    }
    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Deposit

Format:

(*qp*) dep $r_1 = r_2, r_3, pos_6, len_4$	merge_form, register_form	I15
(*qp*) dep $r_1 = imm_1, r_3, pos_6, len_6$	merge_form, imm_form	I14
(*qp*) dep.z $r_1 = r_2, pos_6, len_6$	zero_form, register_form	I12
(*qp*) dep.z $r_1 = imm_8, pos_6, len_6$	zero_form, imm_form	I13

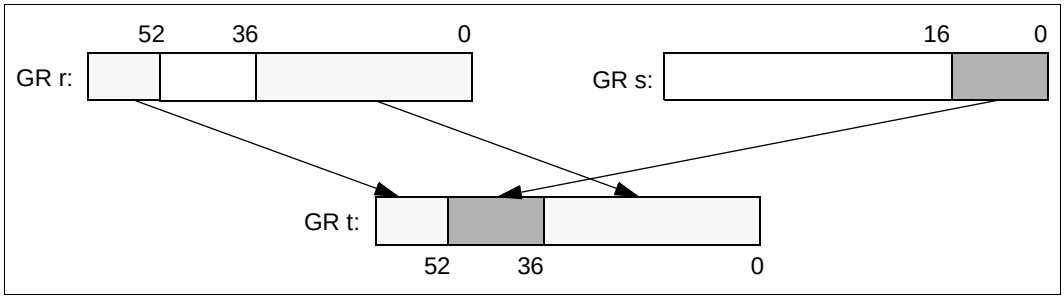
Description: In the merge_form, a right justified bit field taken from the first source operand is deposited into the value in GR r_3 at an arbitrary bit position and the result is placed in GR r_1. In the register_form the first source operand is GR r_2; and in the imm_form it is the sign-extended value specified by imm_1 (either all ones or all zeroes). The deposited bit field begins at the bit position specified by the pos_6 immediate and extends to the left (towards the most significant bit) a number of bits specified by the *len* immediate. Note that *len* has a range of 1-16 in the register_form and 1-64 in the imm_form. The pos_6 immediate has a range of 0 to 63.

In the zero_form, a right justified bit field taken from either the value in GR r_2 (in the register_form) or the sign-extended value in imm_8 (in the imm_form) is deposited into GR r_1 and all other bits in GR r_1 are cleared to zero. The deposited bit field begins at the bit position specified by the pos_6 immediate and extends to the left (towards the most significant bit) a number of bits specified by the *len* immediate. The *len* immediate has a range of 1-64 and the pos_6 immediate has a range of 0 to 63.

In the event that the deposited bit field extends beyond bit 63 of the target, i.e. $len + pos_6 > 64$, the most significant $len + pos_6 - 64$ bits of the deposited bit field are truncated. The *len* immediate is encoded as *len* minus 1 in the instruction.

The operation of `dep t = s, r, 36, 16` is illustrated in Figure 2-5.

Figure 2-5. Deposit Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (imm_form) {
        tmp_src = (merge_form ? sign_ext(imm1,1) : sign_ext(imm8, 8));
        tmp_nat = merge_form ? GR[r3].nat : 0;
        tmp_len = len6 ;
    } else {                                          // register_form
        tmp_src = GR[r2];
        tmp_nat = (merge_form ? GR[r3].nat : 0) || GR[r2].nat;
        tmp_len = merge_form ? len4 : len6 ;
    }
    if (pos6 + tmp_len u> 64)
        tmp_len = 64 - pos6;

    if (merge_form)
        GR[r1] = GR[r3];
    else // zero_form
        GR[r1] = 0;

    GR[r1]{(pos6 + tmp_len - 1):pos6} = tmp_src{(tmp_len - 1):0};
    GR[r1].nat = tmp_nat;
}
```

Interruptions: Illegal Operation fault

Enter Privileged Code

Format: epc B8

Description: This instruction increases the privilege level. The new privilege level is given by the TLB entry for the page containing this instruction. This instruction can be used to implement calls to higher-privileged routines without the overhead of an interruption.

Before increasing the privilege level, a check is performed. The PFS.ppl (previous privilege level) is checked to ensure that it is not more privileged than the current privilege level. If this check fails, the instruction takes an Illegal Operation fault.

If the check succeeds, then the privilege is increased as follows:

- If instruction address translation is enabled and the page containing the `epc` instruction has execute-only page access rights and the privilege level assigned to the page is higher than (numerically less than) the current privilege level, then the current privilege level is set to the privilege level field in the translation for the page containing the `epc` instruction. This instruction can promote but cannot demote, and the new privilege comes from the TLB entry.

  If instruction address translation is disabled, then the current privilege level is set to 0 (most privileged).

  Instructions after the `epc` in the same instruction group may be executed at the old privilege level or the new, higher privilege level. Instructions in subsequent instruction groups will be executed at the new, higher privilege level.
- If the page containing the `epc` instruction has any other access rights besides execute-only, or if the privilege level assigned to the page is lower or equal to (numerically greater than or equal to) the current privilege level, then no action is taken (the current privilege level is unchanged).

Note that the ITLB is actually only read once, at instruction fetch. Information from the access rights and privilege level fields from the translation is then used in executing this instruction.

This instruction cannot be predicated.

Operation:

```
if (AR[PFS].ppl u< PSR.cpl)
    illegal_operation_fault();

if (PSR.it)
    PSR.cpl = tlb_enter_privileged_code();
else
    PSR.cpl = 0;
```

Interruptions: Illegal Operation fault

Extract

Format:

(*qp*) extr $r_1 = r_3, pos_6, len_6$	signed_form	I11
(*qp*) extr.u $r_1 = r_3, pos_6, len_6$	unsigned_form	I11

Description: A field is extracted from GR r_3, either zero extended or sign extended, and placed right-justified in GR r_1. The field begins at the bit position given by the second operand and extends len_6 bits to the left. The bit position where the field begins is specified by the pos_6 immediate. The extracted field is sign extended in the signed_form or zero extended in the unsigned_form. The sign is taken from the most significant bit of the extracted field. If the specified field extends beyond the most significant bit of GR r_3, the sign is taken from the most significant bit of GR r_3. The immediate value len_6 can be any number in the range 1 to 64, and is encoded as len_6-1 in the instruction. The immediate value pos_6 can be any value in the range 0 to 63.

The operation of `extr t = r, 7, 50` is illustrated in Figure 2-6.

Figure 2-6. Extract Example

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_len = len6;

    if (pos6 + tmp_len u> 64)
        tmp_len = 64 - pos6;

    if (unsigned_form)
        GR[r1] = zero_ext(shift_right_unsigned(GR[r3], pos6), tmp_len);
    else // signed_form
        GR[r1] = sign_ext(shift_right_unsigned(GR[r3], pos6), tmp_len);

    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Floating-Point Absolute Value

Format: (*qp*) fabs $f_1 = f_3$ pseudo-op of: (*qp*) fmerge.s f_1 = f0, f_3

Description: The absolute value of the value in FR f_3 is computed and placed in FR f_1.

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See "Floating-Point Merge" on p. 2-63.

Floating-Point Add

Format: (*qp*) fadd.*pc*.*sf* $f_1 = f_3, f_2$ pseudo-op of: (*qp*) fma.*pc*.*sf* $f_1 = f_3$, f1, f_2

Description: FR f_3 and FR f_2 are added (computed to infinite precision), rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*, and placed in FR f_1. If either FR f_3 or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-21. The mnemonic values for *sf* are given in Table 2-22. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Table 2-21. Specified *pc* Mnemonic Values

pc Mnemonic	Precision Specified
.s	single
.d	double
none	dynamic (i.e. use pc value in status field)

Table 2-22. *sf* Mnemonic Values

sf Mnemonic	Status Field Accessed
.s0 or *none*	sf0
.s1	sf1
.s2	sf2
.s3	sf3

Operation: See "Floating-Point Multiply Add" on p. 2-61.

Floating-Point Absolute Maximum

Format: (*qp*) famax.*sf* $f_1 = f_2, f_3$ F8

Description: The operand with the larger absolute value is placed in FR f_1. If the magnitude of FR f_2 equals the magnitude of FR f_3, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaN, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as the `fcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_right = fp_reg_read(FR[f2]);
        tmp_left = fp_reg_read(FR[f3]);
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        FR[f1] = tmp_bool_res ? FR[f2] : FR[f3];

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

Floating-Point Absolute Minimum

Format: (*qp*) famin.*sf* $f_1 = f_2, f_3$ F8

Description: The operand with the smaller absolute value is placed in FR f_1. If the magnitude of FR f_2 equals the magnitude of FR f_3, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaN, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as the `fcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_left = fp_reg_read(FR[f2]);
        tmp_right = fp_reg_read(FR[f3]);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        FR[f1] = tmp_bool_res ? FR[f2] : FR[f3];

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

Floating-Point Logical And

Format: (*qp*) fand $f_1 = f_2, f_3$ F9

Description: The bit-wise logical AND of the significand fields of FR f_2 and FR f_3 is computed. The resulting value is stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand & FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }
    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Floating-Point And Complement

Format: (*qp*) fandcm $f_1 = f_2, f_3$ F9

Description: The bit-wise logical AND of the significand field of FR f_2 with the bit-wise complemented significand field of FR f_3 is computed. The resulting value is stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand & ~FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }
    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Flush Cache

Format: (*qp*) fc r_3 M28

Description: The cache line associated with the address specified by the value of GR r_3 is invalidated from all levels of the processor cache hierarchy. The invalidation is broadcast throughout the coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory it is written to memory before invalidation.

The line size affected is at least 32-bytes (aligned on a 32-byte boundary). An implementation may flush a larger region.

When executed at privilege level 0, `fc` performs no access rights or protection key checks. At other privilege levels, `fc` performs access rights checks as if it were a 1-byte read, but no protection key checks (regardless of PSR.pk).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. This instruction can be used to remove a range of addresses from the cache by first changing the memory attribute to non-cacheable and then flushing the range.

This instruction follows data dependency rules; it is ordered with respect to preceding and following memory references to the same line. `fc` has data dependencies in the sense that any prior stores by this processor will be included in the data written back to memory. `fc` is an unordered operation, and is not affected by a memory fence (`mf`) instruction. It is ordered with respect to the `sync.i` instruction.

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS|FC|READ;
    if (GR[r3].nat)
        register_nat_consumption_fault(itype);
    tmp_paddr = tlb_translate_nonaccess(GR[r3], itype);
    mem_flush(tmp_paddr);
}
```

Interruptions:

Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Access Rights fault

Floating-Point Check Flags

Format: (*qp*) fchkf.*sf* *target*$_{25}$ F14

Description: The flags in FPSR.*sf*.flags are compared with FPSR.s0.flags and FPSR.traps. If any flags set in FPSR.*sf*.flags correspond to FPSR.traps which are enabled, or if any flags set in FPSR.*sf*.flags are not set in FPSR.s0.flags, then a branch to *target*$_{25}$ is taken.

The *target*$_{25}$ operand, specifies a label to branch to. This is encoded in the instruction as a signed immediate displacement (*imm*$_{21}$) between the target bundle and the bundle containing this instruction (*imm*$_{21}$ = *target*$_{25}$ – IP >> 4).

The branching behavior of this instruction can be optionally unimplemented. If the instruction would have branched, and the branching behavior is not implemented, then a Speculative Operation fault is taken and the value specified by *imm*$_{21}$ is zero-extended and placed in the Interruption Immediate control register (IIM). The fault handler emulates the branch by sign-extending the IIM value, adding it to IIP and returning.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    switch (sf) {
        case 's0':
            tmp_flags = AR[FPSR].sf0.flags;
            break;
        case 's1':
            tmp_flags = AR[FPSR].sf1.flags;
            break;
        case 's2':
            tmp_flags = AR[FPSR].sf2.flags;
            break;
        case 's3':
            tmp_flags = AR[FPSR].sf3.flags;
            break;
    }
    if ((tmp_flags & ~AR[FPSR].traps) || (tmp_flags & ~AR[FPSR].sf0.flags)) {
        if (check_branch_implemented(FCHKF)) {
            taken_branch = 1;
            IP = IP + sign_ext((imm21 << 4), 25);
            if ((PSR.it && unimplemented_virtual_address(IP))
                || (!PSR.it && unimplemented_physical_address(IP)))
                unimplemented_instruction_address_trap(0, IP);
            if (PSR.tb)
                taken_branch_trap();
        } else
            speculation_fault(FCHKF, zero_ext(imm21, 21));
    }
}
```

FP Exceptions: None

Interruptions: Speculative Operation fault
Taken Branch trap
Unimplemented Instruction Address trap

Floating-Point Class

Format: (*qp*) fclass.*fcrel*.*fctype* $p_1, p_2 = f_2, fclass_9$ F5

Description: The contents of FR f_2 are classified according to the $fclass_9$ completer as shown in Table 2-24. This produces a boolean result based on whether the contents of FR f_2 agrees with the floating-point number format specified by $fclass_9$, as specified by the *fcrel* completer. This result is written to the two predicate register destinations, p_1 and p_2. The result written to the destinations is determined by the compare type specified by *fctype*.

The allowed types are Normal (or *none*) and unc. See Table 2-25 on page 2-52. The assembly syntax allows the specification of membership or non-membership and the assembler swaps the target predicates to achieve the desired effect.

Table 2-23. Floating-point Class Relations

fcrel	Test Relation
m	FR f_2 agrees with the pattern specified by $fclass_9$ (is a member)
nm	FR f_2 does not agree with the pattern specified by $fclass_9$ (is not a member)

A number agrees with the pattern specified by $fclass_9$ if:

- The number is NaTVal and $fclass_9$ {8} is 1, or
- The number is a quiet NaN and $fclass_9$ {7} is 1, or
- The number is a signaling NaN and $fclass_9$ {6} is 1, or
- The sign of the number agrees with the sign specified by one of the two low-order bits of $fclass_9$, and the type of the number (disregarding the sign) agrees with the number-type specified by the next 4 bits of $fclass_9$, as shown in Table 2-24.

Note: An $fclass_9$ of 0x1FF is equivalent to testing for any supported operand. The class names used in Table 2-24 are defined in Table 5-2 on page 5-3.

Table 2-24. Floating-point Classes

$fclass_9$	Class	Mnemonic
Either these cases can be tested for		
0x0100	NaTVal	@nat
0x080	Quiet NaN	@qnan
0x040	Signaling NaN	@snan
or the OR of the following two cases		
0x001	Positive	@pos
0x002	Negative	@neg
AND'ed with OR of the following 4 cases		
0x004	Zero	@zero
0x008	Unnormalized	@unorm
0x010	Normalized	@norm
0x020	Infinity	@inf

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    tmp_rel = ((fclass9{0} && !FR[f2].sign || fclass9{1} && FR[f2].sign)
                  &&  ((fclass9{2} && fp_is_zero(FR[f2]))||
                       (fclass9{3} && fp_is_unorm(FR[f2])) ||
                       (fclass9{4} && fp_is_normal(FR[f2])) ||
                       (fclass9{5} && fp_is_inf(FR[f2]))
                      )
                  )
              || (fclass9{6} && fp_is_snan(FR[f2]))
              || (fclass9{7} && fp_is_qnan(FR[f2]))
              || (fclass9{8} && fp_is_natval(FR[f2]));

    tmp_nat = fp_is_natval(FR[f2]) && (!fclass9{8});

    if (tmp_nat) {
        PR[p1] = 0;
        PR[p2] = 0;
    } else {
        PR[p1] = tmp_rel;
        PR[p2] = !tmp_rel;
    }
} else {
    if (fctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Floating-Point Clear Flags

Format: (*qp*) fclrf.*sf* F13

Description: The status field's 6-bit flags field is reset to zero.
The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:
```
if (PR[qp]) {
    fp_set_sf_flags(sf, 0);
}
```

FP Exceptions: None

Interruptions: None

Floating-Point Compare

Format: (*qp*) fcmp.*frel*.*fctype*.*sf* $p_1, p_2 = f_2, f_3$ F4

Description: The two source operands are compared for one of twelve relations specified by *frel*. This produces a boolean result which is 1 if the comparison condition is true, and 0 otherwise. This result is written to the two predicate register destinations, p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *fctype*. The allowed types are Normal (or *none*) and unc.

Table 2-25. Floating-point Comparison Types

fctype	PR[*qp*]==0		PR[*qp*]==1					
			result==0, No Source NaTVals		result==1, No Source NaTVals		One or More Source NaTVals	
	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]
none			0	1	1	0	0	0
unc	0	0	0	1	1	0	0	0

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

The relations are defined for each of the comparison types in Table 2-26. Of the twelve relations, not all are directly implemented in hardware. Some are actually pseudo-ops. For these, the assembler simply switches the source operand specifiers and/or switches the predicate target specifiers and uses an implemented relation.

Table 2-26. Floating-point Comparison Relations

frel	*frel* Completer Unabbreviated	Relation	Pseudo-op of		Quiet NaN as Operand Signals Invalid
eq	equal	$f_2 == f_3$			No
lt	less than	$f_2 < f_3$			Yes
le	less than or equal	$f_2 <= f_3$			Yes
gt	greater than	$f_2 > f_3$	lt	$f_2 \leftrightarrow f_3$	Yes
ge	greater than or equal	$f_2 >= f_3$	le	$f_2 \leftrightarrow f_3$	Yes
unord	unordered	f_2 ? f_3			No
neq	not equal	!($f_2 == f_3$)	eq	$p_1 \leftrightarrow p_2$	No
nlt	not less than	!($f_2 < f_3$)	lt	$p_1 \leftrightarrow p_2$	Yes
nle	not less than or equal	!($f_2 <= f_3$)	le	$p_1 \leftrightarrow p_2$	Yes
ngt	not greater than	!($f_2 > f_3$)	lt $f_2 \leftrightarrow f_3$	$p_1 \leftrightarrow p_2$	Yes
nge	not greater than or equal	!($f_2 >= f_3$)	le $f_2 \leftrightarrow f_3$	$p_1 \leftrightarrow p_2$	Yes
ord	ordered	!(f_2 ? f_3)	unord	$p_1 \leftrightarrow p_2$	No

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (tmp_isrcode = fp_reg_disabled(f2, f3, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        PR[p1] = 0;
        PR[p2] = 0;
    } else {
        fcmp_exception_fault_check(f2, f3, frel, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = fp_reg_read(FR[f2]);
        tmp_fr3 = fp_reg_read(FR[f3]);

        if      (frel == 'eq')  tmp_rel = fp_equal(tmp_fr2,
                                                   tmp_fr3);
        else if (frel == 'lt')  tmp_rel = fp_less_than(tmp_fr2,
                                                   tmp_fr3);
        else if (frel == 'le')  tmp_rel = fp_lesser_or_equal(tmp_fr2,
                                                   tmp_fr3);
        else if (frel == 'gt')  tmp_rel = fp_less_than(tmp_fr3,
                                                   tmp_fr2);
        else if (frel == 'ge')  tmp_rel = fp_lesser_or_equal(tmp_fr3,
                                                   tmp_fr2);
        else if (frel == 'unord')tmp_rel = fp_unordered(tmp_fr2,
                                                   tmp_fr3);
        else if (frel == 'neq') tmp_rel = !fp_equal(tmp_fr2,
                                                   tmp_fr3);
        else if (frel == 'nlt') tmp_rel = !fp_less_than(tmp_fr2,
                                                   tmp_fr3);
        else if (frel == 'nle') tmp_rel = !fp_lesser_or_equal(tmp_fr2,
                                                   tmp_fr3);
        else if (frel == 'ngt') tmp_rel = !fp_less_than(tmp_fr3,
                                                   tmp_fr2);
        else if (frel == 'nge') tmp_rel = !fp_lesser_or_equal(tmp_fr3,
                                                   tmp_fr2);
        else                    tmp_rel = !fp_unordered(tmp_fr2,
                                                   tmp_fr3); //'ord'

        PR[p1] = tmp_rel;
        PR[p2] = !tmp_rel;

        fp_update_fpsr(sf, tmp_fp_env);
    }
} else {
    if (fctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Floating-point Exception fault
Disabled Floating-point Register fault

Convert Floating-Point to Integer

Format:

(*qp*) fcvt.fx.*sf* $f_1 = f_2$	signed_form	F10
(*qp*) fcvt.fx.trunc.*sf* $f_1 = f_2$	signed_form, trunc_form	F10
(*qp*) fcvt.fxu.*sf* $f_1 = f_2$	unsigned_form	F10
(*qp*) fcvt.fxu.trunc.*sf* $f_1 = f_2$	unsigned_form, trunc_form	F10

Description:

FR f_2 is treated as a register format floating-point value and converted to a signed (signed_form) or unsigned integer (unsigned_form) using either the rounding mode specified in the FPSR.*sf.rc*, or using Round-to-Zero if the trunc_form of the instruction is used. The result is placed in the 64-bit significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0). If the result of the conversion cannot be represented as a 64-bit integer, the 64-bit integer indefinite value 0x8000000000000000 is used as the result, if the IEEE Invalid Operation Floating-point Exception fault is disabled.

If FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fcvt_exception_fault_check(f2, signed_form,
                                                trunc_form, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result)) {
            FR[f1].significand = INTEGER_INDEFINITE;
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = FP_SIGN_POSITIVE;
        } else {
            tmp_res = fp_ieee_rnd_to_int(fp_reg_read(FR[f2]), &tmp_fp_env);
            if (tmp_res.exponent)
                tmp_res.significand = fp_U64_rsh(
                    tmp_res.significand, (FP_INTEGER_EXP - tmp_res.exponent));
            if (signed_form && tmp_res.sign)
                tmp_res.significand = (~tmp_res.significand) + 1;

            FR[f1].significand = tmp_res.significand;
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = FP_SIGN_POSITIVE;
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:	Invalid Operation (V)	Inexact (I)
	Denormal/Unnormal Operand (D)	
	Software Assist (SWA) fault	
Interruptions:	Illegal Operation fault	Floating-point Exception fault
	Disabled Floating-point Register fault	Floating-point Exception trap

Convert Signed Integer to Floating-point

Format: (*qp*) fcvt.xf $f_1 = f_2$ F11

Description: The 64-bit significand of FR f_2 is treated as a signed integer and its register file precision floating-point representation is placed in FR f_1.

If FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation is always exact and is unaffected by the rounding mode.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2])) {
        FR[f1] = NATVAL;
    } else {
        tmp_res = FR[f2];
        if (tmp_res.significand{63}) {
            tmp_res.significand = (~tmp_res.significand) + 1;
            tmp_res.sign = 1;
        } else
            tmp_res.sign = 0;

        tmp_res.exponent = FP_INTEGER_EXP;
        tmp_res = fp_normalize(tmp_res);

        FR[f1].significand = tmp_res.significand;
        FR[f1].exponent = tmp_res.exponent;
        FR[f1].sign = tmp_res.sign;
    }
    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Convert Unsigned Integer to Floating-point

Format: (*qp*) fcvt.xuf.*pc*.*sf* $f_1 = f_3$ pseudo-op of: (*qp*) fma.*pc*.*sf* $f_1 = f_3$, f1, f0

Description: FR f_3 is multiplied with FR 1, rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*, and placed in FR f_1.

Note: Multiplying FR f_3 with FR 1 (a 1.0) normalizes the canonical representation of an integer in the floating-point register file producing a normal floating-point value. If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-21 on page 2-42. The mnemonic values for *sf* are given in Table 2-22 on page 2-42. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation: See "Floating-Point Multiply Add" on page 2-61

Fetch And Add Immediate

Format:

(*qp*) fetchadd4.*sem.ldhint* $r_1 = [r_3]$, inc_3	four_byte_form	M17
(*qp*) fetchadd8.*sem.ldhint* $r_1 = [r_3]$, inc_3	eight_byte_form	M17

Description: A value consisting of four or eight bytes is read from memory starting at the address specified by the value in GR r_3. The value is zero extended and added to the sign-extended immediate value specified by inc_3. The values that may be specified by inc_3 are: -16, -8, -4, -1, 1, 4, 8, 16. The least significant four or eight bytes of the sum are then written to memory starting at the address specified by the value in GR r_3. The zero-extended value read from memory is placed in GR r_1 and the NaT bit corresponding to GR r_1 is cleared.

The *sem* completer specifies the type of semaphore operation. These operations are described in Table 2-27. See Volume 1 and Volume 2 for details on memory ordering.

Table 2-27. Fetch and Add Semaphore Types

sem Completer	Ordering Semantics	Semaphore Operation
acq	Acquire	The memory read/write is made visible prior to all subsequent data memory accesses.
rel	Release	The memory read/write is made visible after all previous data memory accesses.

The memory read and write are guaranteed to be atomic for accesses to pages with cacheable, writeback memory attribute. For accesses to other memory types, atomicity is platform-dependent.

If the address specified by the value in GR r_3 is not naturally aligned to the size of the value being accessed in memory, an Unaligned Data Reference fault is taken independent of the state of the User Mask alignment checking bit, UM.ac (PSR.ac in the Processor Status Register).

Both read and write access privileges for the referenced page are required. The write access privilege check is performed whether or not the memory write is performed.

Only accesses to UCE pages or cacheable pages with write-back write policy are permitted. Accesses to NaTPages result in a Data NaT Page Consumption fault. Accesses to pages with other memory attributes cause an Unsupported Data Reference fault.

On a processor model that supports exported `fetchadd`, a `fetchadd` to a UCE page causes the fetch-and-add operation to be exported outside of the processor; if the platform does not support exported `fetchadd`, the operation is undefined. On a processor model that does not support exported `fetchadd`, a `fetchadd` to a UCE page causes an Unsupported Data Reference fault. See “Effects of Memory Attributes on Memory Reference Instructions” on p. 4-37 in Volume 2.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 2-32 on page 2-127. Locality hints do not affect program functionality and may be ignored by the implementation.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (GR[r3].nat)
        register_nat_consumption_fault(SEMAPHORE);

    size = four_byte_form ? 4 : 8;

    paddr = tlb_translate(GR[r3], size, SEMAPHORE, PSR.cpl, &mattr,
                          &tmp_unused);
    if (!ma_supports_fetchadd(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

    if (sem == 'acq')
        val = mem_xchg_add(inc3, paddr, size, UM.be, mattr, ACQUIRE, ldhint);
    else // 'rel'
        val = mem_xchg_add(inc3, paddr, size, UM.be, mattr, RELEASE, ldhint);

    alat_inval_multiple_entries(paddr, size);

    GR[r1] = zero_ext(val, size * 8);
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

Flush Register Stack

Format: flushrs M25

Description: All stacked general registers in the dirty partition of the register stack are written to the backing store before execution continues. The dirty partition contains registers from previous procedure frames that have not yet been saved to the backing store. For a description of the register stack partitions, refer to Volume 2. A pending external interrupt can interrupt the RSE store loop when enabled.

After this instruction completes execution BSPSTORE is equal to BSP.

This instruction must be the first instruction in an instruction group. Otherwise, the results are undefined. This instruction cannot be predicated.

Operation:

```
while (AR[BSPSTORE] != AR[BSP]) {
    rse_store(MANDATORY);              // increments AR[BSPSTORE]
    deliver_unmasked_pending_external_interrupt();
}
```

Interruptions:

Unimplemented Data Address fault
VHPT Data fault
Data Nested TLB fault
Data TLB fault
Alternate Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault

Floating-Point Multiply Add

Format: (*qp*) fma.*pc*.*sf* $f_1 = f_3, f_4, f_2$ F1

Description: The product of FR f_3 and FR f_4 is computed to infinite precision and then FR f_2 is added to this product, again in infinite precision. The resulting value is then rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*. The rounded result is placed in FR f_1.

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

If f_2 is f0, an IEEE multiply operation is performed instead of a multiply and add. See "Floating-Point Multiply" on p. 2-68.

The mnemonic values for the opcode's *pc* are given in Table 2-21 on page 2-42. The mnemonic values for *sf* are given in Table 2-22 on page 2-42. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fma_exception_fault_check(f2, f3, f4,
                                                       pc, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[f3]), fp_reg_read(FR[f4]));
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read(FR[f2]), tmp_fp_env);
            FR[f1] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:
- Invalid Operation (V)
- Denormal/Unnormal Operand (D)
- Software Assist (SWA) fault
- Underflow (U)
- Overflow (O)
- Inexact (I)
- Software Assist (SWA) trap

Interruptions:
- Illegal Operation fault
- Disabled Floating-point Register fault
- Floating-point Exception fault
- Floating-point Exception trap

Floating-Point Maximum

Format: (*qp*) fmax.*sf* $f_1 = f_2, f_3$ F8

Description: The operand with the larger value is placed in FR f_1. If FR f_2 equals FR f_3, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaN, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as the `fcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_bool_res = fp_less_than(fp_reg_read(FR[f3]),
                                    fp_reg_read(FR[f2]));
        FR[f1] = (tmp_bool_res ? FR[f2] : FR[f3]);

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Floating-point Exception fault
Disabled Floating-point Register fault

Floating-Point Merge

Format:

(*qp*) fmerge.ns $f_1 = f_2, f_3$	neg_sign_form	F9
(*qp*) fmerge.s $f_1 = f_2, f_3$	sign_form	F9
(*qp*) fmerge.se $f_1 = f_2, f_3$	sign_exp_form	F9

Description: Sign, exponent and significand fields are extracted from FR f_2 and FR f_3, combined, and the result is placed in FR f_1.

For the neg_sign_form, the sign of FR f_2 is negated and concatenated with the exponent and the significand of FR f_3. This form can be used to negate a floating-point number by using the same register for FR f_2 and FR f_3.

For the sign_form, the sign of FR f_2 is concatenated with the exponent and the significand of FR f_3.

For the sign_exp_form, the sign and exponent of FR f_2 is concatenated with the significand of FR f_3.

For all forms, if either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-7. Floating-point Merge Negative Sign Operation

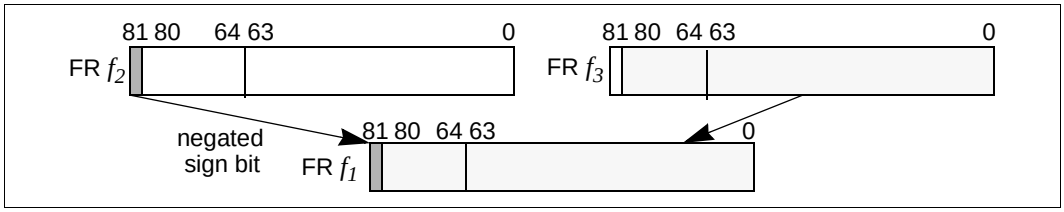


Figure 2-8. Floating-point Merge Sign Operation

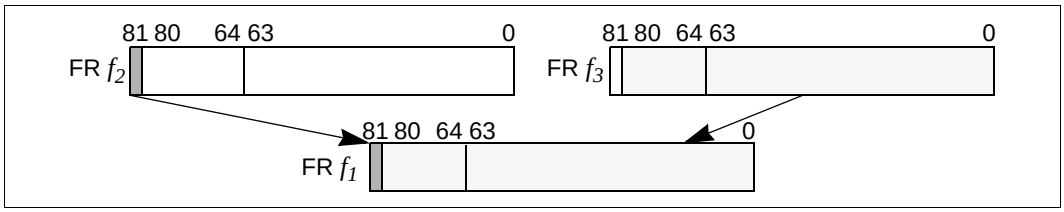
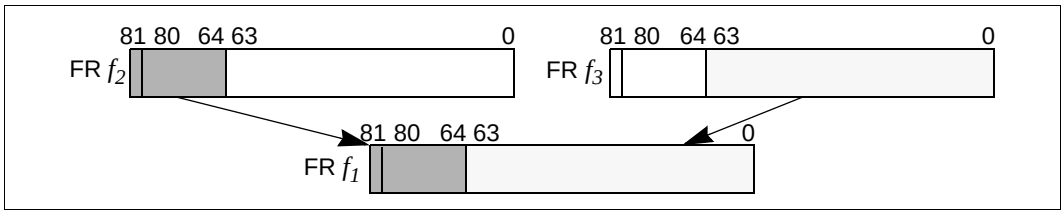


Figure 2-9. Floating-point Merge Sign and Exponent Operation



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f3].significand;
        if (neg_sign_form) {
            FR[f1].exponent = FR[f3].exponent;
            FR[f1].sign = !FR[f2].sign;
        } else if (sign_form) {
            FR[f1].exponent = FR[f3].exponent;
            FR[f1].sign = FR[f2].sign;
        } else {                                        // sign_exp_form
            FR[f1].exponent = FR[f2].exponent;
            FR[f1].sign = FR[f2].sign;
        }
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Floating-Point Minimum

Format: (*qp*) fmin.*sf* $f_1 = f_2, f_3$ F8

Description: The operand with the smaller value is placed in FR f_1. If FR f_2 equals FR f_3, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaN, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as the `fcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_bool_res = fp_less_than(fp_reg_read(FR[f2]),
                                    fp_reg_read(FR[f3]));
        FR[f1] = tmp_bool_res ? FR[f2] : FR[f3];

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

Floating-Point Parallel Mix

Format:

(*qp*) fmix.l $f_1 = f_2, f_3$	mix_l_form	F9
(*qp*) fmix.r $f_1 = f_2, f_3$	mix_r_form	F9
(*qp*) fmix.lr $f_1 = f_2, f_3$	mix_lr_form	F9

Description: For the mix_l_form (mix_r_form), the left (right) single precision value in FR f_2 is concatenated with the left (right) single precision value in FR f_3. For the mix_lr_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3.

For all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For all forms, if either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-10. Floating-point Mix Left

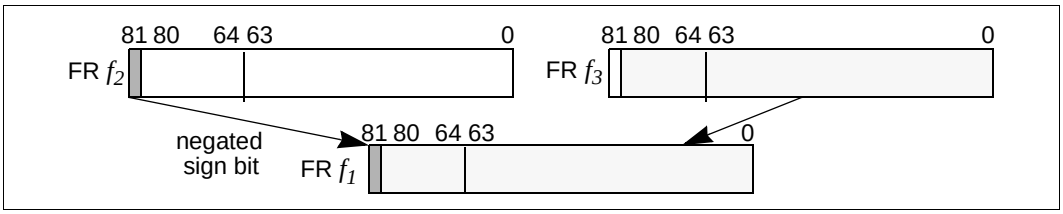


Figure 2-11. Floating-point Mix Right

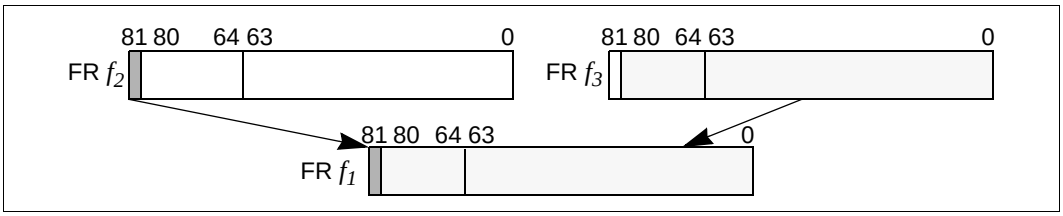
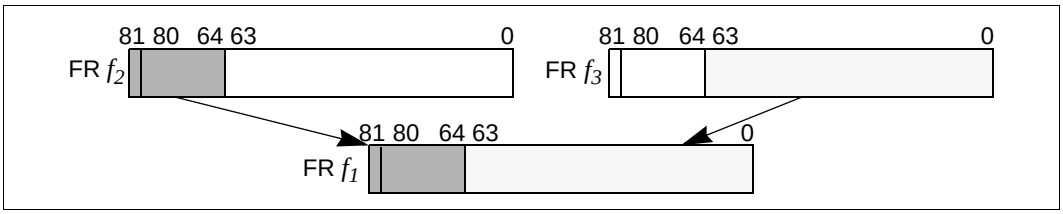


Figure 2-12. Floating-point Mix Left-Right



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (mix_l_form) {
            tmp_res_hi = FR[f2].significand{63:32};
            tmp_res_lo = FR[f3].significand{63:32};
        } else if (mix_r_form) {
            tmp_res_hi = FR[f2].significand{31:0};
            tmp_res_lo = FR[f3].significand{31:0};
        } else {                                          // mix_lr_form
            tmp_res_hi = FR[f2].significand{63:32};
            tmp_res_lo = FR[f3].significand{31:0};
        }
        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Floating-Point Multiply

Format: (*qp*) fmpy.*pc*.*sf* $f_1 = f_3, f_4$ pseudo-op of: (*qp*) fma.*pc*.*sf* $f_1 = f_3, f_4$, f0

Description: The product FR f_3 and FR f_4 is computed to infinite precision. The resulting value is then rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*. The rounded result is placed in FR f_1.

If either FR f_3 or FR f_4 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-21 on page 2-42. The mnemonic values for *sf* are given in Table 2-22 on page 2-42. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation: See "Floating-Point Multiply Add" on p. 2-61.

Floating-Point Multiply Subtract

Format: (*qp*) fms.*pc*.*sf* $f_1 = f_3, f_4, f_2$ F1

Description: The product of FR f_3 and FR f_4 is computed to infinite precision and then FR f_2 is subtracted from this product, again in infinite precision. The resulting value is then rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*. The rounded result is placed in FR f_1.

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, a NaTVal is placed in FR f_1 instead of the computed result.

If f_2 is f0, an IEEE multiply operation is performed instead of a multiply and subtract. See "Floating-Point Multiply" on p. 2-68.

The mnemonic values for the opcode's *pc* are given in Table 2-21 on page 2-42. The mnemonic values for *sf* are given in Table 2-22 on page 2-42. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fms_fnma_exception_fault_check(f2, f3, f4,
                                                            pc, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[f3]), fp_reg_read(FR[f4]));
            tmp_fr2 = fp_reg_read(FR[f2]);
            tmp_fr2.sign = !tmp_fr2.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, tmp_fr2, tmp_fp_env);
            FR[f1] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:
- Invalid Operation (V)
- Denormal/Unnormal Operand (D)
- Software Assist (SWA) fault
- Underflow (U)
- Overflow (O)
- Inexact (I)
- Software Assist (SWA) trap

Interruptions:
- Illegal Operation fault
- Disabled Floating-point Register fault
- Floating-point Exception fault
- Floating-point Exception trap

Floating-Point Negate

Format: (*qp*) fneg $f_1 = f_3$ pseudo-op of: (*qp*) fmerge.ns $f_1 = f_3, f_3$

Description: The value in FR f_3 is negated and placed in FR f_1.

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See "Floating-Point Merge" on p. 2-63.

Floating-Point Negate Absolute Value

Format: (*qp*) fnegabs $f_1 = f_3$ pseudo-op of: (*qp*) fmerge.ns $f_1 = f0, f_3$

Description: The absolute value of the value in FR f_3 is computed, negated, and placed in FR f_1.

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See "Floating-Point Merge" on p. 2-63.

Floating-Point Negative Multiply Add

Format: (*qp*) fnma.*pc*.*sf* $f_1 = f_3, f_4, f_2$ F1

Description: The product of FR f_3 and FR f_4 is computed to infinite precision, negated, and then FR f_2 is added to this product, again in infinite precision. The resulting value is then rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*. The rounded result is placed in FR f_1.

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

If f_2 is f0, an IEEE multiply operation is performed, followed by negation of the product. See "Floating-Point Negative Multiply" on p. 2-73.

The mnemonic values for the opcode's *pc* are given in Table 2-21 on page 2-42. The mnemonic values for *sf* are given in Table 2-22 on page 2-42. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fms_fnma_exception_fault_check(f2, f3, f4,
                                                            pc, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[f3]), fp_reg_read(FR[f4]));
            tmp_res.sign = !tmp_res.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read(FR[f2]), tmp_fp_env);
            FR[f1] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:
- Invalid Operation (V)
- Denormal/Unnormal Operand (D)
- Software Assist (SWA) fault
- Underflow (U)
- Overflow (O)
- Inexact (I)
- Software Assist (SWA) trap

Interruptions:
- Disabled Floating-point Register fault
- Floating-point Exception fault
- Floating-point Exception trap

Floating-Point Negative Multiply

Format: (*qp*) fnmpy.*pc*.*sf* $f_1 = f_3, f_4$ pseudo-op of: (*qp*) fnma.*pc*.*sf* $f_1 = f_3, f_4$,f0

Description: The product FR f_3 and FR f_4 is computed to infinite precision and then negated. The resulting value is then rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*. The rounded result is placed in FR f_1.

If either FR f_3 or FR f_4 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-21 on page 2-42. The mnemonic values for *sf* are given in Table 2-22 on page 2-42. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation: See "Floating-Point Negative Multiply Add" on p. 2-72.

Floating-Point Normalize

Format: (*qp*) fnorm.*pc*.*sf* $f_1 = f_3$ pseudo-op of: (*qp*) fma.*pc*.*sf* $f_1 = f_3$, f1, f0

Description: FR f_3 is normalized and rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*, and placed in FR f_1.

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-21 on page 2-42. The mnemonic values for *sf* are given in Table 2-22 on page 2-42. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation: See "Floating-Point Multiply Add" on p. 2-61.

Floating-Point Logical Or

Format: (*qp*) for $f_1 = f_2, f_3$ F9

Description: The bit-wise logical OR of the significand fields of FR f_2 and FR f_3 is computed. The resulting value is stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand | FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Floating-Point Parallel Absolute Value

Format: (*qp*) fpabs $f_1 = f_3$ pseudo-op of: (*qp*) fpmerge.s f_1 = f0, f_3

Description: The absolute values of the pair of single precision values in the significand field of FR f_3 are computed and stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See "Floating-Point Parallel Merge" on p. 2-90.

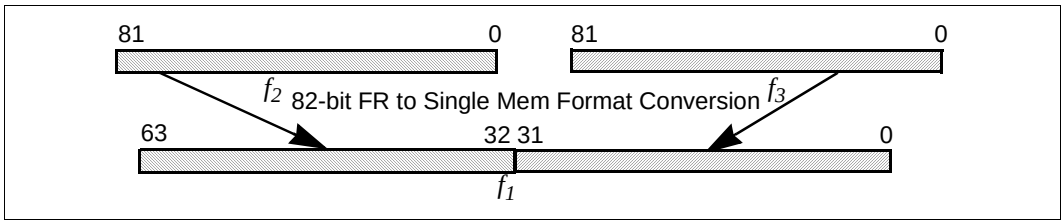
Floating-Point Pack

Format: (*qp*) fpack $f_1 = f_2, f_3$ pack_form F9

Description: The register format numbers in FR f_2 and FR f_3 are converted to single precision memory format. These two single precision numbers are concatenated and stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-13. Floating-point Pack



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        tmp_res_hi = fp_single(FR[f2]);
        tmp_res_lo = fp_single(FR[f3]);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }
    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Floating-Point Parallel Absolute Maximum

Format: (*qp*) fpamax.*sf* $f_1 = f_2, f_3$ F8

Description: The paired single precision values in the significands of FR f_2 and FR f_3 are compared. The operands with the larger absolute value are returned in the significand field of FR f_1.

If the magnitude of high (low) FR f_3 is less than the magnitude of high (low) FR f_2, high (low) FR f_1 gets high (low) FR f_2. Otherwise high (low) FR f_1 gets high (low) FR f_3.

If high (low) FR f_2 or high (low) FR f_3 is a NaN, and neither FR f_2 or FR f_3 is a NaTVal, high (low) FR f_1 gets high (low) FR f_3.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as for the `fpcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_right = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_left = fp_reg_read_hi(f3);
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        tmp_fr2 = tmp_right = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_left = fp_reg_read_lo(f3);
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

Floating-Point Parallel Absolute Minimum

Format: (*qp*) fpamin.*sf* $f_1 = f_2, f_3$ F8

Description: The paired single precision values in the significands of FR f_2 or FR f_3 are compared. The operands with the smaller absolute value is returned in the significand of FR f_1.

If the magnitude of high (low) FR f_2 is less than the magnitude of high (low) FR f_3, high (low) FR f_1 gets high (low) FR f_2. Otherwise high (low) FR f_1 gets high (low) FR f_3.

If high (low) FR f_2 or high (low) FR f_3 is a NaN, and neither FR f_2 or FR f_3 is a NaTVal, high (low) FR f_1 gets high (low) FR f_3.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as for the `fpcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_left = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_right = fp_reg_read_hi(f3);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        tmp_fr2 = tmp_left = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_right = fp_reg_read_lo(f3);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

Floating-Point Parallel Compare

Format: (*qp*) fpcmp.*frel*.*sf* $f_1 = f_2, f_3$ F8

Description: The two pairs of single precision source operands in the significand fields of FR f_2 and FR f_3 are compared for one of twelve relations specified by *frel*. This produces a boolean result which is a mask of 32 1's if the comparison condition is true, and a mask of 32 0's otherwise. This result is written to a pair of 32-bit integers in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

Table 2-28. Floating-point Parallel Comparison Results

PR[*qp*]==0	PR[*qp*]==1		
	result==false, No Source NaTVals	result==true, No Source NaTVals	One or More Source NaTVals
unchanged	0...0	1...1	NaTVal

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

The relations are defined for each of the comparison types in Table 2-28. Of the twelve relations, not all are directly implemented in hardware. Some are actually pseudo-ops. For these, the assembler simply switches the source operand specifiers and/or switches the predicate type specifiers and uses an implemented relation.

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Table 2-29. Floating-point Parallel Comparison Relations

frel	*frel* Completer unabbreviated	Relation	Pseudo-op of	Quiet NaN as Operand Signals Invalid
eq	equal	$f_2 == f_3$		No
lt	less than	$f_2 < f_3$		Yes
le	less than or equal	$f_2 <= f_3$		Yes
gt	greater than	$f_2 > f_3$	lt $f_2 \leftrightarrow f_3$	Yes
ge	greater than or equal	$f_2 >= f_3$	le $f_2 \leftrightarrow f_3$	Yes
unord	unordered	$f_2 ? f_3$		No
neq	not equal	$!(f_2 == f_3)$		No
nlt	not less than	$!(f_2 < f_3)$		Yes
nle	not less than or equal	$!(f_2 <= f_3)$		Yes
ngt	not greater than	$!(f_2 > f_3)$	nlt $f_2 \leftrightarrow f_3$	Yes
nge	not greater than or equal	$!(f_2 >= f_3)$	nle $f_2 \leftrightarrow f_3$	Yes
ord	ordered	$!(f_2 ? f_3)$		No

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpcmp_exception_fault_check(f2, f3, frel, sf, &tmp_fp_env);

        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = fp_reg_read_hi(f2);
        tmp_fr3 = fp_reg_read_hi(f3);

        if      (frel == 'eq')  tmp_rel = fp_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'lt')  tmp_rel = fp_less_than(tmp_fr2, tmp_fr3);
        else if (frel == 'le')  tmp_rel = fp_lesser_or_equal(tmp_fr2,
                                                  tmp_fr3);
        else if (frel == 'gt')  tmp_rel = fp_less_than(tmp_fr3, tmp_fr2);
        else if (frel == 'ge')  tmp_rel = fp_lesser_or_equal(tmp_fr3,
                                                  tmp_fr2);
        else if (frel == 'unord')tmp_rel = fp_unordered(tmp_fr2, tmp_fr3);
        else if (frel == 'neq') tmp_rel = !fp_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'nlt') tmp_rel = !fp_less_than(tmp_fr2, tmp_fr3);
        else if (frel == 'nle') tmp_rel = !fp_lesser_or_equal(tmp_fr2,
                                                  tmp_fr3);
        else if (frel == 'ngt') tmp_rel = !fp_less_than(tmp_fr3, tmp_fr2);
        else if (frel == 'nge') tmp_rel = !fp_lesser_or_equal(tmp_fr3,
                                                  tmp_fr2);
        else                    tmp_rel = !fp_unordered(tmp_fr2,
                                                  tmp_fr3); //'ord'

        tmp_res_hi = (tmp_rel ? 0xFFFFFFFF : 0x00000000);

        tmp_fr2 = fp_reg_read_lo(f2);
        tmp_fr3 = fp_reg_read_lo(f3);

        if      (frel == 'eq')  tmp_rel = fp_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'lt')  tmp_rel = fp_less_than(tmp_fr2, tmp_fr3);
        else if (frel == 'le')  tmp_rel = fp_lesser_or_equal(tmp_fr2,
                                                  tmp_fr3);
        else if (frel == 'gt')  tmp_rel = fp_less_than(tmp_fr3, tmp_fr2);
        else if (frel == 'ge')  tmp_rel = fp_lesser_or_equal(tmp_fr3,
                                                  tmp_fr2);
        else if (frel == 'unord')tmp_rel = fp_unordered(tmp_fr2, tmp_fr3);
        else if (frel == 'neq') tmp_rel = !fp_equal(tmp_fr2, tmp_fr3);
        else if (frel == 'nlt') tmp_rel = !fp_less_than(tmp_fr2, tmp_fr3);
        else if (frel == 'nle') tmp_rel = !fp_lesser_or_equal(tmp_fr2,
                                                  tmp_fr3);
        else if (frel == 'ngt') tmp_rel = !fp_less_than(tmp_fr3, tmp_fr2);
        else if (frel == 'nge') tmp_rel = !fp_lesser_or_equal(tmp_fr3,
                                                  tmp_fr2);
        else                    tmp_rel = !fp_unordered(tmp_fr2,
                                                  tmp_fr3); //'ord'

        tmp_res_lo = (tmp_rel ? 0xFFFFFFFF : 0x00000000);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
```

```
            fp_update_fpsr(sf, tmp_fp_env);
        }
        fp_update_psr(f1);
    }
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Floating-point Exception fault
Disabled Floating-point Register fault

Convert Parallel Floating-Point to Integer

Format:

(*qp*) fpcvt.fx.*sf* $f_1 = f_2$	signed_form	F10
(*qp*) fpcvt.fx.trunc.*sf* $f_1 = f_2$	signed_form, trunc_form	F10
(*qp*) fpcvt.fxu.*sf* $f_1 = f_2$	unsigned_form	F10
(*qp*) fpcvt.fxu.trunc.*sf* $f_1 = f_2$	unsigned_form, trunc_form	F10

Description: The pair of single precision values in the significand field of FR f_2 is converted to a pair of 32-bit signed integers (signed_form) or unsigned integers (unsigned_form) using either the rounding mode specified in the FPSR.*sf.rc*, or using Round-to-Zero if the trunc_form of the instruction is used. The result is written as a pair of 32-bit integers into the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0). If the result of the conversion cannot be represented as a 32-bit integer, the 32-bit integer indefinite value 0x80000000 is used as the result, if the IEEE Invalid Operation Floating-point Exception fault is disabled.

If FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result_pair = fpcvt_exception_fault_check(f2,
                                  signed_form, trunc_form, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result_pair.hi)) {
            tmp_res_hi = INTEGER_INDEFINITE_32_BIT;
        } else {
            tmp_res = fp_ieee_rnd_to_int_sp(fp_reg_read_hi(f2), HIGH,
                                            &tmp_fp_env);
            if (tmp_res.exponent)
                tmp_res.significand = fp_U64_rsh(
                    tmp_res.significand, (FP_INTEGER_EXP - tmp_res.exponent));
            if (signed_form && tmp_res.sign)
                tmp_res.significand = (~tmp_res.significand) + 1;

            tmp_res_hi = tmp_res.significand{31:0};
        }

        if (fp_is_nan(tmp_default_result_pair.lo)) {
            tmp_res_lo = INTEGER_INDEFINITE_32_BIT;
        } else {
            tmp_res = fp_ieee_rnd_to_int_sp(fp_reg_read_lo(f2), LOW,
                                            &tmp_fp_env);
            if (tmp_res.exponent)
                tmp_res.significand = fp_U64_rsh(
                    tmp_res.significand, (FP_INTEGER_EXP - tmp_res.exponent));
            if (signed_form && tmp_res.sign)
                tmp_res.significand = (~tmp_res.significand) + 1;
```

```
            tmp_res_lo = tmp_res.significand{31:0};
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) Fault
Inexact (I)

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault
Floating-point Exception trap

Floating-Point Parallel Multiply Add

Format: (*qp*) fpma.*sf* $f_1 = f_3, f_4, f_2$ F1

Description: The pair of products of the pairs of single precision values in the significand fields of FR f_3 and FR f_4 are computed to infinite precision and then the pair of single precision values in the significand field of FR f_2 is added to these products, again in infinite precision. The resulting values are then rounded to single precision using the rounding mode specified by FPSR.*sf.rc*. The pair of rounded results are stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed results.

Note: If f_2 is f0 in the `fpma` instruction, just the IEEE multiply operation is performed. (See "Floating-Point Parallel Multiply" on p. 2-93.) FR f1, as an operand, is not a packed pair of 1.0 values, it is just the register file format's 1.0 value.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 5-7.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result_pair = fpma_exception_fault_check(f2,
                                                f3, f4, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
        } else {
            tmp_res = fp_mul(fp_reg_read_hi(f3), fp_reg_read_hi(f4));
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_hi(f2), tmp_fp_env);
            tmp_res_hi = fp_ieee_round_sp(tmp_res, HIGH, &tmp_fp_env);
        }

        if (fp_is_nan_or_inf(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
        } else {
            tmp_res = fp_mul(fp_reg_read_lo(f3), fp_reg_read_lo(f4));
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_lo(f2), tmp_fp_env);
            tmp_res_lo = fp_ieee_round_sp(tmp_res, LOW, &tmp_fp_env);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
```

```
            if (fp_raise_traps(tmp_fp_env))
                fp_exception_trap(fp_decode_trap(tmp_fp_env));
        }
    }
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) Fault
Underflow (U)
Overflow (O)
Inexact (I)
Software Assist (SWA) trap

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault
Floating-point Exception trap

Floating-Point Parallel Maximum

Format: (*qp*) fpmax.*sf* $f_1 = f_2, f_3$ F8

Description: The paired single precision values in the significands of FR f_2 or FR f_3 are compared. The operands with the larger value is returned in the significand of FR f_1.

If the value of high (low) FR f_3 is less than the value of high (low) FR f_2, high (low) FR f_1 gets high (low) FR f_2. Otherwise high (low) FR f_1 gets high (low) FR f_3.

If high (low) FR f_2 or high (low) FR f_3 is a NaN, high (low) FR f_1 gets high (low) FR f_3.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as for the `fpcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_right = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_left = fp_reg_read_hi(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2 : tmp_fr3);

        tmp_fr2 = tmp_right = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_left = fp_reg_read_lo(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2 : tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

Floating-Point Parallel Merge

Format:

(*qp*) fpmerge.ns $f_1 = f_2, f_3$	neg_sign_form	F9
(*qp*) fpmerge.s $f_1 = f_2, f_3$	sign_form	F9
(*qp*) fpmerge.se $f_1 = f_2, f_3$	sign_exp_form	F9

Description: For the neg_sign_form, the signs of the pair of single precision values in the significand field of FR f_2 are negated and concatenated with the exponents and the significands of the pair of single precision values in the significand field of FR f_3 and stored in the significand field of FR f_1. This form can be used to negate a pair of single precision floating-point numbers by using the same register for f_2 and f_3.

For the sign_form, the signs of the pair of single precision values in the significand field of FR f_2 are concatenated with the exponents and the significands of the pair of single precision values in the significand field of FR f_3 and stored in FR f_1.

For the sign_exp_form, the signs and exponents of the pair of single precision values in the significand field of FR f_2 are concatenated with the pair of single precision significands in the significand field of FR f_3 and stored in the significand field of FR f_1.

For all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For all forms, if either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-14. Floating-point Merge Negative Sign Operation

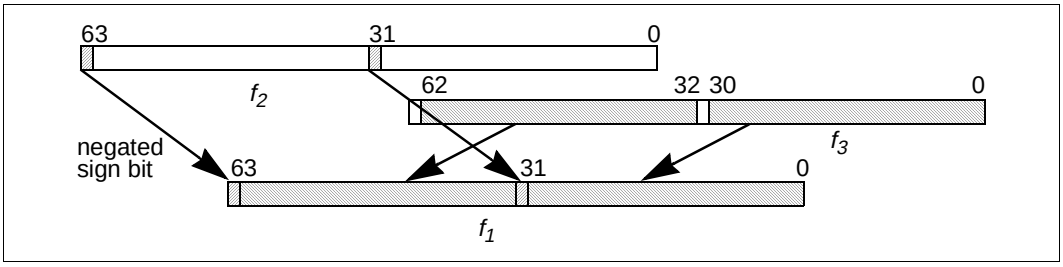


Figure 2-15. Floating-point Merge Sign Operation

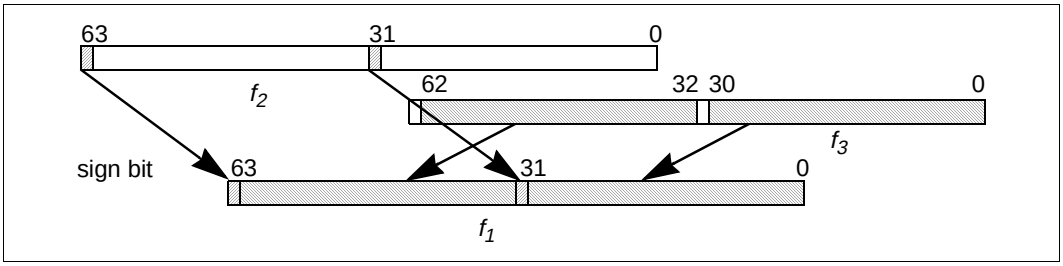
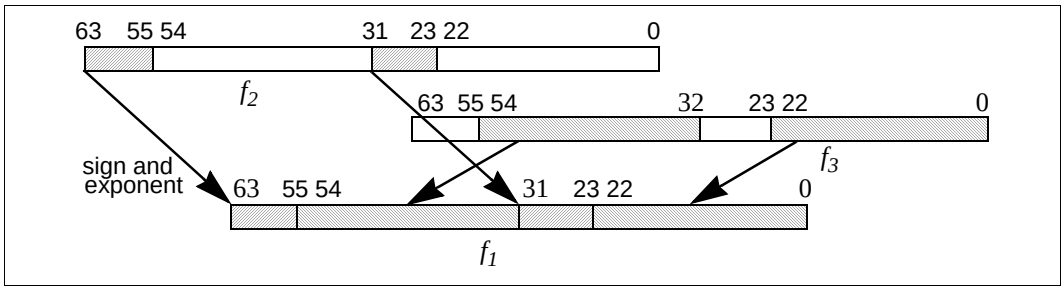


Figure 2-16. Floating-point Merge Sign and Exponent Operation



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (neg_sign_form) {
            tmp_res_hi = (!FR[f2].significand{63} << 31)
                       | (FR[f3].significand{62:32});
            tmp_res_lo = (!FR[f2].significand{31} << 31)
                       | (FR[f3].significand{30:0});
        } else if (sign_form) {
            tmp_res_hi = (FR[f2].significand{63} << 31)
                       | (FR[f3].significand{62:32});
            tmp_res_lo = (FR[f2].significand{31} << 31)
                       | (FR[f3].significand{30:0});
        } else {                                        // sign_exp_form
            tmp_res_hi = (FR[f2].significand{63:55} << 23)
                       | (FR[f3].significand{54:32});
            tmp_res_lo = (FR[f2].significand{31:23} << 23)
                       | (FR[f3].significand{22:0});
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Floating-Point Parallel Minimum

Format: (*qp*) fpmin.*sf* $f_1 = f_2, f_3$ F8

Description: The paired single precision values in the significands of FR f_2 or FR f_3 are compared. The operands with the smaller value is returned in significand of FR f_1.

If the value of high (low) FR f_2 is less than the value of high (low) FR f_3, high (low) FR f_1 gets high (low) FR f_2. Otherwise high (low) FR f_1 gets high (low) FR f_3.

If high (low) FR f_2 or high (low) FR f_3 is a NaN, high (low) FR f_1 gets high (low) FR f_3.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as for the `fpcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_left = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_right = fp_reg_read_hi(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        tmp_fr2 = tmp_left = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_right = fp_reg_read_lo(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

Floating-Point Parallel Multiply

Format: (*qp*) fpmpy.*sf* $f_1 = f_3, f_4$ pseudo-op of: (*qp*) fpma.*sf* $f_1 = f_3, f_4$, f0

Description: The pair of products of the pairs of single precision values in the significand fields of FR f_3 and FR f_4 are computed to infinite precision. The resulting values are then rounded to single precision using the rounding mode specified by FPSR.*sf*.*rc*. The pair of rounded results are stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_3, or FR f_4 is a NaTVal, FR f_1 is set to NaTVal instead of the computed results.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 5-7.

Operation: See "Floating-Point Parallel Multiply Add" on p. 2-87.

Floating-Point Parallel Multiply Subtract

Format: (*qp*) fpms.*sf* $f_1 = f_3, f_4, f_2$ F1

Description: The pair of products of the pairs of single precision values in the significand fields of FR f_3 and FR f_4 are computed to infinite precision and then the pair of single precision values in the significand field of FR f_2 is subtracted from these products, again in infinite precision. The resulting values are then rounded to single precision using the rounding mode specified by FPSR.*sf.rc*. The pair of rounded results are stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

Note: If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed results.

Mapping: If f_2 is f0 in the fpms instruction, just the IEEE multiply operation is performed.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 5-7.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result_pair = fpms_fpnma_exception_fault_check(f2, f3,
                                                        f4, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
        } else {
            tmp_res = fp_mul(fp_reg_read_hi(f3), fp_reg_read_hi(f4));
            if (f2 != 0) {
                tmp_sub = fp_reg_read_hi(f2);
                tmp_sub.sign = !tmp_sub.sign;
                tmp_res = fp_add(tmp_res, tmp_sub, tmp_fp_env);
            }
            tmp_res_hi = fp_ieee_round_sp(tmp_res, HIGH, &tmp_fp_env);
        }

        if (fp_is_nan_or_inf(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
        } else {
            tmp_res = fp_mul(fp_reg_read_lo(f3), fp_reg_read_lo(f4));
            if (f2 != 0) {
                tmp_sub = fp_reg_read_lo(f2);
                tmp_sub.sign = !tmp_sub.sign;
                tmp_res = fp_add(tmp_res, tmp_sub, tmp_fp_env);
            }
            tmp_res_lo = fp_ieee_round_sp(tmp_res, LOW, &tmp_fp_env);
        }
```

```
            FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = FP_SIGN_POSITIVE;

            fp_update_fpsr(sf, tmp_fp_env);
            fp_update_psr(f1);
            if (fp_raise_traps(tmp_fp_env))
                fp_exception_trap(fp_decode_trap(tmp_fp_env));
        }
    }
```

FP Exceptions:

Invalid Operation (V)	Underflow (U)
Denormal/Unnormal Operand (D)	Overflow (O)
Software Assist (SWA) fault	Inexact (I)
	Software Assist (SWA) trap

Interruptions:

Illegal Operation fault	Floating-point Exception fault
Disabled Floating-point Register fault	Floating-point Exception trap

Floating-Point Parallel Negate

Format: (*qp*) fpneg $f_1 = f_3$ pseudo-op of: (*qp*) fpmerge.ns $f_1 = f_3, f_3$

Description: The pair of single precision values in the significand field of FR f_3 are negated and stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See "Floating-Point Parallel Merge" on p. 2-90.

Floating-Point Parallel Negate Absolute Value

Format: (*qp*) fpnegabs $f_1 = f_3$ pseudo-op of: (*qp*) fpmerge.ns $f_1 = f0, f_3$

Description: The absolute values of the pair of single precision values in the significand field of FR f_3 are computed, negated and stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See "Floating-Point Parallel Merge" on p. 2-90.

Floating-Point Parallel Negative Multiply Add

Format: (*qp*) fpnma.*sf* $f_1 = f_3, f_4, f_2$ F1

Description: The pair of products of the pairs of single precision values in the significand fields of FR f_3 and FR f_4 are computed to infinite precision, negated, and then the pair of single precision values in the significand field of FR f_2 are added to these (negated) products, again in infinite precision. The resulting values are then rounded to single precision using the rounding mode specified by FPSR.*sf.rc*. The pair of rounded results are stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Note: If f_2 is f0 in the `fpnma` instruction, just the IEEE multiply operation (with the product being negated before rounding) is performed.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 5-7.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result_pair = fpms_fpnma_exception_fault_check(f2, f3,
                                                    f4, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
        } else {
            tmp_res = fp_mul(fp_reg_read_hi(f3), fp_reg_read_hi(f4));
            tmp_res.sign = !tmp_res.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_hi(f2), tmp_fp_env);
            tmp_res_hi = fp_ieee_round_sp(tmp_res, HIGH, &tmp_fp_env);
        }

        if (fp_is_nan_or_inf(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
        } else {
            tmp_res = fp_mul(fp_reg_read_lo(f3), fp_reg_read_lo(f4));
            tmp_res.sign = !tmp_res.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_lo(f2), tmp_fp_env);
            tmp_res_lo = fp_ieee_round_sp(tmp_res, LOW, &tmp_fp_env);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
```

```
            fp_update_fpsr(sf, tmp_fp_env);
            fp_update_psr(f1);
            if (fp_raise_traps(tmp_fp_env))
                fp_exception_trap(fp_decode_trap(tmp_fp_env));
        }
    }
```

FP Exceptions:

Invalid Operation (V)	Underflow (U)
Denormal/Unnormal Operand (D)	Overflow (O)
Software Assist (SWA) fault	Inexact (I)
	Software Assist (SWA) trap

Interruptions:

Illegal Operation fault	Floating-point Exception fault
Disabled Floating-point Register fault	Floating-point Exception trap

Floating-Point Parallel Negative Multiply

Format: (*qp*) fpnmpy.*sf* $f_1 = f_3, f_4$ pseudo-op of: (*qp*) fpnma.*sf* $f_1 = f_3, f_4$,f0

Description: The pair of products of the pairs of single precision values in the significand fields of FR f_3 and FR f_4 are computed to infinite precision and then negated. The resulting values are then rounded to single precision using the rounding mode specified by FPSR.*sf*.*rc*. The pair of rounded results are stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_3 or FR f_4 is a NaTVal, FR f_1 is set to NaTVal instead of the computed results.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 5-7.

Operation: See "Floating-Point Parallel Negative Multiply Add" on p. 2-98.

Floating-Point Parallel Reciprocal Approximation

Format: (*qp*) fprcpa.*sf* $f_1, p_2 = f_2, f_3$ F6

Description: If PR *qp* is 0, PR p_2 is cleared and FR f_1 remains unchanged.

If PR *qp* is 1, the following will occur:

- Each half of the significand of FR f_1 is either set to an approximation (with a relative error $<2^{-8.886}$) of the reciprocal of the corresponding half of FR f_3, or set to the IEEE-754 mandated response for the quotient FR f_2/FR f_3 of the corresponding half — if that half of FR f_2 or of FR f_3 is in the set {-Infinity, -0, +0, +Infinity, NaN}.
- If either half of FR f_1 is set to the IEEE-754 mandated quotient, or is set to an approximation of the reciprocal which may cause the Newton-Raphson iterations to fail to produce the correct IEEE-754 divide result, then PR p_2 is set to 0, otherwise it is set to 1.

  For correct IEEE divide results, when PR p_2 is cleared, user software is expected to compute the quotient (FR f_2/FR f_3) for each half (using the non-parallel `frcpa` instruction), and merge the results into FR f_1, keeping PR p_2 cleared.
- The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).
- If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result, and PR p_2 is cleared.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result_pair = fprcpa_exception_fault_check(f2, f3, sf,
                                                &tmp_fp_env, &limits_check);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi) ||
            limits_check.hi_fr3) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
            tmp_pred_hi = 0;
        } else {
            num = fp_normalize(fp_reg_read_hi(f2));
            den = fp_normalize(fp_reg_read_hi(f3));
            if (fp_is_inf(num) && fp_is_finite(den)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_hi = 0;
            } else if (fp_is_finite(num) && fp_is_inf(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_hi = 0;
            } else if (fp_is_zero(num) && fp_is_finite(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_hi = 0;
```

```
            } else {
                tmp_res = fp_ieee_recip(den);
                if (limits_check.hi_fr2_or_quot)
                    tmp_pred_hi = 0;
                else
                    tmp_pred_hi = 1;
            }
            tmp_res_hi = fp_single(tmp_res);
        }
        if (fp_is_nan_or_inf(tmp_default_result_pair.lo) ||
            limits_check.lo_fr3) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
            tmp_pred_lo = 0;
        } else {
            num = fp_normalize(fp_reg_read_lo(f2));
            den = fp_normalize(fp_reg_read_lo(f3));
            if (fp_is_inf(num) && fp_is_finite(den)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_lo = 0;
            } else if (fp_is_finite(num) && fp_is_inf(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_lo = 0;
            } else if (fp_is_zero(num) && fp_is_finite(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_lo = 0;
            } else {
                tmp_res = fp_ieee_recip(den);
                if (limits_check.lo_fr2_or_quot)
                    tmp_pred_lo = 0;
                else
                    tmp_pred_lo = 1;
            }
            tmp_res_lo = fp_single(tmp_res);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
        PR[p2] = tmp_pred_hi && tmp_pred_lo;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}
```

FP Exceptions: Invalid Operation (V)
Zero Divide (Z)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Floating-point Exception fault
Disabled Floating-point Register fault

Floating-Point Parallel Reciprocal Square Root Approximation

Format: (*qp*) fprsqrta.*sf* $f_1, p_2 = f_3$ F7

Description: If PR *qp* is 0, PR p_2 is cleared and FR f_1 remains unchanged.

If PR *qp* is 1, the following will occur:

- Each half of the significand of FR f_1 is either set to an approximation (with a relative error $<2^{-8.831}$) of the reciprocal square root of the corresponding half of FR f_3, or set to the IEEE-754 compliant response for the reciprocal square root of the corresponding half of FR f_3 — if that half of FR f_3 is in the set {-Infinity, -Finite, -0, +0, +Infinity, NaN}.
- If either half of FR f_1 is set to the IEEE-754 mandated reciprocal square root, or is set to an approximation of the reciprocal square root which may cause the Newton-Raphson iterations to fail to produce the correct IEEE-754 square root result, then PR p_2 is set to 0, otherwise it is set to 1.

  For correct IEEE square root results, when PR p_2 is cleared, user software is expected to compute the square root for each half (using the non-parallel `frsqrta` instruction), and merge the results in FR f_1, keeping PR p_2 cleared.
- The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).
- If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result, and PR p_2 is cleared.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f3, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result_pair = fprsqrta_exception_fault_check(f3, sf,
                                                &tmp_fp_env, &limits_check);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
            tmp_pred_hi = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read_hi(f3));
            if (fp_is_zero(tmp_fr3)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = tmp_fr3.sign;
                tmp_pred_hi = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                tmp_res = FP_ZERO;
                tmp_pred_hi = 0;
            } else {
                tmp_res = fp_ieee_recip_sqrt(tmp_fr3);
                if (limits_check.hi)
                    tmp_pred_hi = 0;
                else
                    tmp_pred_hi = 1;
```

```
            }
            tmp_res_hi = fp_single(tmp_res);
        }

        if (fp_is_nan(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
            tmp_pred_lo = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read_lo(f3));
            if (fp_is_zero(tmp_fr3)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = tmp_fr3.sign;
                tmp_pred_lo = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                tmp_res = FP_ZERO;
                tmp_pred_lo = 0;
            } else {
                tmp_res = fp_ieee_recip_sqrt(tmp_fr3);
                if (limits_check.lo)
                    tmp_pred_lo = 0;
                else
                    tmp_pred_lo = 1;
            }
            tmp_res_lo = fp_single(tmp_res);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
        PR[p2] = tmp_pred_hi && tmp_pred_lo;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Floating-point Exception fault
Disabled Floating-point Register fault

Floating-Point Reciprocal Approximation

Format: (*qp*) frcpa.*sf* $f_1, p_2 = f_2, f_3$ F6

Description: If PR *qp* is 0, PR p_2 is cleared and FR f_1 remains unchanged.

If PR *qp* is 1, the following will occur:

- FR f_1 is either set to an approximation (with a relative error $<2^{-8.886}$) of the reciprocal of FR f_3, or to the IEEE-754 mandated quotient of FR f_2/FR f_3 — if either FR f_2 or FR f_3 is in the set {-Infinity, -0, Pseudo-zero, +0, +Infinity, NaN, Unsupported}.
- If FR f_1 is set to the approximation of the reciprocal of FR f_3, then PR p_2 is set to 1; otherwise, it is set to 0.
- If FR f_2 and FR f_3 are such that the approximation of FR f_3's reciprocal may cause the Newton-Raphson iterations to fail to produce the correct IEEE-754 result of FR f_2/FR f_3, then a Floating-point Exception fault for Software Assist occurs.

  System software is expected to compute the IEEE-754 quotient (FR f_2/FR f_3), return the result in FR f_1, and set PR p_2 to 0.
- If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result, and PR p_2 is cleared.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result = frcpa_exception_fault_check(f2, f3, sf,
                                                         &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
            PR[p2] = 0;
        } else {
            num = fp_normalize(fp_reg_read(FR[f2]));
            den = fp_normalize(fp_reg_read(FR[f3]));
            if (fp_is_inf(num) && fp_is_finite(den)) {
                FR[f1] = FP_INFINITY;
                FR[f1].sign = num.sign ^ den.sign;
                PR[p2] = 0;
            } else if (fp_is_finite(num) && fp_is_inf(den)) {
                FR[f1] = FP_ZERO;
                FR[f1].sign = num.sign ^ den.sign;
                PR[p2] = 0;
            } else if (fp_is_zero(num) && fp_is_finite(den)) {
                FR[f1] = FP_ZERO;
                FR[f1].sign = num.sign ^ den.sign;
                PR[p2] = 0;
            } else {
                FR[f1] = fp_ieee_recip(den);
                PR[p2] = 1;
```

```
            }
        }
        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}

// fp_ieee_recip()

fp_ieee_recip(den)
{
    RECIP_TABLE[256] = {
        0x3fc, 0x3f4, 0x3ec, 0x3e4, 0x3dd, 0x3d5, 0x3cd, 0x3c6,
        0x3be, 0x3b7, 0x3af, 0x3a8, 0x3a1, 0x399, 0x392, 0x38b,
        0x384, 0x37d, 0x376, 0x36f, 0x368, 0x361, 0x35b, 0x354,
        0x34d, 0x346, 0x340, 0x339, 0x333, 0x32c, 0x326, 0x320,
        0x319, 0x313, 0x30d, 0x307, 0x300, 0x2fa, 0x2f4, 0x2ee,
        0x2e8, 0x2e2, 0x2dc, 0x2d7, 0x2d1, 0x2cb, 0x2c5, 0x2bf,
        0x2ba, 0x2b4, 0x2af, 0x2a9, 0x2a3, 0x29e, 0x299, 0x293,
        0x28e, 0x288, 0x283, 0x27e, 0x279, 0x273, 0x26e, 0x269,
        0x264, 0x25f, 0x25a, 0x255, 0x250, 0x24b, 0x246, 0x241,
        0x23c, 0x237, 0x232, 0x22e, 0x229, 0x224, 0x21f, 0x21b,
        0x216, 0x211, 0x20d, 0x208, 0x204, 0x1ff, 0x1fb, 0x1f6,
        0x1f2, 0x1ed, 0x1e9, 0x1e5, 0x1e0, 0x1dc, 0x1d8, 0x1d4,
        0x1cf, 0x1cb, 0x1c7, 0x1c3, 0x1bf, 0x1bb, 0x1b6, 0x1b2,
        0x1ae, 0x1aa, 0x1a6, 0x1a2, 0x19e, 0x19a, 0x197, 0x193,
        0x18f, 0x18b, 0x187, 0x183, 0x17f, 0x17c, 0x178, 0x174,
        0x171, 0x16d, 0x169, 0x166, 0x162, 0x15e, 0x15b, 0x157,
        0x154, 0x150, 0x14d, 0x149, 0x146, 0x142, 0x13f, 0x13b,
        0x138, 0x134, 0x131, 0x12e, 0x12a, 0x127, 0x124, 0x120,
        0x11d, 0x11a, 0x117, 0x113, 0x110, 0x10d, 0x10a, 0x107,
        0x103, 0x100, 0x0fd, 0x0fa, 0x0f7, 0x0f4, 0x0f1, 0x0ee,
        0x0eb, 0x0e8, 0x0e5, 0x0e2, 0x0df, 0x0dc, 0x0d9, 0x0d6,
        0x0d3, 0x0d0, 0x0cd, 0x0ca, 0x0c8, 0x0c5, 0x0c2, 0x0bf,
        0x0bc, 0x0b9, 0x0b7, 0x0b4, 0x0b1, 0x0ae, 0x0ac, 0x0a9,
        0x0a6, 0x0a4, 0x0a1, 0x09e, 0x09c, 0x099, 0x096, 0x094,
        0x091, 0x08e, 0x08c, 0x089, 0x087, 0x084, 0x082, 0x07f,
        0x07c, 0x07a, 0x077, 0x075, 0x073, 0x070, 0x06e, 0x06b,
        0x069, 0x066, 0x064, 0x061, 0x05f, 0x05d, 0x05a, 0x058,
        0x056, 0x053, 0x051, 0x04f, 0x04c, 0x04a, 0x048, 0x045,
        0x043, 0x041, 0x03f, 0x03c, 0x03a, 0x038, 0x036, 0x033,
        0x031, 0x02f, 0x02d, 0x02b, 0x029, 0x026, 0x024, 0x022,
        0x020, 0x01e, 0x01c, 0x01a, 0x018, 0x015, 0x013, 0x011,
        0x00f, 0x00d, 0x00b, 0x009, 0x007, 0x005, 0x003, 0x001,
    };

    tmp_index = den.significand{62:55};
    tmp_res.significand = (1 << 63) | (RECIP_TABLE[tmp_index] << 53);
    tmp_res.exponent = FP_REG_EXP_ONES - 2 - den.exponent;
    tmp_res.sign = den.sign;
    return (tmp_res);
}
```

FP Exceptions: Invalid Operation (V)
Zero Divide (Z)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: 0Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

Floating-Point Reciprocal Square Root Approximation

Format: (*qp*) frsqrta.*sf* f_1, p_2 = f_3 F7

Description: If PR *qp* is 0, PR p_2 is cleared and FR f_1 remains unchanged.

If PR *qp* is 1, the following will occur:

- FR f_1 is either set to an approximation (with a relative error $<2^{-8.831}$) of the reciprocal square root of FR f_3, or set to the IEEE-754 mandated square root of FR f_3 — if FR f_3 is in the set {-Infinity, -Finite, -0, Pseudo-zero, +0, +Infinity, NaN, Unsupported}.
- If FR f_1 is set to an approximation of the reciprocal square root of FR f_3, then PR p_2 is set to 1; otherwise, it is set to 0.
- If FR f_3 is such the approximation of its reciprocal square root may cause the Newton-Raphson iterations to fail to produce the correct IEEE-754 square root result, then a Floating-point Exception fault for Software Assist occurs.

  System software is expected to compute the IEEE-754 square root, return the result in FR f_1, and set PR p_2 to 0.
- If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result, and PR p_2 is cleared.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f3, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result = frsqrta_exception_fault_check(f3, sf,
                                                           &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result)) {
            FR[f1] = tmp_default_result;
            PR[p2] = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read(FR[f3]));
            if (fp_is_zero(tmp_fr3)) {
                FR[f1] = tmp_fr3;
                PR[p2] = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                FR[f1] = tmp_fr3;
                PR[p2] = 0;
            } else {
                FR[f1] = fp_ieee_recip_sqrt(tmp_fr3);
                PR[p2] = 1;
            }
        }
        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
```

```
}

// fp_ieee_recip_sqrt()

fp_ieee_recip_sqrt(root)
{
    RECIP_SQRT_TABLE[256] = {
        0x1a5, 0x1a0, 0x19a, 0x195, 0x18f, 0x18a, 0x185, 0x180,
        0x17a, 0x175, 0x170, 0x16b, 0x166, 0x161, 0x15d, 0x158,
        0x153, 0x14e, 0x14a, 0x145, 0x140, 0x13c, 0x138, 0x133,
        0x12f, 0x12a, 0x126, 0x122, 0x11e, 0x11a, 0x115, 0x111,
        0x10d, 0x109, 0x105, 0x101, 0x0fd, 0x0fa, 0x0f6, 0x0f2,
        0x0ee, 0x0ea, 0x0e7, 0x0e3, 0x0df, 0x0dc, 0x0d8, 0x0d5,
        0x0d1, 0x0ce, 0x0ca, 0x0c7, 0x0c3, 0x0c0, 0x0bd, 0x0b9,
        0x0b6, 0x0b3, 0x0b0, 0x0ad, 0x0a9, 0x0a6, 0x0a3, 0x0a0,
        0x09d, 0x09a, 0x097, 0x094, 0x091, 0x08e, 0x08b, 0x088,
        0x085, 0x082, 0x07f, 0x07d, 0x07a, 0x077, 0x074, 0x071,
        0x06f, 0x06c, 0x069, 0x067, 0x064, 0x061, 0x05f, 0x05c,
        0x05a, 0x057, 0x054, 0x052, 0x04f, 0x04d, 0x04a, 0x048,
        0x045, 0x043, 0x041, 0x03e, 0x03c, 0x03a, 0x037, 0x035,
        0x033, 0x030, 0x02e, 0x02c, 0x029, 0x027, 0x025, 0x023,
        0x020, 0x01e, 0x01c, 0x01a, 0x018, 0x016, 0x014, 0x011,
        0x00f, 0x00d, 0x00b, 0x009, 0x007, 0x005, 0x003, 0x001,
        0x3fc, 0x3f4, 0x3ec, 0x3e5, 0x3dd, 0x3d5, 0x3ce, 0x3c7,
        0x3bf, 0x3b8, 0x3b1, 0x3aa, 0x3a3, 0x39c, 0x395, 0x38e,
        0x388, 0x381, 0x37a, 0x374, 0x36d, 0x367, 0x361, 0x35a,
        0x354, 0x34e, 0x348, 0x342, 0x33c, 0x336, 0x330, 0x32b,
        0x325, 0x31f, 0x31a, 0x314, 0x30f, 0x309, 0x304, 0x2fe,
        0x2f9, 0x2f4, 0x2ee, 0x2e9, 0x2e4, 0x2df, 0x2da, 0x2d5,
        0x2d0, 0x2cb, 0x2c6, 0x2c1, 0x2bd, 0x2b8, 0x2b3, 0x2ae,
        0x2aa, 0x2a5, 0x2a1, 0x29c, 0x298, 0x293, 0x28f, 0x28a,
        0x286, 0x282, 0x27d, 0x279, 0x275, 0x271, 0x26d, 0x268,
        0x264, 0x260, 0x25c, 0x258, 0x254, 0x250, 0x24c, 0x249,
        0x245, 0x241, 0x23d, 0x239, 0x235, 0x232, 0x22e, 0x22a,
        0x227, 0x223, 0x220, 0x21c, 0x218, 0x215, 0x211, 0x20e,
        0x20a, 0x207, 0x204, 0x200, 0x1fd, 0x1f9, 0x1f6, 0x1f3,
        0x1f0, 0x1ec, 0x1e9, 0x1e6, 0x1e3, 0x1df, 0x1dc, 0x1d9,
        0x1d6, 0x1d3, 0x1d0, 0x1cd, 0x1ca, 0x1c7, 0x1c4, 0x1c1,
        0x1be, 0x1bb, 0x1b8, 0x1b5, 0x1b2, 0x1af, 0x1ac, 0x1aa,
    };

    tmp_index = (root.exponent{0} << 7) | root.significand{62:56};
    tmp_res.significand = (1 << 63) | (RECIP_SQRT_TABLE[tmp_index] << 53);
    tmp_res.exponent = FP_REG_EXP_HALF -
                       ((root.exponent - FP_REG_BIAS) >> 1);
    tmp_res.sign = FP_SIGN_POSITIVE;
    return (tmp_res);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Floating-point Exception fault
Disabled Floating-point Register fault

Floating-Point Select

Format: (*qp*) fselect $f_1 = f_3, f_4, f_2$ F3

Description: The significand field of FR f_3 is logically AND-ed with the significand field of FR f_2 and the significand field of FR f_4 is logically AND-ed with the one's complement of the significand field of FR f_2. The two results are logically OR-ed together. The result is placed in the significand field of FR f_1.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E). The sign bit field of FR f_1 is set to positive (0).

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand   = (FR[f3].significand & FR[f2].significand)
                             | (FR[f4].significand & ~FR[f2].significand);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Floating-Point Set Controls

Format: (*qp*) fsetc.*sf* *amask*$_7$, *omask*$_7$ F12

Description: The status field's control bits are initialized to the value obtained by logically AND-ing the sf0.controls and *amask*$_7$ immediate field and logically OR-ing the *omask*$_7$ immediate field.

The mnemonic values for *sf* are given in Table 2-22 on page 2-42.

Operation:

```
if (PR[qp]) {
    tmp_controls = (AR[FPSR].sf0.controls & amask7) | omask7;
    if (is_reserved_field(FSETC, sf, tmp_controls))
        reserved_register_field_fault();
    fp_set_sf_controls(sf, tmp_controls);
}
```

FP Exceptions: None

Interruptions: Reserved Register/Field fault

Floating-Point Subtract

Format: (*qp*) fsub.*pc*.*sf* $f_1 = f_3, f_2$ pseudo-op of: (*qp*) fms.*pc*.*sf* $f_1 = f_3$, f1, f_2

Description: FR f_2 is subtracted from FR f_3 (computed to infinite precision), rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*, and placed in FR f_1.

If either FR f_3 or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-21 on page 2-42. The mnemonic values for *sf* are given in Table 2-22 on page 2-42. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation: See "Floating-Point Multiply Subtract" on p. 2-69.

Floating-Point Swap

Format:

(*qp*) fswap $f_1 = f_2, f_3$	swap_form	F9
(*qp*) fswap.nl $f_1 = f_2, f_3$	swap_nl_form	F9
(*qp*) fswap.nr $f_1 = f_2, f_3$	swap_nr_form	F9

Description: For the swap_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3. The concatenated pair is then swapped.

For the swap_nl_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3. The concatenated pair is then swapped, and the left single precision value is negated.

For the swap_nr_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3. The concatenated pair is then swapped, and the right single precision value is negated.

For all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For all forms, if either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-17. Floating-point Swap

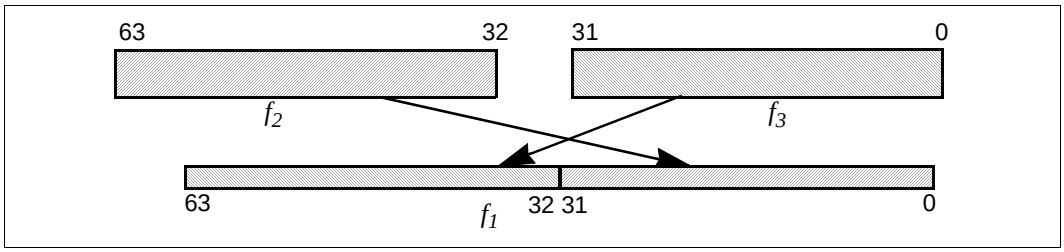
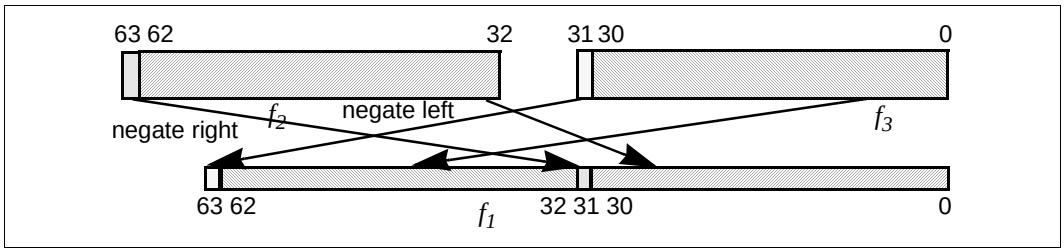


Figure 2-18. Floating-point Swap Negate Left or Right



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (swap_form) {
            tmp_res_hi = FR[f3].significand{31:0};
            tmp_res_lo = FR[f2].significand{63:32};
        } else if (swap_nl_form) {
            tmp_res_hi = (!FR[f3].significand{31} << 31)
                       | (FR[f3].significand{30:0});
            tmp_res_lo = FR[f2].significand{63:32};
        } else { // swap_nr_form
            tmp_res_hi = FR[f3].significand{31:0};
            tmp_res_lo = (!FR[f2].significand{63} << 31)
                       | (FR[f2].significand{62:32});
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Floating-Point Sign Extend

Format:

(*qp*) fsxt.l $f_1 = f_2, f_3$	sxt_l_form	F9
(*qp*) fsxt.r $f_1 = f_2, f_3$	sxt_r_form	F9

Description: For the sxt_l_form (sxt_r_form), the sign of the left (right) single precision value in FR f_2 is extended to 32-bits and is concatenated with the left (right) single precision value in FR f_3.

For all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For all forms, if either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-19. Floating-point Sign Extend Left

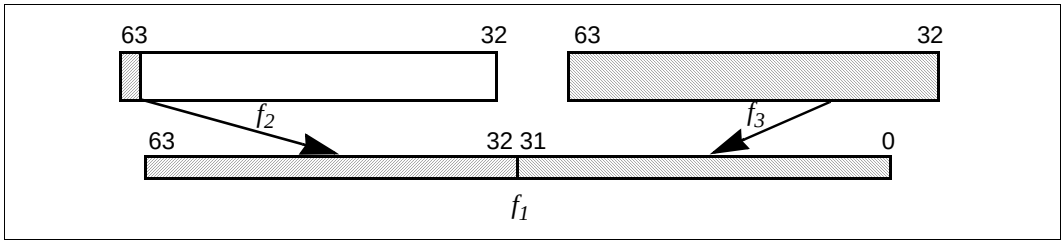
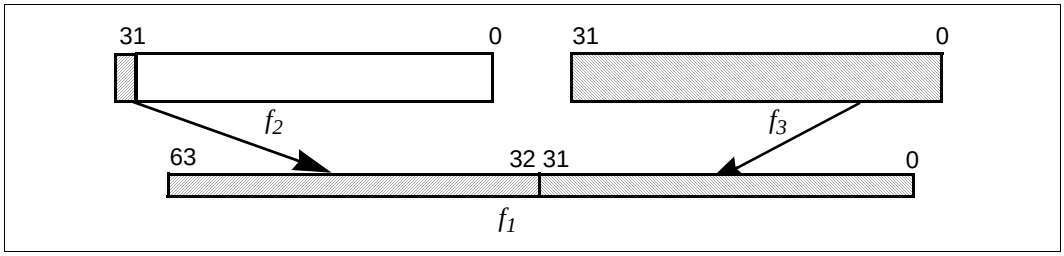


Figure 2-20. Floating-point Sign Extend Right



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (sxt_l_form) {
            tmp_res_hi = (FR[f2].significand{63} ? 0xFFFFFFFF : 0x00000000);
            tmp_res_lo = FR[f3].significand{63:32};
        } else {                                        // sxt_r_form
            tmp_res_hi = (FR[f2].significand{31} ? 0xFFFFFFFF : 0x00000000);
            tmp_res_lo = FR[f3].significand{31:0};
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Flush Write Buffers

Format: (*qp*) fwb M24

Description: The processor is instructed to expedite flushing of any pending stores held in write or coalescing buffers. Since this operation is a hint, the processor may or may not take any action and actually flush any outstanding stores. The processor gives no indication when flushing of any prior stores is completed. An `fwb` instruction does not ensure ordering of stores, since later stores may be flushed before prior stores.

To ensure prior coalesced stores are made visible before later stores, software must issue a release operation between stores (see Table 4-14 on page 4-35 for a list of release operations).

This instruction can be used to help ensure stores held in write or coalescing buffers are not delayed for long periods or to expedite high priority stores out of the processors.

Operation:

```
if (PR[qp]) {
    mem_flush_pending_stores();
}
```

Interruptions: None

Floating-Point Exclusive Or

Format: (*qp*) fxor $f_1 = f_2, f_3$ F9

Description: The bit-wise logical exclusive-OR of the significand fields of FR f_2 and FR f_3 is computed. The resulting value is stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either of FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand ^ FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Get Floating-Point Value or Exponent or Significand

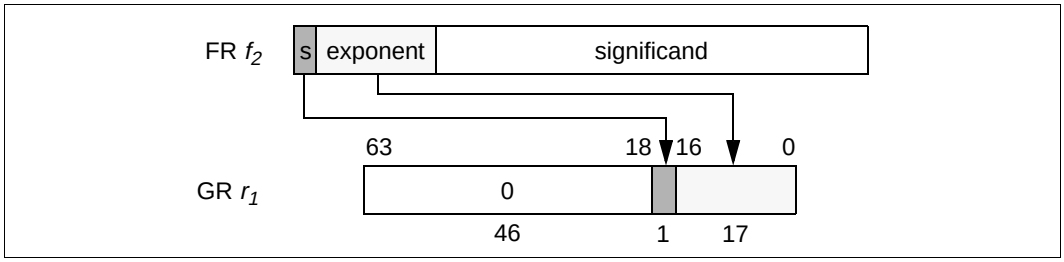
Format:

(*qp*) getf.s $r_1 = f_2$	single_form	M19
(*qp*) getf.d $r_1 = f_2$	double_form	M19
(*qp*) getf.exp $r_1 = f_2$	exponent_form	M19
(*qp*) getf.sig $r_1 = f_2$	significand_form	M19

Description: In the single and double forms, the value in FR f_2 is converted into a single precision (single_form) or double precision (double_form) memory representation and placed in GR r_1. In the single_form, the most-significant 32 bits of GR r_1 are set to 0.

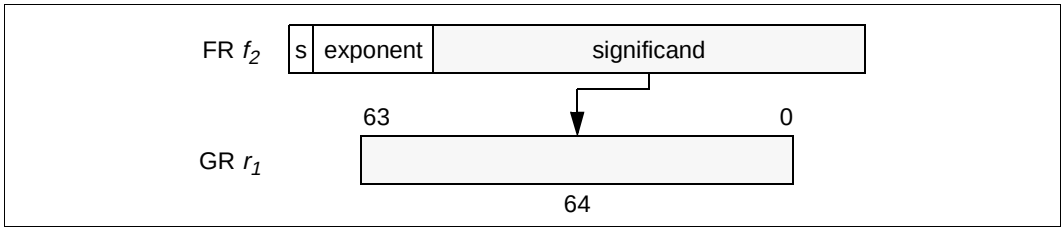
In the exponent_form, the exponent field of FR f_2 is copied to bits 16:0 of GR r_1 and the sign bit of the value in FR f_2 is copied to bit 17 of GR r_1. The most-significant 46-bits of GR r_1 are set to zero.

Figure 2-21. Function of getf.exp



In the significand_form, the significand field of the value in FR f_2 is copied to GR r_1.

Figure 2-22. Function of getf.sig



For all forms, if FR f_2 contains a NaTVal, then the NaT bit corresponding to GR r_1 is set to 1.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);
    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (single_form) {
        GR[r1]{31:0} = fp_fr_to_mem_format(FR[f2], 4, 0);
        GR[r1]{63:32} = 0;
    } else if (double_form) {
        GR[r1] = fp_fr_to_mem_format(FR[f2], 8, 0);
    } else if (exponent_form) {
        GR[r1]{63:18} = 0;
        GR[r1]{16:0} = FR[f2].exponent;
        GR[r1]{17} = FR[f2].sign;
    } else // significand_form
        GR[r1] = FR[f2].significand;
    if (fp_is_natval(FR[f2]))
        GR[r1].nat = 1;
```

```
    else
        GR[r1].nat = 0;
}
```

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Invalidate ALAT

Format:

(*qp*) invala	complete_form	M24
(*qp*) invala.e r_1	gr_form, entry_form	M26
(*qp*) invala.e f_1	fr_form, entry_form	M27

Description: The selected entry or entries in the ALAT are invalidated.

In the complete_form, all ALAT entries are invalidated. In the entry_form, the ALAT is queried using the general register specifier r_1 (gr_form), or the floating-point register specifier f_1 (fr_form), and if any ALAT entry matches, it is invalidated.

Operation:

```
if (PR[qp]) {
    if (complete_form)
        alat_inval();
    else { // entry_form
        if (gr_form)
            alat_inval_single_entry(GENERAL, r1);
        else // fr_form
            alat_inval_single_entry(FLOAT, f1);
    }
}
```

Interruptions: None

Insert Translation Cache

Format:

(*qp*) itc.i r_2	instruction_form	M41
(*qp*) itc.d r_2	data_form	M41

Description: An entry is inserted into the instruction or data translation cache. GR r_2 specifies the physical address portion of the translation. ITIR specifies the protection key, page size and additional information. The virtual address is specified by the IFA register and the region register is selected by IFA{63:61}. The processor determines which entry to replace based on an implementation-specific replacement algorithm.

The visibility of the `itc` instruction to externally generated purges (`ptc.g`, `ptc.ga`) must occur before subsequent memory operations. From a software perspective, this is similar to acquire semantics. Serialization is still required to observe the side-effects of a translation being present.

`itc` must be the last instruction in an instruction group; otherwise, its behavior (including its ordering semantics) is undefined.

The TLB is first purged of any overlapping entries as specified by Table 4-1 on page 4-7.

This instruction can only be executed at the most privileged level, and when PSR.ic is zero.

To ensure forward progress, software must ensure that PSR.ic remains 0 until `rfi`-ing to the instruction that requires the translation.

Operation:

```
if (PR[qp]) {
    if (!followed_by_stop())
        undefined_behavior();
    if (PSR.ic)
        illegal_operation_fault();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r2].nat)
        register_nat_consumption_fault(0);

    tmp_size = CR[ITIR].ps;
    tmp_va = CR[IFA]{60:0};
    tmp_rid = RR[CR[IFA]{63:61}].rid;
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    if (is_reserved_field(TLB_TYPE, GR[r2], CR[ITIR]))
        reserved_register_field_fault();
    if (unimplemented_virtual_address(CR[IFA]))
        unimplemented_data_address_fault(0);

    if (instruction_form) {
        tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        slot = tlb_replacement_algorithm(ITC_TYPE);
        tlb_insert_inst(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TC);
    } else {                                          // data_form
        tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        slot = tlb_replacement_algorithm(DTC_TYPE);
        tlb_insert_data(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TC);
    }
}
```

Interruptions: Machine Check abort
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Unimplemented Data Address fault

Serialization: For the instruction_form, software must issue an instruction serialization operation before a dependent instruction fetch access. For the data_form, software must issue a data serialization operation before issuing a data access or non-access reference dependent on the new translation.

Insert Translation Register

Format:

(*qp*) itr.i itr[r_3] = r_2	instruction_form	M42
(*qp*) itr.d dtr[r_3] = r_2	data_form	M42

Description: A translation is inserted into the instruction or data translation register specified by the contents of GR r_3. GR r_2 specifies the physical address portion of the translation. ITIR specifies the protection key, page size and additional information. The virtual address is specified by the IFA register and the region register is selected by IFA{63:61}.

As described in Table 4-1 on page 4-7, the TLB is first purged of any entries that overlap with the newly inserted translation. The translation previously contained in the TR slot specified by GR r_3 is not purged from the processor's TLBs. To remove previous TR translations, software must use explicit `ptr` instructions.

This instruction can only be executed at the most privileged level, and when PSR.ic is zero.

Operation:

```
if (PR[qp]) {
    if (PSR.ic)
        illegal_operation_fault();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);

    slot = GR[r3]{7:0};
    tmp_size = CR[ITIR].ps;
    tmp_va = CR[IFA]{60:0};
    tmp_rid = RR[CR[IFA]{63:61}].rid;
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    tmp_tr_type = instruction_form ? ITR_TYPE : DTR_TYPE;

    if (is_reserved_reg(tmp_tr_type, slot))
        reserved_register_field_fault();
    if (is_reserved_field(TLB_TYPE, GR[r2], CR[ITIR]))
        reserved_register_field_fault();
    if (unimplemented_virtual_address(CR[IFA]))
        unimplemented_data_address_fault(0);

    if (instruction_form) {
        tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_insert_inst(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TR);
    } else {                                          // data_form
        tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_insert_data(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TR);
    }
}
```

Interruptions:

Machine Check abort
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Unimplemented Data Address fault

Serialization: For the instruction_form, software must issue an instruction serialization operation before a dependent instruction fetch access. For the data_form, software must issue a data serialization operation before issuing a data access or non-access reference dependent on the new translation.

Notes: The processor may use invalid translation registers for translation cache entries. Performance can be improved on some processor models by ensuring translation registers are allocated beginning at translation register zero and continuing contiguously upwards.

Load

Format:

(*qp*) ld*sz*.*ldtype*.*ldhint* $r_1 = [r_3]$	no_base_update_form	M1
(*qp*) ld*sz*.*ldtype*.*ldhint* $r_1 = [r_3], r_2$	reg_base_update_form	M2
(*qp*) ld*sz*.*ldtype*.*ldhint* $r_1 = [r_3], imm_9$	imm_base_update_form	M3
(*qp*) ld8.fill.*ldhint* $r_1 = [r_3]$	fill_form, no_base_update_form	M1
(*qp*) ld8.fill.*ldhint* $r_1 = [r_3], r_2$	fill_form, reg_base_update_form	M2
(*qp*) ld8.fill.*ldhint* $r_1 = [r_3], imm_9$	fill_form, imm_base_update_form	M3

Description: A value consisting of *sz* bytes is read from memory starting at the address specified by the value in GR r_3. The value is then zero extended and placed in GR r_1. The values of the *sz* completer are given in Table 2-30. The NaT bit corresponding to GR r_1 is cleared, except as described below for speculative loads. The *ldtype* completer specifies special load operations, which are described in Table 2-31.

For the fill_form, an 8-byte value is loaded, and a bit in the UNAT application register is copied into the target register NaT bit. This instruction is used for reloading a spilled register/NaT pair. See Volume 1 for details.

In the base update forms, the value in GR r_3 is added to either a signed immediate value (imm_9) or a value from GR r_2, and the result is placed back in GR r_3. This base register update is done after the load, and does not affect the load address. In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set and no fault is raised.

For more details on ordered, biased, speculative, advanced and check loads see Volume 1.

Table 2-30. *sz* Completers

sz Completer	Bytes Accessed
1	1 byte
2	2 bytes
4	4 bytes
8	8 bytes

Table 2-31. Load Types

ldtype Completer	Interpretation	Special Load Operation
none	Normal load	
s	Speculative load	Certain exceptions may be deferred rather than generating a fault. Deferral causes the target register's NaT bit to be set. The NaT bit is later used to detect deferral.
a	Advanced load	An entry is added to the ALAT. This allows later instructions to check for colliding stores. If the referenced data page has a non-speculative attribute, the target register and NaT bit is cleared, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.

Table 2-31. Load Types (Continued)

ldtype Completer	Interpretation	Special Load Operation
sa	Speculative Advanced load	An entry is added to the ALAT, and certain exceptions may be deferred. Deferral causes the target register's NaT bit to be set, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.
c.nc	Check load - no clear	The ALAT is searched for a matching entry. If found, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a load is performed, and an entry is added to the ALAT (unless the referenced data page has a non-speculative attribute, in which case no ALAT entry is allocated).
c.clr	Check load - clear	The ALAT is searched for a matching entry. If found, the entry is removed, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a clear check load behaves like a normal load.
c.clr.acq	Ordered check load – clear	This type behaves the same as the unordered clear form, except that the ALAT lookup (and resulting load, if no ALAT entry is found) is performed with acquire semantics.
acq	Ordered load	An ordered load is performed with acquire semantics.
bias	Biased load	A hint is provided to the implementation to acquire exclusive ownership of the accessed cache line.

For the non-speculative load types, if NaT bit associated with GR r_3 is 1, a Register NaT Consumption fault is taken. For speculative and speculative advanced loads, no fault is raised, and the exception is deferred. For the base-update calculation, if the NaT bit associated with GR r_2 is 1, the NaT bit associated with GR r_3 is set to 1 and no fault is raised.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 2-32. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *ldhint*. Prefetch and locality hints do not affect program functionality and may be ignored by the implementation.

Table 2-32. Load Hints

ldhint Completer	Interpretation
none	Temporal locality, level 1
nt1	No temporal locality, level 1
nta	No temporal locality, all levels

In the no_base_update form, the value in GR r_3 is not modified and no prefetch hint is implied.

For the base update forms, specifying the same register address in r_1 and r_3 will cause an Illegal Operation fault.

Operation:

```
if (PR[qp]) {
    size = fill_form ? 8 : sz;

    speculative = (ldtype == 's' || ldtype == 'sa');
    advanced = (ldtype == 'a' || ldtype == 'sa');
    check_clear = (ldtype == 'c.clr' || ldtype == 'c.clr.acq');
    check_no_clear = (ldtype == 'c.nc');
    check = check_clear || check_no_clear;
    acquire = (ldtype == 'acq' || ldtype == 'c.clr.acq');
    bias = (ldtype == 'bias') ? BIAS : 0 ;

    itype = READ;
    if (speculative) itype |= SPEC ;
    if (advanced)itype |= ADVANCE ;

    if ((reg_base_update_form || imm_base_update_form) && (r1 == r3))
        illegal_operation_fault();
    check_target_register(r1);
    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);

    if (reg_base_update_form) {
        tmp_r2 = GR[r2];
        tmp_r2nat = GR[r2].nat;
    }

    if (!speculative && GR[r3].nat)              // fault on NaT address
        register_nat_consumption_fault(itype);
    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(GENERAL, r1)) {     // no load on ld.c & ALAT hit
        if (check_clear)        // remove entry on ld.c.clr or ld.c.clr.acq
            alat_inval_single_entry(GENERAL, r1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                  &defer);
            if (!defer) {
                otype = acquire ? ACQUIRE : UNORDERED;
                val = mem_read(paddr, size, UM.be, mattr, otype,
                               bias | ldhint);
            }
        }
        if (check_clear || advanced)             // remove any old ALAT entry
            alat_inval_single_entry(GENERAL, r1);
        if (defer) {
            if (speculative) {
                GR[r1] = natd_gr_read(paddr, size, UM.be, mattr, otype,
                                      bias | ldhint);
                GR[r1].nat = 1;
            } else {
                GR[r1] = 0;                      // ld.a to sequential memory
                GR[r1].nat = 0;
            }
        } else {                                     // execute load normally
            if (fill_form) {                         // fill NaT on ld8.fill
                bit_pos = GR[r3]{8:3};
```

```
                GR[r1] = val;
                GR[r1].nat = AR[UNAT]{bit_pos};
            } else {                                    // clear NaT on other types
                GR[r1] = zero_ext(val, size * 8);
                GR[r1].nat = 0;
            }
            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                        // add entry to ALAT
                alat_write(GENERAL, r1, paddr, size);
        }
    }

    if (imm_base_update_form) {                         // update base register
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = GR[r3].nat;
    } else if (reg_base_update_form) {
        GR[r3] = GR[r3] + tmp_r2;
        GR[r3].nat = GR[r3].nat || tmp_r2nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
        mem_implicit_prefetch(GR[r3], ldhint | bias, itype);
}
```

Interruptions: Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault

Floating-Point Load

Format:

(*qp*) ldf*fsz*.*fldtype*.*ldhint* $f_1 = [r_3]$	no_base_update_form	M6
(*qp*) ldf*fsz*.*fldtype*.*ldhint* $f_1 = [r_3], r_2$	reg_base_update_form	M7
(*qp*) ldf*fsz*.*fldtype*.*ldhint* $f_1 = [r_3], imm_9$	imm_base_update_form	M8
(*qp*) ldf8.*fldtype*.*ldhint* $f_1 = [r_3]$	integer_form, no_base_update_form	M6
(*qp*) ldf8.*fldtype*.*ldhint* $f_1 = [r_3], r_2$	integer_form, reg_base_update_form	M7
(*qp*) ldf8.*fldtype*.*ldhint* $f_1 = [r_3], imm_9$	integer_form, imm_base_update_form	M8
(*qp*) ldf.fill.*ldhint* $f_1 = [r_3]$	fill_form, no_base_update_form	M6
(*qp*) ldf.fill.*ldhint* $f_1 = [r_3], r_2$	fill_form, reg_base_update_form	M7
(*qp*) ldf.fill.*ldhint* $f_1 = [r_3], imm_9$	fill_form, imm_base_update_form	M8

Description:

A value consisting of *fsz* bytes is read from memory starting at the address specified by the value in GR r_3. The value is then converted into the floating-point register format and placed in FR f_1. See Volume 1 for details on conversion to floating-point register format. The values of the *fsz* completer are given in Table 2-33. The *fldtype* completer specifies special load operations, which are described in Table 2-34.

For the integer_form, an 8-byte value is loaded and placed in the significand field of FR f_1 without conversion. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For the fill_form, a 16-byte value is loaded, and the appropriate fields are placed in FR f_1 without conversion. This instruction is used for reloading a spilled register. See Volume 1 for details.

In the base update forms, the value in GR r_3 is added to either a signed immediate value (imm_9) or a value from GR r_2, and the result is placed back in GR r_3. This base register update is done after the load, and does not affect the load address. In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set and no fault is raised.

For more details on speculative, advanced and check loads see Volume 1.

For the non-speculative load types, if NaT bit associated with GR r_3 is 1, a Register NaT Consumption fault is taken. For speculative and speculative advanced loads, no fault is raised, and the exception is deferred. For the base-update calculation, if the NaT bit associated with GR r_2 is 1, the NaT bit associated with GR r_3 is set to 1 and no fault is raised.

Table 2-33. *fsz* Completers

fsz Completer	Bytes Accessed	Memory Format
s	4 bytes	Single precision
d	8 bytes	Double precision
e	10 bytes	Extended precision

Table 2-34. FP Load Types

fldtype Completer	Interpretation	Special Load Operation
none	Normal load	
s	Speculative load	Certain exceptions may be deferred rather than generating a fault. Deferral causes NaTVal to be placed in the target register. The NaTVal value is later used to detect deferral.

Table 2-34. FP Load Types (Continued)

fldtype Completer	Interpretation	Special Load Operation
a	Advanced load	An entry is added to the ALAT. This allows later instructions to check for colliding stores. If the referenced data page has a non-speculative attribute, no ALAT entry is added to the ALAT and the target register is set as follows: for the integer_form, the exponent is set to 0x1003E and the sign and significand are set to zero; for all other forms, the sign, exponent and significand are set to zero. The absence of an ALAT entry is later used to detect deferral or collision.
sa	Speculative Advanced load	An entry is added to the ALAT, and certain exceptions may be deferred. Deferral causes NaTVal to be placed in the target register, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.
c.nc	Check load - no clear	The ALAT is searched for a matching entry. If found, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a load is performed, and an entry is added to the ALAT (unless the referenced data page has a non-speculative attribute, in which case no ALAT entry is allocated).
c.clr	Check load – clear	The ALAT is searched for a matching entry. If found, the entry is removed, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a clear check load behaves like a normal load.

The value of the *ldhint* modifier specifies the locality of the memory access. The mnemonic values of *ldhint* are given in Table 2-32 on page 2-127. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *ldhint*. Prefetch and locality hints do not affect program functionality and may be ignored by the implementation. In the no_base_update form, the value in GR r_3 is not modified and no prefetch hint is implied.

The PSR.mfl and PSR.mfh bits are updated to reflect the modification of FR f_1.

Hardware support for `ldfe` (10-byte) instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `ldfe` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted. The fault is delivered only on the normal, advanced, and check load flavors. Control-speculative flavors of `ldfe` always defer the Unsupported Data Reference fault.

Operation:

```
if (PR[qp]) {
    size = (fill_form ? 16 : (integer_form ? 8 : fsz));
    speculative = (fldtype == 's' || fldtype == 'sa');
    advanced = (fldtype == 'a' || fldtype == 'sa');
    check_clear = (fldtype == 'c.clr' );
    check_no_clear = (fldtype == 'c.nc');
    check = check_clear || check_no_clear;

    itype = READ;
    if (speculative) itype |= SPEC ;
    if (advanced) itype |= ADVANCE ;

    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, itype);

    if (!speculative && GR[r3].nat)              // fault on NaT address
        register_nat_consumption_fault(itype);

    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(FLOAT, f1)) {       // no load on ldf.c & ALAT hit
        if (check_clear)                      // remove entry on ldf.c.clr
            alat_inval_single_entry(FLOAT, f1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                  &defer);
            if (!defer)
                val = mem_read(paddr, size, UM.be, mattr, UNORDERED, ldhint);
        }
        if (check_clear || advanced)             // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
        if (speculative && defer) {
            FR[f1] = NATVAL;
        } else if (advanced && !speculative && defer) {
            FR[f1] = (integer_form ? FP_INT_ZERO : FP_ZERO);
        } else {                                 // execute load normally
            FR[f1] = fp_mem_to_fr_format(val, size, integer_form);

            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                 // add entry to ALAT
                alat_write(FLOAT, f1, paddr, size);
        }
    }

    if (imm_base_update_form) {                  // update base register
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = GR[r3].nat;
    } else if (reg_base_update_form) {
        GR[r3] = GR[r3] + GR[r2];
        GR[r3].nat = GR[r3].nat || GR[r2].nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
        mem_implicit_prefetch(GR[r3], ldhint, itype);

    fp_update_psr(f1);
}
```

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

Floating-Point Load Pair

Format:

(*qp*) ldfps.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$	single_form, no_base_update_form	M11
(*qp*) ldfps.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$, 8	single_form, base_update_form	M12
(*qp*) ldfpd.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$	double_form, no_base_update_form	M11
(*qp*) ldfpd.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$, 16	double_form, base_update_form	M12
(*qp*) ldfp8.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$	integer_form, no_base_update_form	M11
(*qp*) ldfp8.*fldtype*.*ldhint* $f_1, f_2 = [r_3]$, 16	integer_form, base_update_form	M12

Description:

Eight (single_form) or sixteen (double_form/integer_form) bytes are read from memory starting at the address specified by the value in GR r_3. The value read is treated as a contiguous pair of floating-point numbers for the single_form/double_form and as integer/Parallel FP data for the integer_form. Each number is converted into the floating-point register format. The value at the lowest address is placed in FR f_1, and the value at the highest address is placed in FR f_2. See Volume 1 for details on conversion to floating-point register format. The *fldtype* completer specifies special load operations, which are described in Table 2-34 on page 2-130.

For more details on speculative, advanced and check loads see Volume 1.

For the non-speculative load types, if NaT bit associated with GR r_3 is 1, a Register NaT Consumption fault is taken. For speculative and speculative advanced loads, no fault is raised, and the exception is deferred.

In the base_update_form, the value in GR r_3 is added to an implied immediate value (equal to double the data size) and the result is placed back in GR r_3. This base register update is done after the load, and does not affect the load address.

The value of the *ldhint* modifier specifies the locality of the memory access. The mnemonic values of *ldhint* are given in Table 2-32 on page 2-127. A prefetch hint is implied in the base update form. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *ldhint*. Prefetch and locality hints do not affect program functionality and may be ignored by the implementation. In the no_base_update form, the value in GR r_3 is not modified and no prefetch hint is implied.

The PSR.mfl and PSR.mfh bits are updated to reflect the modification of FR f_1 and FR f_2.

There is a restriction on the choice of target registers. Register specifiers f_1 and f_2 must specify one odd-numbered physical FR and one even-numbered physical FR. Specifying two odd or two even registers will cause an Illegal Operation fault to be raised. The restriction is on physical register numbers after register rotation. This means that if f_1 and f_2 both specify static registers or both specify rotating registers, then f_1 and f_2 must be odd/even or even/odd. If f_1 and f_2 specify one static and one rotating register, the restriction depends on CFM.rrb.fr. If CFM.rrb.fr is even, the restriction is the same; f_1 and f_2 must be odd/even or even/odd. If CFM.rrb.fr is odd, then f_1 and f_2 must be even/even or odd/odd. Specifying one static and one rotating register should only be done when CFM.rrb.fr will have a predictable value (such as 0).

Operation:

```
if (PR[qp]) {
    size = single_form ? 8 : 16;

    speculative = (fldtype == 's' || fldtype == 'sa');
    advanced = (fldtype == 'a' || fldtype == 'sa');
    check_clear = (fldtype == 'c.clr');
    check_no_clear = (fldtype == 'c.nc');
    check = check_clear || check_no_clear;

    itype = READ;
    if (speculative) itype |= SPEC;
    if (advanced) itype |= ADVANCE;

    if (fp_reg_bank_conflict(f1, f2))
        illegal_operation_fault();

    if (base_update_form)
        check_target_register(r3);

    fp_check_target_register(f1);
    fp_check_target_register(f2);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, itype);

    if (!speculative && GR[r3].nat)              // fault on NaT address
        register_nat_consumption_fault(itype);

    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(FLOAT, f1)) {       // no load on ldfp.c & ALAT hit
        if (check_clear)                      // remove entry on ldfp.c.clr
            alat_inval_single_entry(FLOAT, f1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                  &defer);
            if (!defer)
                val = mem_read(paddr, size, UM.be, mattr, UNORDERED, ldhint);
        }
        if (check_clear || advanced)             // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
        if (speculative && defer) {
            FR[f1] = NATVAL;
            FR[f2] = NATVAL;
        } else if (advanced && !speculative && defer) {
            FR[f1] = (integer_form ? FP_INT_ZERO : FP_ZERO);
            FR[f2] = (integer_form ? FP_INT_ZERO : FP_ZERO);
        } else {                                 // execute load normally
            if (UM.be) {
                FR[f1] = fp_mem_to_fr_format(val u>> (size/2*8), size/2,
                       integer_form);
                FR[f2] = fp_mem_to_fr_format(val, size/2, integer_form);
            } else {
                FR[f1] = fp_mem_to_fr_format(val, size/2, integer_form);
                FR[f2] = fp_mem_to_fr_format(val u>> (size/2*8), size/2,
                                             integer_form);
            }

            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                 // add entry to ALAT
                alat_write(FLOAT, f1, paddr, size);
        }
```

```
    }

    if (base_update_form) {                        // update base register
        GR[r3] = GR[r3] + size;
        GR[r3].nat = GR[r3].nat;
        if (!GR[r3].nat)
            mem_implicit_prefetch(GR[r3], ldhint, itype);
    }

    fp_update_psr(f1);
    fp_update_psr(f2);
}
```

Interruptions:

Illegal Operation fault
Disabled Floating-point Register fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault

Line Prefetch

Format:

(*qp*) lfetch.*lftype*.*lfhint* [r_3]	no_base_update_form	M13
(*qp*) lfetch.*lftype*.*lfhint* [r_3], r_2	reg_base_update_form	M14
(*qp*) lfetch.*lftype*.*lfhint* [r_3], imm_9	imm_base_update_form	M15
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3]	no_base_update_form, exclusive_form	M13
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], r_2	reg_base_update_form, exclusive_form	M14
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], imm_9	imm_base_update_form, exclusive_form	M15

Description: The line containing the address specified by the value in GR r_3 is moved to the highest level of the data memory hierarchy. The value of the *lfhint* modifier specifies the locality of the memory access; see Section 4.4 in Volume 1 for details. The mnemonic values of *lfhint* are given in Table 2-36.

The behavior of the memory read is also determined by the memory attribute associated with the accessed page. For details, refer to Volume 2. Line size is implementation dependent but must be a power of two greater than or equal to 32 bytes. In the exclusive form, the cache line is allowed to be marked in an exclusive state. This qualifier is used when the program expects soon to modify a location in that line. If the memory attribute for the page containing the line is not cacheable, then no reference is made.

The completer, *lftype*, specifies whether or not the instruction raises faults normally associated with a regular load. Table 2-35 defines these two options.

Table 2-35. *lftype* Mnemonic Values

lftype Mnemonic	Interpretation
none	Ignore faults
fault	Raise faults

In the base update forms, after being used to address memory, the value in GR r_3 is incremented by either the sign-extended value in imm_9 (in the imm_base_update_form) or the value in GR r_2 (in the reg_base_update_form). In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set – no fault is raised.

In the reg_base_update_form and the imm_base_update_form, if the NaT bit corresponding to GR r_3 is clear, then the address specified by the value in GR r_3 after the post-increment acts as a hint to implicitly prefetch the indicated cache line. This implicit prefetch uses the locality hints specified by *lfhint*. The implicit prefetch does not affect program functionality, does not raise any faults, and may be ignored by the implementation.

In the no_base_update_form, the value in GR r_3 is not modified and no implicit prefetch hint is implied.

If the NaT bit corresponding to GR r_3 is set then the state of memory is not affected. In the reg_base_update_form and imm_base_update_form, the post increment of GR r_3 is performed and prefetch is hinted as described above.

`lfetch` instructions, like hardware prefetches, are not orderable operations, i.e. they have no order with respect to prior or subsequent memory operations.

Table 2-36. *lfhint* Mnemonic Values

lfhint Mnemonic	Interpretation
none	Temporal locality, level 1
nt1	No temporal locality, level 1
nt2	No temporal locality, level 2
nta	No temporal locality, all levels

A faulting `lfetch` to an unimplemented address results in an Unimplemented Data Address fault. A non-faulting `lfetch` to an unimplemented address does not take the fault and will not issue a prefetch request, but, if specified, will perform a register post-increment.

Operation:

```
if (PR[qp]) {
    itype = READ|NON_ACCESS;
    itype |= (lftype == 'fault') ? LFETCH_FAULT : LFETCH;

    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);

    if (lftype == 'fault') {           // faulting form
        if (GR[r3].nat && !PSR.ed)     // fault on NaT address
            register_nat_consumption_fault(itype);
    }

    excl_hint = (exclusive_form) ? EXCLUSIVE : 0;

    if (!GR[r3].nat && !PSR.ed) {// faulting form already faulted if r3 is nat
        paddr = tlb_translate(GR[r3], 1, itype, PSR.cpl, &mattr, &defer);
        if (!defer)
            mem_promote(paddr, mattr, lfhint | excl_hint);
    }

    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = GR[r3].nat;
    } else if (reg_base_update_form) {
        GR[r3] = GR[r3] + GR[r2];
        GR[r3].nat = GR[r2].nat || GR[r3].nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
        mem_implicit_prefetch(GR[r3], lfhint | excl_hint, itype);
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Access Bit fault
Data Debug fault

Load Register Stack

Format: loadrs M25

Description: This instruction ensures that a specified number of bytes (registers values and/or NaT collections) below the current BSP have been loaded from the backing store into the stacked general registers. The loaded registers are placed into the dirty partition of the register stack. All other stacked general registers are marked as invalid, without being saved to the backing store.

The number of bytes to be loaded is specified in a sub-field of the RSC application register (RSC.loadrs). Backing store addresses are always 8-byte aligned, and therefore the low order 3 bits of the `loadrs` field (RSC.loadrs{2:0}) are ignored. This instruction can be used to invalidate all stacked registers outside the current frame, by setting RSC.loadrs to zero.

This instruction will fault with an Illegal Operation fault under any of the following conditions:

- The RSE is not in enforced lazy mode (RSC.mode is non-zero).
- CFM.sof and RSC.loadrs are both non-zero.
- An attempt is made to load up more registers than are available in the physical stacked register file.

This instruction must be the first instruction in an instruction group. Otherwise, the results are undefined. This instruction cannot be predicated.

Operation:

```
if (AR[RSC].mode != 0)
    illegal_operation_fault();

if ((CFM.sof != 0) && (AR[RSC].loadrs != 0))
    illegal_operation_fault();

rse_ensure_regs_loaded(AR[RSC].loadrs);  // can raise faults listed below
AR[RNAT] = undefined();
```

Interruptions:

Illegal Operation fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Access Bit fault
Data Debug fault

Memory Fence

Format:

(*qp*) mf	ordering_form	M24
(*qp*) mf.a	acceptance_form	M24

Description: This instruction forces ordering between prior and subsequent memory accesses. The ordering_form ensures all prior data memory accesses are made visible prior to any subsequent data memory accesses being made visible. It does not ensure prior data memory references have been accepted by the external platform, nor that prior data memory references are visible.

The acceptance_form prevents any subsequent data memory accesses by the processor from initiating transactions to the external platform until:

- All prior loads to sequential pages have returned data, and
- All prior stores to sequential pages have been accepted by the external platform.

The definition of “acceptance” is platform dependent. The acceptance_form is typically used to ensure the processor has “waited” until a memory-mapped I/O transaction has been “accepted”, before initiating additional external transactions. The acceptance_form does not ensure ordering, or acceptance to memory areas other than sequential pages.

Operation:

```
if (PR[qp]){
    if (acceptance_form)
        acceptance_fence();
    else // ordering_form
        ordering_fence();
}
```

Interruptions: None

Mix

Format:

(*qp*) mix1.l $r_1 = r_2, r_3$	one_byte_form, left_form	I2
(*qp*) mix2.l $r_1 = r_2, r_3$	two_byte_form, left_form	I2
(*qp*) mix4.l $r_1 = r_2, r_3$	four_byte_form, left_form	I2
(*qp*) mix1.r $r_1 = r_2, r_3$	one_byte_form, right_form	I2
(*qp*) mix2.r $r_1 = r_2, r_3$	two_byte_form, right_form	I2
(*qp*) mix4.r $r_1 = r_2, r_3$	four_byte_form, right_form	I2

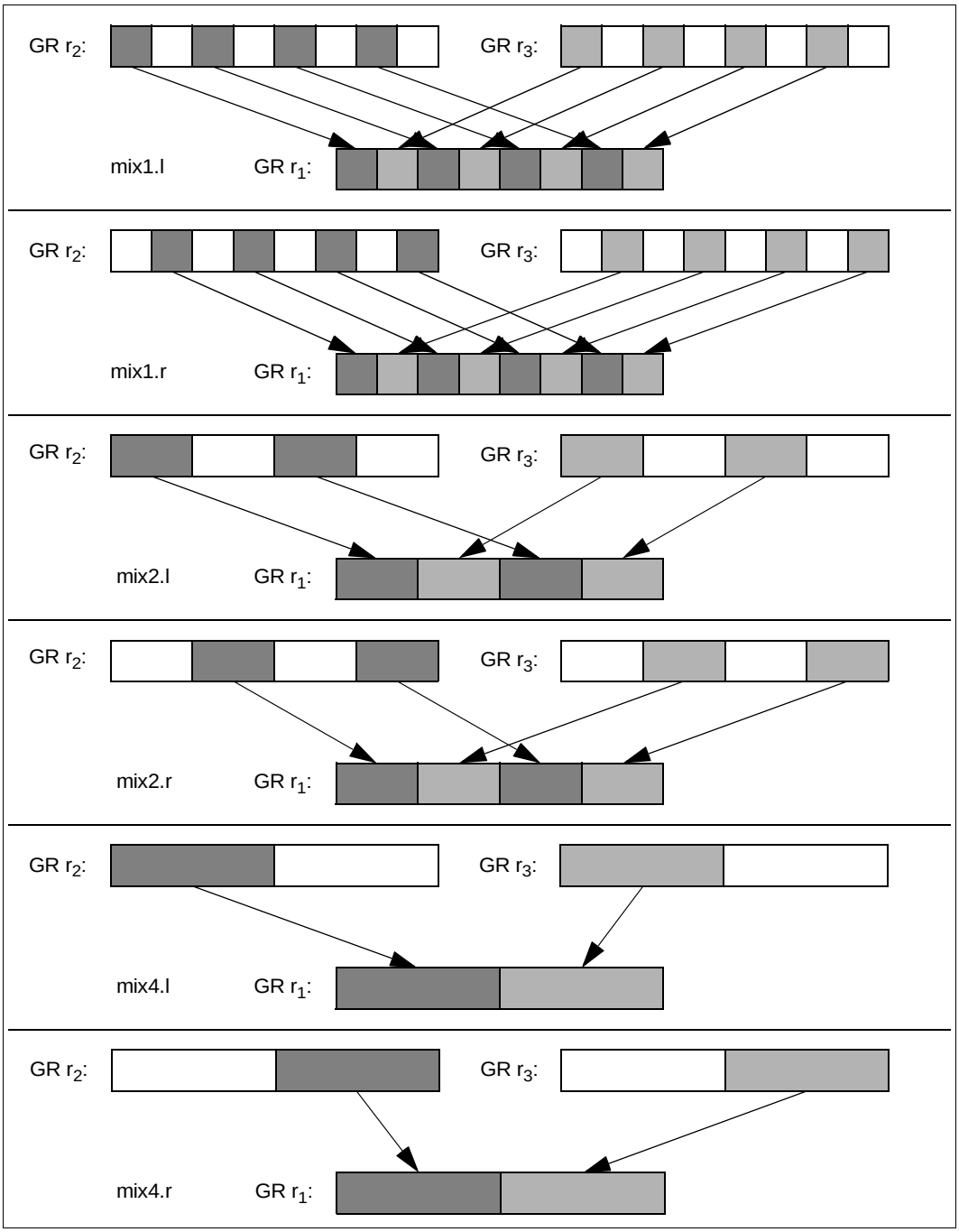
Description: The data elements of GR r_2 and r_3 are mixed as shown in Figure 2-23, and the result placed in GR r_1. The data elements in the source registers are grouped in pairs, and one element from each pair is selected for the result. In the left_form, the result is formed from the leftmost elements from each of the pairs. In the right_form, the result is formed from the rightmost elements. Elements are selected alternately from the two source registers.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                  // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (left_form)
            GR[r1] = concatenate8(x[7], y[7], x[5], y[5],
                                  x[3], y[3], x[1], y[1]);
        else // right_form
            GR[r1] = concatenate8(x[6], y[6], x[4], y[4],
                                  x[2], y[2], x[0], y[0]);

    } else if (two_byte_form) {                           // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (left_form)
            GR[r1] = concatenate4(x[3], y[3], x[1], y[1]);
        else // right_form
            GR[r1] = concatenate4(x[2], y[2], x[0], y[0]);

    } else {                                              // four-byte elements
        x[0] = GR[r2]{31:0};     y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};    y[1] = GR[r3]{63:32};

        if (left_form)
            GR[r1] = concatenate2(x[1], y[1]);
        else // right_form
            GR[r1] = concatenate2(x[0], y[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Figure 2-23. Mix Example



Move Application Register

Format:

(*qp*) mov $r_1 = ar_3$	pseudo-op	
(*qp*) mov $ar_3 = r_2$	pseudo-op	
(*qp*) mov $ar_3 = imm_8$	pseudo-op	
(*qp*) mov.i $r_1 = ar_3$	i_form, from_form	I28
(*qp*) mov.i $ar_3 = r_2$	i_form, register_form, to_form	I26
(*qp*) mov.i $ar_3 = imm_8$	i_form, immediate_form, to_form	I27
(*qp*) mov.m $r_1 = ar_3$	m_form, from_form	M31
(*qp*) mov.m $ar_3 = r_2$	m_form, register_form, to_form	M29
(*qp*) mov.m $ar_3 = imm_8$	m_form, immediate_form, to_form	M30

Description: The source operand is copied to the destination register.

In the from_form, the application register specified by ar_3 is copied into GR r_1 and the corresponding NaT bit is cleared.

In the to_form, the value in GR r_2 (in the register_form), or the sign-extended value in imm_8 (in the immediate_form), is placed in AR ar_3. In the register_form if the NaT bit corresponding to GR r_2 is set, then a Register NaT Consumption fault is raised.

Only a subset of the application registers can be accessed by each execution unit (M or I). Table 3-3 on page 3-6 in Volume 1 indicates which application registers may be accessed from which execution unit type. An access to an application register from the wrong unit type causes an Illegal Operation fault.

This instruction has multiple forms with the pseudo operation eliminating the need for specifying the execution unit. Accesses of the ARs are always implicitly serialized. While implicitly serialized, read-after-write and write-after-write dependency violations must be avoided (e.g. setting CCV, followed by `cmpxchg` in the same instruction group, or simultaneous writes to the UNAT register by `ld.fill` and mov to UNAT).

Operation:

```
if (PR[qp]) {
    tmp_type = (i_form ? AR_I_TYPE : AR_M_TYPE);
    if (is_reserved_reg(tmp_type, ar3))
        illegal_operation_fault();

    if (from_form) {
        check_target_register(r1);
        if (((ar3 == BSPSTORE) || (ar3 == RNAT)) && (AR[RSC].mode != 0))
            illegal_operation_fault();

        if (ar3 == ITC && PSR.si && PSR.cpl != 0)
            privileged_register_fault();

        GR[r1] = (is_ignored_reg(ar3)) ? 0 : AR[ar3];
        GR[r1].nat = 0;
    } else {                                          // to_form
        tmp_val = (register_form) ? GR[r2] : sign_ext(imm8, 8);

        if (is_read_only_register(AR_TYPE, ar3) ||
            (((ar3 == BSPSTORE) || (ar3 == RNAT)) && (AR[RSC].mode != 0)))
            illegal_operation_fault();

        if (register_form && GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(AR_TYPE, ar3, tmp_val))
```

```
            reserved_register_field_fault();

        if ((is_kernel_reg(ar3) || ar3 == ITC) && (PSR.cpl != 0))
            privileged_register_fault();

        if (!is_ignored_reg(ar3)) {
            tmp_val = ignored_field_mask(AR_TYPE, ar3, tmp_val);
            // check for illegal promotion
            if (ar3 == RSC && tmp_val{3:2} u< PSR.cpl)
                tmp_val{3:2} = PSR.cpl;
            AR[ar3] = tmp_val;

            if (ar3 == BSPSTORE) {
                AR[BSP] = rse_update_internal_stack_pointers(tmp_val);
                AR[RNAT] = undefined();
            }
        }
    }
}
```

Interruptions: Illegal Operation fault
Reserved Register/Field fault
Register NaT Consumption fault
Privileged Register fault

Move Branch Register

Format:

(*qp*) mov $r_1 = b_2$	from_form	I22
(*qp*) mov $b_1 = r_2$	pseudo-op	
(*qp*) mov.*mwh.ih* $b_1 = r_2, tag_{13}$	to_form	I21
(*qp*) mov.ret.*mwh.ih* $b_1 = r_2, tag_{13}$	return_form, to_form	I21

Description: The source operand is copied to the destination register.

In the from_form, the branch register specified by b_2 is copied into GR r_1. The NaT bit corresponding to GR r_1 is cleared.

In the to_form, the value in GR r_2 is copied into BR b_1. If the NaT bit corresponding to GR r_2 is 1, then a Register NaT Consumption fault is taken.

A set of hints can also be provided when moving to a branch register. These hints are very similar to those provided on the `brp` instruction, and provide prediction information about a future branch which may use the value being moved into BR b_1. The return_form is used to provide the hint that this value will be used in a return-type branch.

The values for the *mwh* whether hint completer are given in Table 2-37. For a description of the *ih* hint completer see the Branch Prediction instruction and Table 2-12 on page 2-20.

Table 2-37. Move to BR Whether Hints

mwh Completer	Move to BR Whether Hint
none	Ignore all hints
sptk	Static Taken
dptk	Dynamic

A pseudo-op is provided for copying a general register into a branch register when there is no hint information to be specified. This is encoded with a value of 0 for tag_{13} and values corresponding to *none* for the hint completers.

Operation:

```
if (PR[qp]) {
    if (from_form) {
        check_target_register(r1);
        GR[r1] = BR[b2];
        GR[r1].nat = 0;
    } else { // to_form
        tmp_tag = IP + sign_ext((timm9 << 4), 13);
        if (GR[r2].nat)
            register_nat_consumption_fault(0);
        BR[b1] = GR[r2];
        branch_predict(mwh, ih, return_form, GR[r2], tmp_tag);
    }
}
```

Interruptions: Illegal Operation fault Register NaT Consumption fault

Move Control Register

Format:

(*qp*) mov $r_1 = cr_3$ from_form M33
(*qp*) mov $cr_3 = r_2$ to_form M32

Description: The source operand is copied to the destination register.

For the from_form, the control register specified by cr_3 is read and the value copied into GR r_1.

For the to_form, GR r_2 is read and the value copied into CR cr_3.

Control registers can only be accessed at the most privileged level. Reading or writing an interruption control register (CR16-CR25), when the PSR.ic bit is one, will result in an Illegal Operation fault.

Operation:

```
if (PR[qp]) {
    if (is_reserved_reg(CR_TYPE, cr3)
        || to_form && is_read_only_reg(CR_TYPE, cr3)
        || PSR.ic && is_interruption_cr(cr3))
    {
        illegal_operation_fault();
    }

    if (from_form)
        check_target_register(r1);
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (from_form) {
        if (cr3 == IVR)
            check_interrupt_request();

        GR[r1] = CR[cr3];
        GR[r1].nat = 0;
    } else {              // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(CR_TYPE, cr3, GR[r2]))
            reserved_register_field_fault();
        if (cr3 == EOI)
            end_of_interrupt();

        tmp_val = ignored_field_mask(CR_TYPE, cr3, GR[r2]);
        CR[cr3] = tmp_val;
        if (cr3 == IIPA)
            last_IP = tmp_val;
    }
}
```

Interruptions:

Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault

Serialization: Reads of control registers reflect the results of all prior instruction groups and interruptions.

In general, writes to control registers do not immediately affect subsequent instructions. Software must issue a serialize operation before a dependent instruction uses a modified resource.

Control register writes are not implicitly synchronized with a corresponding control register read and requires data serialization.

Move Floating-Point Register

Format: (*qp*) mov $f_1 = f_3$ pseudo-op of: (*qp*) fmerge.s $f_1 = f_3, f_3$

Description: The value of FR f_3 is copied to FR f_1.

Operation: See "Floating-Point Merge" on p. 2-63.

Move General Register

Format: (*qp*) mov $r_1 = r_3$ pseudo-op of: (*qp*) adds $r_1 = 0, r_3$

Description: The value of GR r_3 is copied to GR r_1.

Operation: See "Add" on p. 2-3.

Move Immediate

Format: (*qp*) mov $r_1 = imm_{22}$ pseudo-op of: (*qp*) addl $r_1 = imm_{22}$, r0

Description: The immediate value, imm_{22}, is sign extended to 64 bits and placed in GR r_1.

Operation: See "Add" on p. 2-3.

Move Indirect Register

Format:

(*qp*) mov $r_1 = ireg[r_3]$	from_form	M43
(*qp*) mov $ireg[r_3] = r_2$	to_form	M42

Description: The source operand is copied to the destination register.

For move from indirect register, GR r_3 is read and the value used as an index into the register file specified by *ireg* (see Table 2-38 below). The indexed register is read and its value is copied into GR r_1.

For move to indirect register, GR r_3 is read and the value used as an index into the register file specified by *ireg*. GR r_2 is read and its value copied into the indexed register.

Table 2-38. Indirect Register File Mnemonics

ireg	Register File
cpuid	Processor Identification Register
dbr	Data Breakpoint Register
ibr	Instruction Breakpoint Register
pkr	Protection Key Register
pmc	Performance Monitor Configuration Register
pmd	Performance Monitor Data Register
rr	Region Register

For all register files other than the region registers, bits {7:0} of GR r_3 are used as the index. For region registers, bits {63:61} are used. The remainder of the bits are ignored.

Instruction and data breakpoint, performance monitor configuration, protection key, and region registers can only be accessed at the most privileged level. Performance monitor data registers can only be written at the most privileged level.

The CPU identification registers can only be read. There is no to_form of this instruction.

For move to protection key register, the processor ensures uniqueness of protection keys by checking new valid protection keys against all protection key registers. If any matching keys are found, duplicate protection keys are invalidated.

Apart from the PMC and PMD register files, access of a non-existent register results in a Reserved Register/Field fault. All accesses to the implementation-dependent portion of PMC and PMD register files result in implementation dependent behavior but do not fault.

Modifying a region register or a protection key register which is being used to translate:

- The executing instruction stream when PSR.it == 1, or
- The data space for an eager RSE reference when PSR.rt == 1

is an undefined operation.

Operation:

```
if (PR[qp]) {
    if (ireg == RR_TYPE)
        tmp_index = GR[r3]{63:61};
    else // all other register types
        tmp_index = GR[r3]{7:0};

    if (from_form) {
        check_target_register(r1);

        if (PSR.cpl != 0 && !(ireg == PMD_TYPE || ireg == CPUID_TYPE))
            privileged_operation_fault(0);

        if (GR[r3].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_reg(ireg, tmp_index))
            reserved_register_field_fault();

        if (ireg == PMD_TYPE) {
            if ((PSR.cpl != 0) && ((PSR.sp == 1) ||
                 (tmp_index > 3 &&
                 tmp_index <= IMPL_MAXGENERIC_PMCPMD &&
                 PMC[tmp_index].pm == 1)))
                GR[r1] = 0;
            else
                GR[r1] = pmd_read(tmp_index);
        } else
            switch (ireg) {
                case CPUID_TYPE: GR[r1] = CPUID[tmp_index]; break;
                case DBR_TYPE:   GR[r1] = DBR[tmp_index]; break;
                case IBR_TYPE:   GR[r1] = IBR[tmp_index]; break;
                case PKR_TYPE:   GR[r1] = PKR[tmp_index]; break;
                case PMC_TYPE:   GR[r1] = pmc_read(tmp_index); break;
                case RR_TYPE:    GR[r1] = RR[tmp_index]; break;
            }
        GR[r1].nat = 0;
    } else {         // to_form
        if (PSR.cpl != 0)
            privileged_operation_fault(0);

        if (GR[r2].nat || GR[r3].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_reg(ireg, tmp_index)
            || is_reserved_field(ireg, tmp_index, GR[r2]))
            reserved_register_field_fault();
        if (ireg == PKR_TYPE && GR[r2]{0} == 1) { // writing valid prot key
            if ((tmp_slot = tlb_search_pkr(GR[r2]{31:8})) != NOT_FOUND)
                PKR[tmp_slot].v = 0; // clear valid bit of matching key reg
        }
        tmp_val = ignored_field_mask(ireg, tmp_index, GR[r2]);
        switch (ireg) {
               case DBR_TYPE:   DBR[tmp_index] = tmp_val; break;
               case IBR_TYPE:   IBR[tmp_index] = tmp_val; break;
               case PKR_TYPE:   PKR[tmp_index] = tmp_val; break;
               case PMC_TYPE:   pmc_write(tmp_index, tmp_val); break;
               case PMD_TYPE:   pmd_write(tmp_index, tmp_val); break;
               case RR_TYPE:    RR[tmp_index]= tmp_val; break;
            }
    }
}
```

Interruptions: Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault

Serialization: For move to data breakpoint registers, software must issue a data serialize operation before issuing a memory reference dependent on the modified register.

For move to instruction breakpoint registers, software must issue an instruction serialize operation before fetching an instruction dependent on the modified register.

For move to protection key, region, performance monitor configuration, and performance monitor data registers, software must issue an instruction or data serialize operation to ensure the changes are observed before issuing any dependent instruction.

To obtain improved accuracy, software can issue an instruction or data serialize operation before reading the performance monitors.

Move Instruction Pointer

Format: (*qp*) mov r_1 = ip I25

Description: The Instruction Pointer (IP) for the bundle containing this instruction is copied into GR r_1.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = IP;
    GR[r1].nat = 0;
}
```

Interruptions: Illegal Operation fault

Move Predicates

Format:

(*qp*) mov r_1 = pr	from_form	I25
(*qp*) mov pr = r_2, $mask_{17}$	to_form	I23
(*qp*) mov pr.rot = imm_{44}	to_rotate_form	I24

Description: The source operand is copied to the destination register.

For moving the predicates to a GR, PR i is copied to bit position i within GR r_1.

For moving to the predicates, the source can either be a general register, or an immediate value. In the to_form, the source operand is GR r_2 and only those predicates specified by the immediate value $mask_{17}$ are written. The value $mask_{17}$ is encoded in the instruction in an imm_{16} field such that: $imm_{16} = mask_{17} >> 1$. Predicate register 0 is always one. The $mask_{17}$ value is sign extended. The most significant bit of $mask_{17}$, therefore, is the mask bit for all of the rotating predicates. If there is a deferred exception for GR r_2 (the NaT bit is 1), a Register NaT Consumption fault is taken.

In the to_rotate_form, only the 48 rotating predicates can be written. The source operand is taken from the imm_{44} operand (which is encoded in the instruction in an imm_{28} field, such that: $imm_{28} = imm_{44} >> 16$). The low 16-bits correspond to the static predicates. The immediate is sign extended to set the top 21 predicates. Bit position i in the source operand is copied to PR i.

This instruction operates as if the predicate rotation base in the Current Frame Marker (CFM.rrb.pr) were zero.

Operation:

```
if (PR[qp]) {
    if (from_form) {
        check_target_register(r1);
        GR[r1] = 1;                              // PR[0] is always 1
        for (i = 1; i <= 63; i++) {
            GR[r1]{i} = PR[pr_phys_to_virt(i)];
        }
        GR[r1].nat = 0;
    } else if (to_form) {
        if (GR[r2].nat)
            register_nat_consumption_fault(0);
        tmp_src = sign_ext(mask17, 17);
        for (i = 1; i <= 63; i++) {
            if (tmp_src{i})
                PR[pr_phys_to_virt(i)] = GR[r2]{i};
        }
    } else {  // to_rotate_form
        tmp_src = sign_ext(imm44, 44);
        for (i = 16; i <= 63; i++) {
            PR[pr_phys_to_virt(i)] = tmp_src{i};
        }
    }
}
```

Interruptions: Illegal Operation fault Register NaT Consumption fault

Move Processor Status Register

Format:

(*qp*) mov r_1 = psr — from_form M36

(*qp*) mov psr.l = r_2 — to_form M35

Description: The source operand is copied to the destination register.

For move from processor status register, PSR bits {36:35} and {31:0} are read, and copied into GR r_1. All other bits of the PSR read as zero.

For move to processor status register, GR r_2 is read, bits {31:0} copied into PSR{31:0} and bits {45:32} are ignored. All bits of GR r_2 corresponding to reserved fields of the PSR must be 0 or a Reserved Register/Field fault will result.

Moves to and from the PSR can only be performed at the most privileged level.

The contents of the interruption resources (that are overwritten when the PSR.ic bit is 1) are undefined if an interruption occurs between the enabling of the PSR.ic bit and a subsequent instruction serialize operation.

Operation:

```
if (PR[qp]) {
    if (from_form)
        check_target_register(r1);
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (from_form) {
        tmp_val = zero_ext(PSR{31:0}, 32);     // read lower 32 bits
        tmp_val |= PSR{36:35} << 35;           // read mc and it bits
        GR[r1] = tmp_val;                      // other bits read as zero
        GR[r1].nat = 0;
    } else {          // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(PSR_TYPE, PSR_MOVPART, GR[r2]))
            reserved_register_field_fault();

        PSR{31:0} = GR[r2]{31:0};
    }
}
```

Interruptions:

Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault

Serialization: Software must issue an instruction or data serialize operation before issuing instructions dependent upon the altered PSR bits. Unlike with the `rsm` instruction, the PSR.i bit is not treated specially when cleared.

Move User Mask

Format:

(*qp*) mov r_1 = psr.um — from_form M36
(*qp*) mov psr.um = r_2 — to_form M35

Description: The source operand is copied to the destination register.

For move from user mask, PSR{5:0} is read, zero-extend, and copied into GR r_1.

For move to user mask, PSR{5:0} is written by bits {5:0} of GR r_2. PSR.up can only be modified if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Writing a non-zero value into any other parts of the PSR results in a Reserved Register/Field fault.

Operation:

```
if (PR[qp]) {
    if (from_form) {
        check_target_register(r1);

        GR[r1] = zero_ext(PSR{5:0}, 6);
        GR[r1].nat = 0;
    } else {                                          // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(PSR_TYPE, PSR_UM, GR[r2]))
            reserved_register_field_fault();

        PSR{1:0} = GR[r2]{1:0};

        if (PSR.sp == 0)            // unsecured perf monitor
            PSR{2} = GR[r2]{2};

        PSR{5:3} = GR[r2]{5:3};
    }
}
```

Interruptions: Illegal Operation fault — Reserved Register/Field fault
Register NaT Consumption fault

Serialization: All user mask modifications are observed by the next instruction group.

Move Long Immediate

Format: (*qp*) movl $r_1 = imm_{64}$ X2

Description: The immediate value imm_{64} is copied to GR r_1. The L slot of the bundle contains 41 bits of imm_{64}.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = imm64;
    GR[r1].nat = 0;
}
```

Interruptions: Illegal Operation fault

Mux

Format:

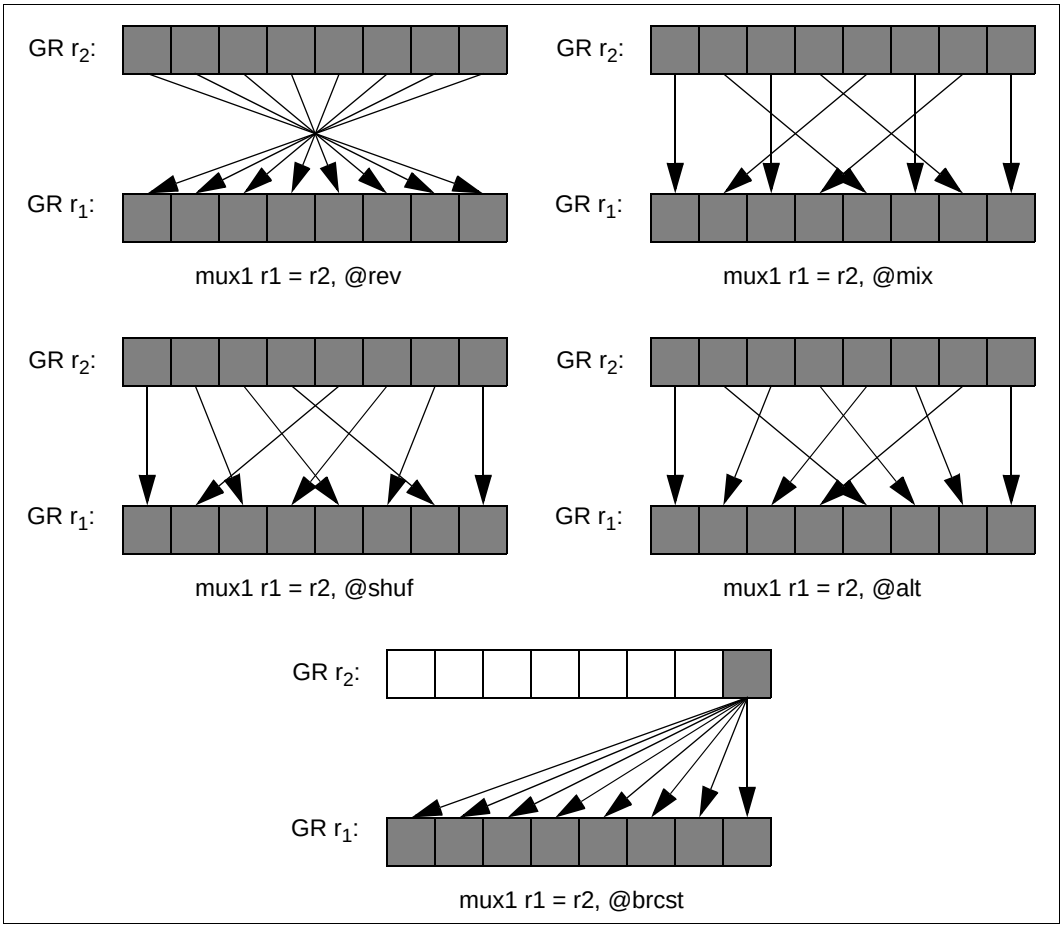
(*qp*) mux1 $r_1 = r_2$, $mbtype_4$	one_byte_form	I3
(*qp*) mux2 $r_1 = r_2$, $mhtype_8$	two_byte_form	I4

Description: A permutation is performed on the packed elements in a single source register, GR r_2, and the result is placed in GR r_1. For 8-bit elements, only some of all possible permutations can be specified. The five possible permutations are given in Table 2-39 and shown in Figure 2-24.

Table 2-39. Mux Permutations for 8-bit Elements

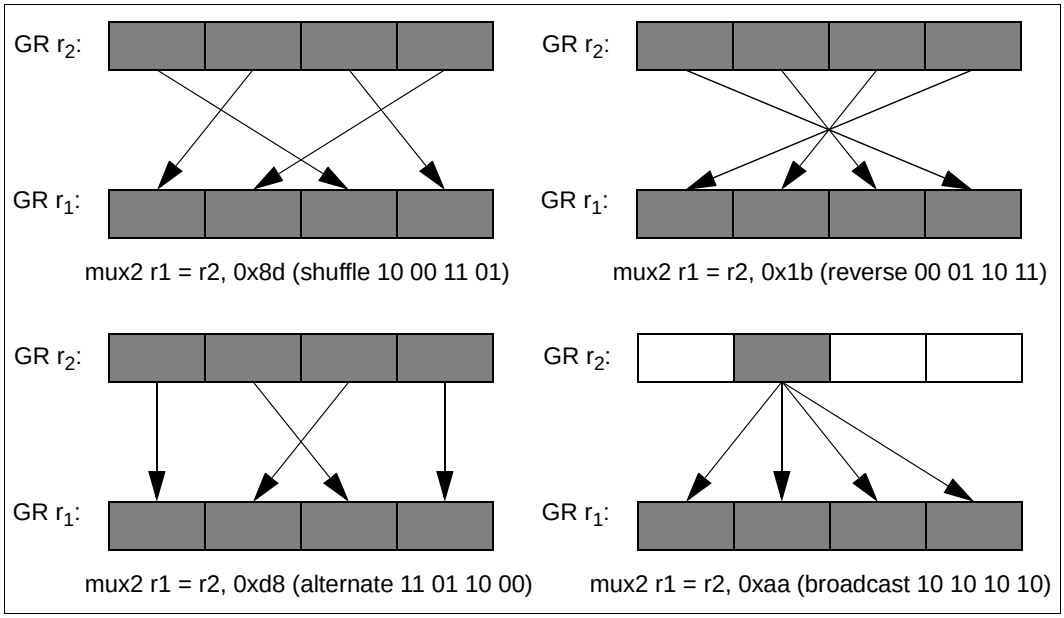
$mbtype_4$	Function
@rev	Reverse the order of the bytes
@mix	Perform a Mix operation on the two halves of GR r_2
@shuf	Perform a Shuffle operation on the two halves of GR r_2
@alt	Perform an Alternate operation on the two halves of GR r_2
@brcst	Perform a Broadcast operation on the least significand byte of GR r_2

Figure 2-24. Mux1 Operation (8-bit elements)



For 16-bit elements, all possible permutations, with and without repetitions can be specified. They are expressed with an 8-bit $mhtype_8$ field, which encodes the indices of the four 16-bit data elements. The indexed 16-bit elements of GR r_2 are copied to corresponding 16-bit positions in the target register GR r_1. The indices are encoded in little-endian order. (The 8 bits of $mhtype_8$[7:0] are grouped in pairs of bits and named $mhtype_8$[3], $mhtype_8$[2], $mhtype_8$[1], $mhtype_8$[0] in the Operation section).

Figure 2-25. Mux2 Examples (16-bit elements)



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {
        x[0] = GR[r2]{7:0};
        x[1] = GR[r2]{15:8};
        x[2] = GR[r2]{23:16};
        x[3] = GR[r2]{31:24};
        x[4] = GR[r2]{39:32};
        x[5] = GR[r2]{47:40};
        x[6] = GR[r2]{55:48};
        x[7] = GR[r2]{63:56};

        switch (mbtype) {
            case '@rev':
                GR[r1] = concatenate8(x[0], x[1], x[2], x[3],
                                      x[4], x[5], x[6], x[7]);
                break;

            case '@mix':
                GR[r1] = concatenate8(x[7], x[3], x[5], x[1],
                                      x[6], x[2], x[4], x[0]);
                break;

            case '@shuf':
                GR[r1] = concatenate8(x[7], x[3], x[6], x[2],
                                      x[5], x[1], x[4], x[0]);
                break;
```

```
            case '@alt':
                GR[r1] = concatenate8(x[7], x[5], x[3], x[1],
                                      x[6], x[4], x[2], x[0]);
                break;

            case '@brcst':
                GR[r1] = concatenate8(x[0], x[0], x[0], x[0],
                                      x[0], x[0], x[0], x[0]);
                break;
        }
    } else {                                          // two_byte_form
        x[0] = GR[r2]{15:0};
        x[1] = GR[r2]{31:16};
        x[2] = GR[r2]{47:32};
        x[3] = GR[r2]{63:48};

        res[0] = x[mhtype8{1:0}];
        res[1] = x[mhtype8{3:2}];
        res[2] = x[mhtype8{5:4}];
        res[3] = x[mhtype8{7:6}];

        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat;
}
```

Interruptions: Illegal Operation fault

No Operation

Format:

(*qp*) nop *imm*$_{21}$	pseudo-op	
(*qp*) nop.i *imm*$_{21}$	i_unit_form	I19
(*qp*) nop.b *imm*$_{21}$	b_unit_form	B9
(*qp*) nop.m *imm*$_{21}$	m_unit_form	M37
(*qp*) nop.f *imm*$_{21}$	f_unit_form	F15
(*qp*) nop.x *imm*$_{62}$	x_unit_form	X1

Description: No operation is done.

The immediate, *imm*$_{21}$ or *imm*$_{62}$, can be used by software as a marker in program code. It is ignored by hardware.

For the x_unit_form, the L slot of the bundle contains the upper 41 bits of *imm*$_{62}$.

A `nop.i` instruction may be encoded in an MLI-template bundle, in which case the L slot of the bundle is ignored.

This instruction has five forms, each of which can be executed only on a particular execution unit type. The pseudo-op can be used if the unit type to execute on is unimportant.

Operation:

```
if (PR[qp]) {
    ; // no operation
}
```

Interruptions: None

Logical Or

Format:

(*qp*) or $r_1 = r_2, r_3$	register_form	A1
(*qp*) or $r_1 = imm_8, r_3$	imm8_form	A3

Description: The two source operands are logically ORed and the result placed in GR r_1. In the register form the first operand is GR r_2; in the immediate form the first operand is taken from the imm_8 encoding field.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src | GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Pack

Format:

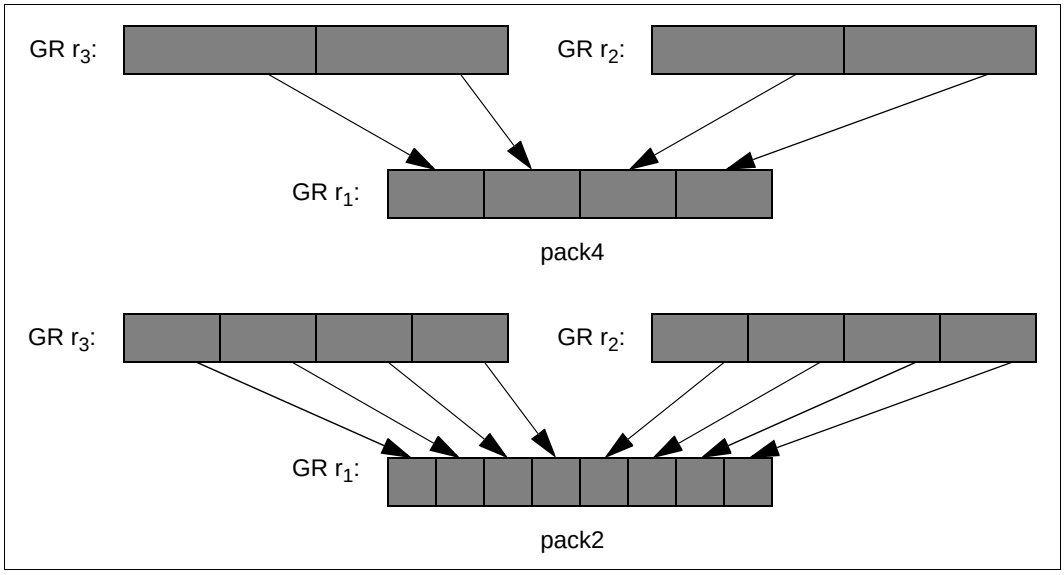
(*qp*) pack*2*.sss $r_1 = r_2, r_3$	two_byte_form, signed_saturation_form	I2
(*qp*) pack*2*.uss $r_1 = r_2, r_3$	two_byte_form, unsigned_saturation_form	I2
(*qp*) pack4.sss $r_1 = r_2, r_3$	four_byte_form, signed_saturation_form	I2

Description: 32-bit or 16-bit elements from GR r_2 and GR r_3 are converted into 16-bit or 8-bit elements respectively, and the results are placed GR r_1. The source elements are treated as signed values. If a source element cannot be represented in the result element, then saturation clipping is performed. The saturation can either be signed or unsigned. If an element is larger than the upper limit value, the result is the upper limit value. If it is smaller than the lower limit value, the result is the lower limit value. The saturation limits are given in Table 2-40.

Table 2-40. Pack Saturation Limits

Size	Source Element Width	Result Element Width	Saturation	Upper Limit	Lower Limit
2	16 bit	8 bit	signed	0x7f	0x80
2	16 bit	8 bit	unsigned	0xff	0x00
4	32 bit	16 bit	signed	0x7fff	0x8000

Figure 2-26. Pack Operation



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (two_byte_form) {
        if (signed_saturation_form) {
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);
        } else {                                    // unsigned_saturation_form
            max = 0xff;
            min = 0x00;
        }
        temp[0] = sign_ext(GR[r2]{15:0},  16);
        temp[1] = sign_ext(GR[r2]{31:16}, 16);
        temp[2] = sign_ext(GR[r2]{47:32}, 16);
        temp[3] = sign_ext(GR[r2]{63:48}, 16);
        temp[4] = sign_ext(GR[r3]{15:0},  16);
        temp[5] = sign_ext(GR[r3]{31:16}, 16);
        temp[6] = sign_ext(GR[r3]{47:32}, 16);
        temp[7] = sign_ext(GR[r3]{63:48}, 16);

        for (i = 0; i < 8; i++) {
            if (temp[i] > max)
                temp[i] = max;

            if (temp[i] < min)
                temp[i] = min;
        }

        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                              temp[3], temp[2], temp[1], temp[0]);

    } else {                                        // four_byte_form
        max = sign_ext(0x7fff, 16);                 // signed_saturation_form
        min = sign_ext(0x8000, 16);
        temp[0] = sign_ext(GR[r2]{31:0},  32);
        temp[1] = sign_ext(GR[r2]{63:32}, 32);
        temp[2] = sign_ext(GR[r3]{31:0},  32);
        temp[3] = sign_ext(GR[r3]{63:32}, 32);

        for (i = 0; i < 4; i++) {
            if (temp[i] > max)
                temp[i] = max;

            if (temp[i] < min)
                temp[i] = min;
        }

        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Parallel Add

Format:

(*qp*) padd1 $r_1 = r_2, r_3$	one_byte_form, modulo_form	A9
(*qp*) padd1.sss $r_1 = r_2, r_3$	one_byte_form, sss_saturation_form	A9
(*qp*) padd1.uus $r_1 = r_2, r_3$	one_byte_form, uus_saturation_form	A9
(*qp*) padd1.uuu $r_1 = r_2, r_3$	one_byte_form, uuu_saturation_form	A9
(*qp*) padd2 $r_1 = r_2, r_3$	two_byte_form, modulo_form	A9
(*qp*) padd2.sss $r_1 = r_2, r_3$	two_byte_form, sss_saturation_form	A9
(*qp*) padd2.uus $r_1 = r_2, r_3$	two_byte_form, uus_saturation_form	A9
(*qp*) padd2.uuu $r_1 = r_2, r_3$	two_byte_form, uuu_saturation_form	A9
(*qp*) padd4 $r_1 = r_2, r_3$	four_byte_form, modulo_form	A9

Description: The sets of elements from the two source operands are added, and the results placed in GR r_1.

If a sum of two elements cannot be represented in the result element and a saturation completer is specified, then saturation clipping is performed. The saturation can either be signed or unsigned, as given in Table 2-41. If the sum of two elements is larger than the upper limit value, the result is the upper limit value. If it is smaller than the lower limit value, the result is the lower limit value. The saturation limits are given in Table 2-42.

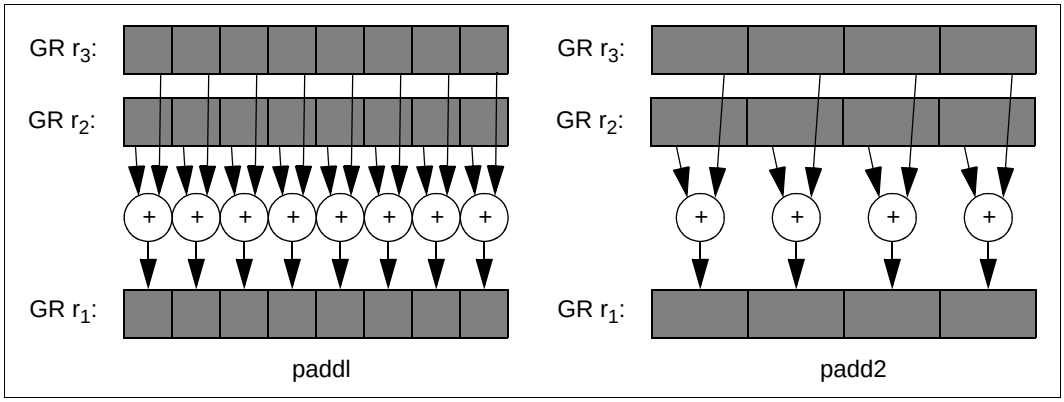
Table 2-41. Parallel Add Saturation Completers

Completer	Result r_1 Treated As	Source r_2 Treated As	Source r_3 Treated As
sss	signed	signed	signed
uus	unsigned	unsigned	signed
uuu	unsigned	unsigned	unsigned

Table 2-42. Parallel Add Saturation Limits

Size	Element Width	Result r_1 Signed		Result r_1 Unsigned	
		Upper Limit	Lower Limit	Upper Limit	Lower Limit
1	8 bit	0x7f	0x80	0xff	0x00
2	16 bit	0x7fff	0x8000	0xffff	0x0000

Figure 2-27. Parallel Add Examples



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                  // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (sss_saturation_form) {
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);

            for (i = 0; i < 8; i++) {
                temp[i] = sign_ext(x[i], 8) + sign_ext(y[i], 8);
            }
        } else if (uus_saturation_form) {
            max = 0xff;
            min = 0x00;

            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + sign_ext(y[i], 8);
            }
        } else if (uuu_saturation_form) {
            max = 0xff;
            min = 0x00;

            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
            }
        } else {                                          // modulo_form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
            }
        }

        if (sss_saturation_form || uus_saturation_form ||
            uuu_saturation_form) {
            for (i = 0; i < 8; i++) {
                if (temp[i] > max)
                    temp[i] = max;

                if (temp[i] < min)
                    temp[i] = min;
            }
        }
        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                              temp[3], temp[2], temp[1], temp[0]);

    } else if (two_byte_form) {                           // 2-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (sss_saturation_form) {
            max = sign_ext(0x7fff, 16);
            min = sign_ext(0x8000, 16);
```

```
            for (i = 0; i < 4; i++) {
                temp[i] = sign_ext(x[i], 16) + sign_ext(y[i], 16);
            }
        } else if (uus_saturation_form) {
            max = 0xffff;
            min = 0x0000;

            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + sign_ext(y[i], 16);
            }
        } else if (uuu_saturation_form) {
            max = 0xffff;
            min = 0x0000;

            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
            }
        } else {                                        // modulo_form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
            }
        }

        if (sss_saturation_form || uus_saturation_form ||
            uuu_saturation_form) {
            for (i = 0; i < 4; i++) {
                if (temp[i] > max)
                    temp[i] = max;

                if (temp[i] < min)
                    temp[i] = min;
            }
        }
        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);

    } else {                                            // four-byte elements
        x[0] = GR[r2]{31:0};    y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};   y[1] = GR[r3]{63:32};

        for (i = 0; i < 2; i++) {                       // modulo_form
            temp[i] = zero_ext(x[i], 32) + zero_ext(y[i], 32);
        }

        GR[r1] = concatenate2(temp[1], temp[0]);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Parallel Average

Format:	(*qp*) pavg1 $r_1 = r_2, r_3$	normal_form, one_byte_form	A9
	(*qp*) pavg1.raz $r_1 = r_2, r_3$	raz_form, one_byte_form	A9
	(*qp*) pavg2 $r_1 = r_2, r_3$	normal_form, two_byte_form	A9
	(*qp*) pavg2.raz $r_1 = r_2, r_3$	raz_form, two_byte_form	A9

Description: The unsigned data elements of GR r_2 are added to the unsigned data elements of GR r_3. The results of the add are then each independently shifted to the right by one bit position. The high-order bits of each element are filled with the carry bits of the sums. To prevent cumulative round-off errors, an averaging is performed. The unsigned results are placed in GR r_1.

The averaging operation works as follows. In the normal_form, the low-order bit of each result is set to 1 if at least one of the two least significant bits of the corresponding sum is 1. In the raz_form, the average rounds away from zero by adding 1 to each of the sums.

Figure 2-28. Parallel Average Example

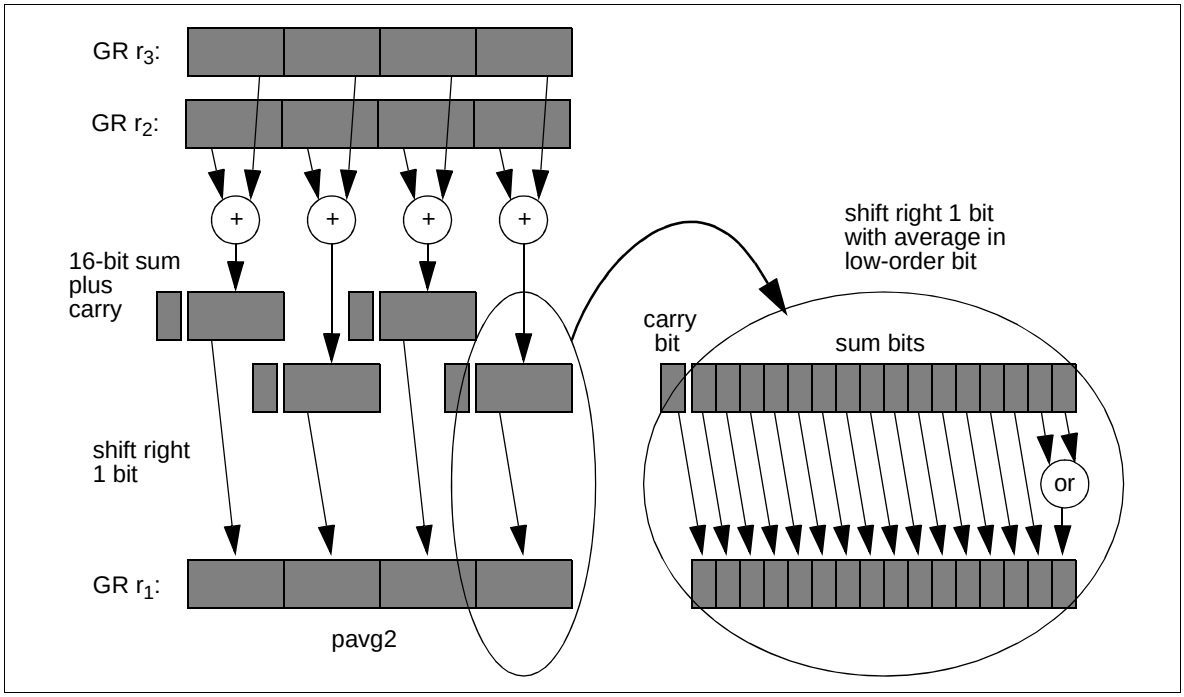
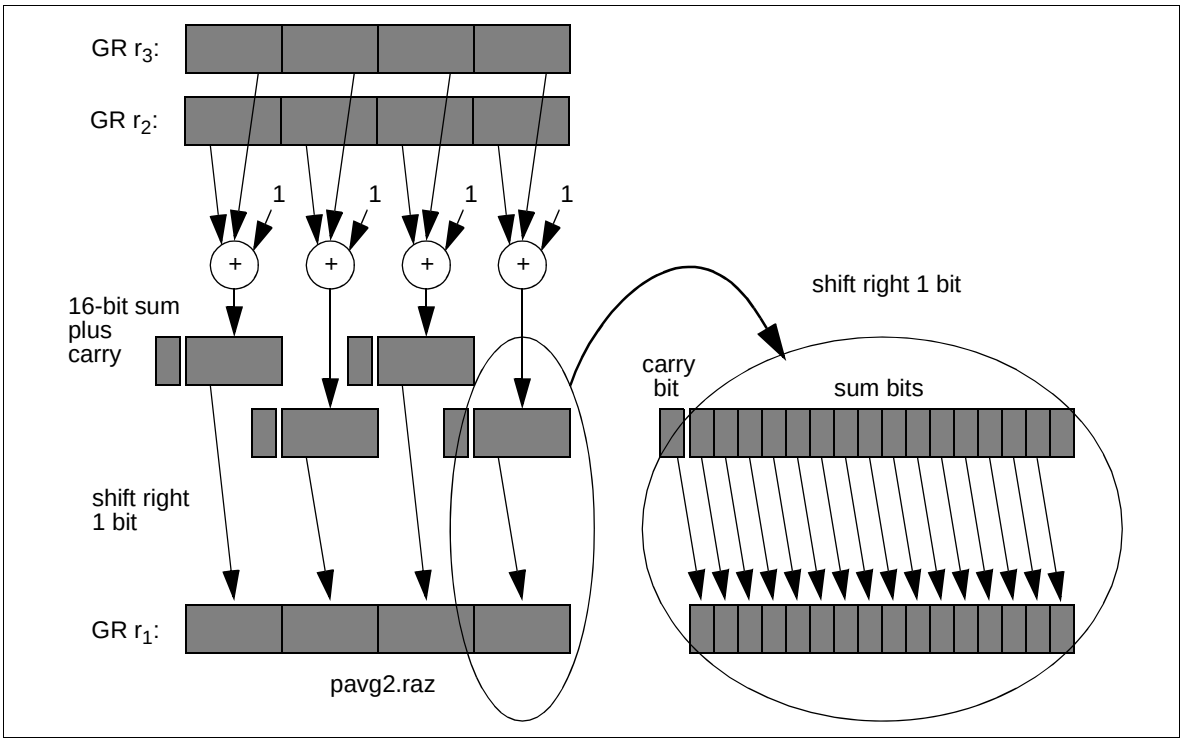


Figure 2-29. Parallel Average with Round Away from Zero Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (raz_form) {
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8) + 1;
                res[i] = shift_right_unsigned(temp[i], 1);
            }
        } else {                                        // normal form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
                res[i] = shift_right_unsigned(temp[i], 1) | (temp[i]{0});
            }
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);

    } else {                                            // two_byte_form
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
```

```
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (raz_form) {
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16) + 1;
                res[i] = shift_right_unsigned(temp[i], 1);
            }
        } else {                                        // normal form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
                res[i] = shift_right_unsigned(temp[i], 1) | (temp[i]{0});
            }
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

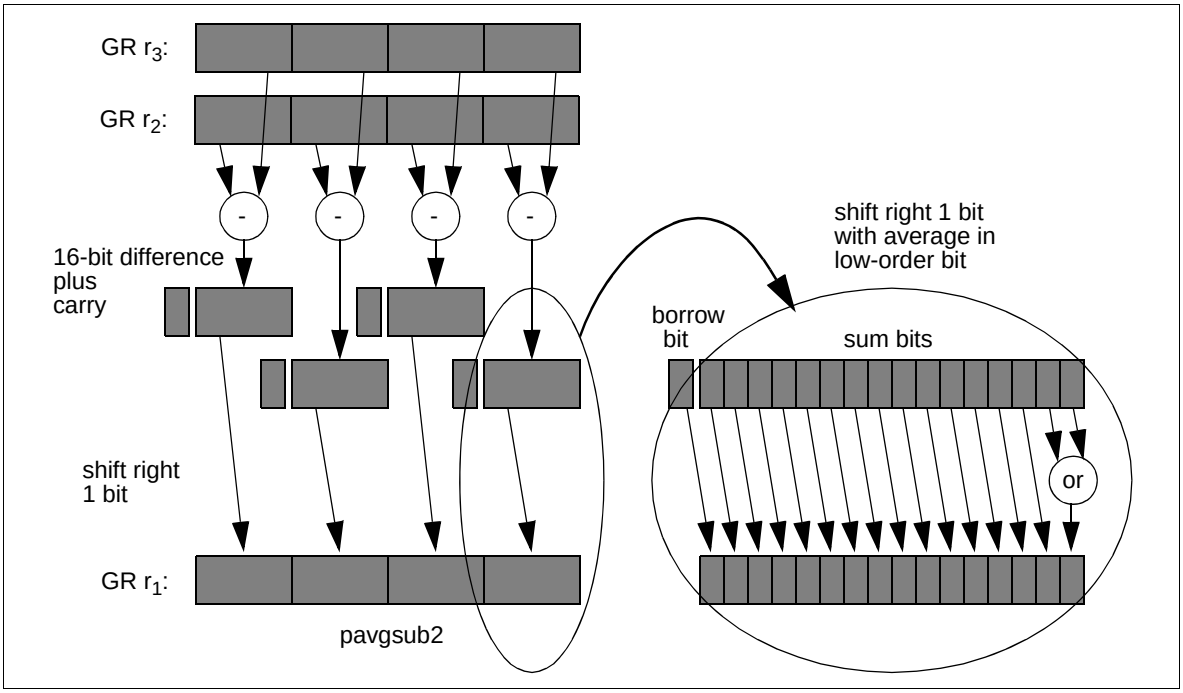
Parallel Average Subtract

Format:

(*qp*) pavgsub1 $r_1 = r_2, r_3$	one_byte_form	A9
(*qp*) pavgsub2 $r_1 = r_2, r_3$	two_byte_form	A9

Description: The unsigned data elements of GR r_3 are subtracted from the unsigned data elements of GR r_2. The results of the subtraction are then each independently shifted to the right by one bit position. The high-order bits of each element are filled with the borrow bits of the subtraction (the complements of the ALU carries). To prevent cumulative round-off errors, an averaging is performed. The low-order bit of each result is set to 1 if at least one of the two least significant bits of the corresponding difference is 1. The signed results are placed in GR r_1.

Figure 2-30. Parallel Average Subtract Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        for (i = 0; i < 8; i++) {
            temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            res[i] = (temp[i]{8:0} u>> 1) | (temp[i]{0});
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
```

```
    } else {                                              // two_byte_form
       x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
       x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
       x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
       x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

       for (i = 0; i < 4; i++) {
          temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
          res[i] = (temp[i]{16:0} u>> 1) | (temp[i]{0});
       }
       GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Parallel Compare

Format:

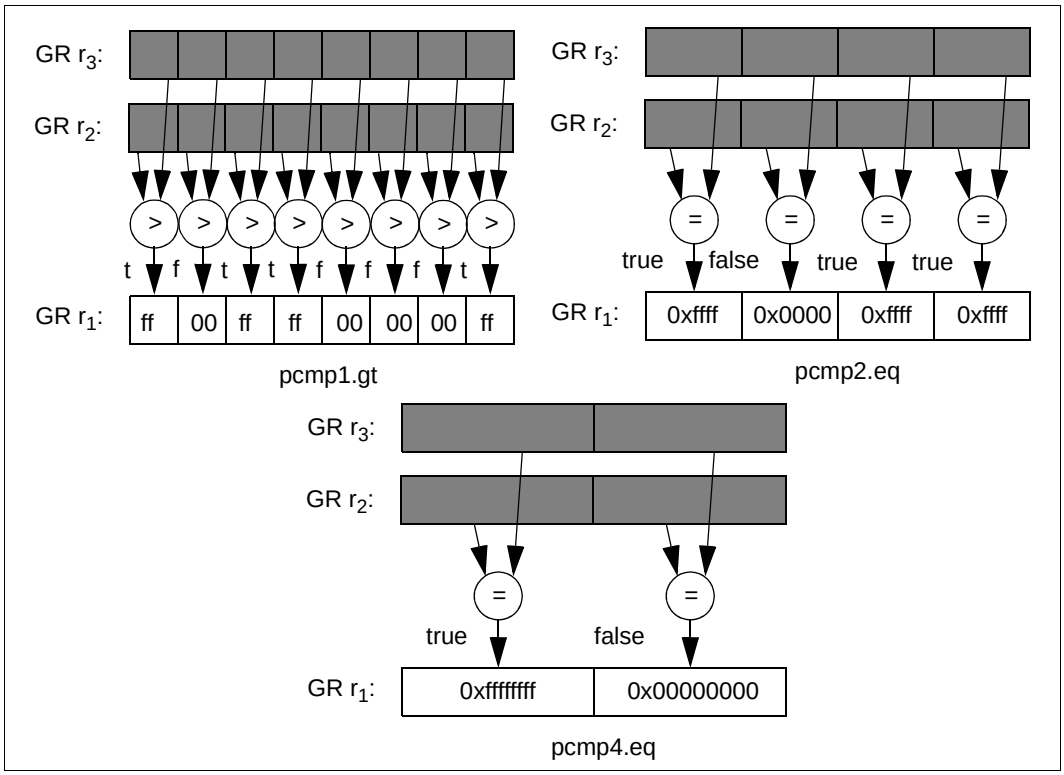
(*qp*) pcmp1.*prel* $r_1 = r_2, r_3$	one_byte_form	A9
(*qp*) pcmp2.*prel* $r_1 = r_2, r_3$	two_byte_form	A9
(*qp*) pcmp4.*prel* $r_1 = r_2, r_3$	four_byte_form	A9

Description: The two source operands are compared for one of the two relations shown in Table 2-43. If the comparison condition is true for corresponding data elements of GR r_2 and GR r_3, then the corresponding data element in GR r_1 is set to all ones. If the comparison condition is false, then the corresponding data element in GR r_1 is set to all zeros. For the '>' relation, both operands are interpreted as signed.

Table 2-43. Pcmp Relations

prel	Compare Relation (r_2 ***prel*** r_3)
eq	$r_2 == r_3$
gt	$r_2 > r_3$ (signed)

Figure 2-31. Parallel Compare Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                              // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
```

```
        x[5] = GR[r2]{47:40};   y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};   y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};   y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else // 'gt'
                tmp_rel = greater_signed(sign_ext(x[i], 8),
                                         sign_ext(y[i], 8));

            if (tmp_rel)
                res[i] = 0xff;
            else
                res[i] = 0x00;
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
    } else if (two_byte_form) {                     // two-byte elements
        x[0] = GR[r2]{15:0};    y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};   y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};   y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};   y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else // 'gt'
                tmp_rel = greater_signed(sign_ext(x[i], 16),
                                         sign_ext(y[i], 16));

            if (tmp_rel)
                res[i] = 0xffff;
            else
                res[i] = 0x0000;
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    } else {                                        // four-byte elements
        x[0] = GR[r2]{31:0};    y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};   y[1] = GR[r3]{63:32};
        for (i = 0; i < 2; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else // 'gt'
                tmp_rel = greater_signed(sign_ext(x[i], 32),
                                         sign_ext(y[i], 32));

            if (tmp_rel)
                res[i] = 0xffffffff;
            else
                res[i] = 0x00000000;
        }
        GR[r1] = concatenate2(res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

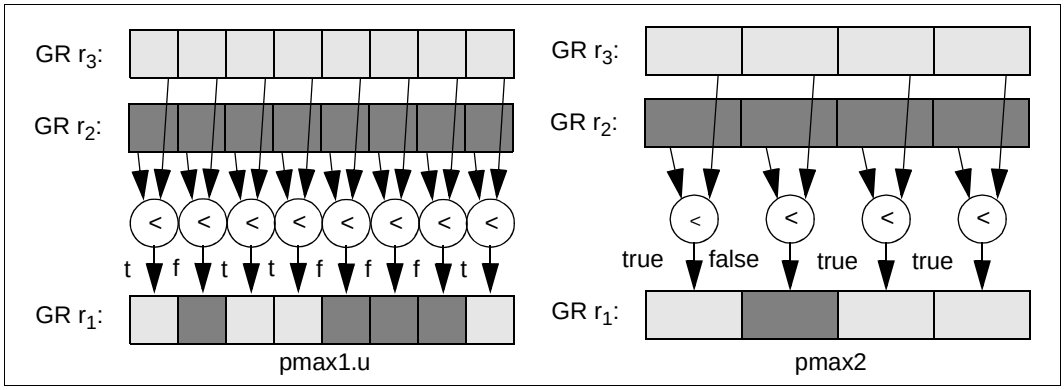
Parallel Maximum

Format:

(*qp*) pmax1.u $r_1 = r_2, r_3$	one_byte_form	I2
(*qp*) pmax2 $r_1 = r_2, r_3$	two_byte_form	I2

Description: The maximum of the two source operands is placed in the result register. In the one_byte_form, each unsigned 8-bit element of GR r_2 is compared with the corresponding unsigned 8-bit element of GR r_3 and the greater of the two is placed in the corresponding 8-bit element of GR r_1. In the two_byte_form, each signed 16-bit element of GR r_2 is compared with the corresponding signed 16-bit element of GR r_3 and the greater of the two is placed in the corresponding 16-bit element of GR r_1.

Figure 2-32. Parallel Maximum Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                  // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            res[i] = (zero_ext(x[i],8) < zero_ext(y[i],8)) ? y[i] : x[i];
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
    } else {                                              // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            res[i] = (sign_ext(x[i],16) < sign_ext(y[i],16)) ? y[i] : x[i];
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

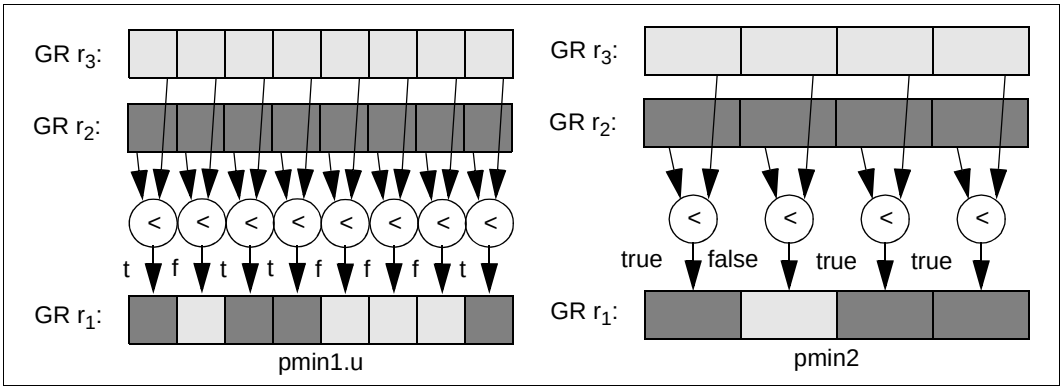
Parallel Minimum

Format:

(*qp*) pmin1.u $r_1 = r_2, r_3$	one_byte_form	I2
(*qp*) pmin2 $r_1 = r_2, r_3$	two_byte_form	I2

Description: The minimum of the two source operands is placed in the result register. In the one_byte_form, each unsigned 8-bit element of GR r_2 is compared with the corresponding unsigned 8-bit element of GR r_3 and the smaller of the two is placed in the corresponding 8-bit element of GR r_1. In the two_byte_form, each signed 16-bit element of GR r_2 is compared with the corresponding signed 16-bit element of GR r_3 and the smaller of the two is placed in the corresponding 16-bit element of GR r_1.

Figure 2-33. Parallel Minimum Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                  // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            res[i] = (zero_ext(x[i],8) < zero_ext(y[i],8)) ? x[i] : y[i];
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
    } else {                                              // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            res[i] = (sign_ext(x[i],16) < sign_ext(y[i],16)) ? x[i] : y[i];
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

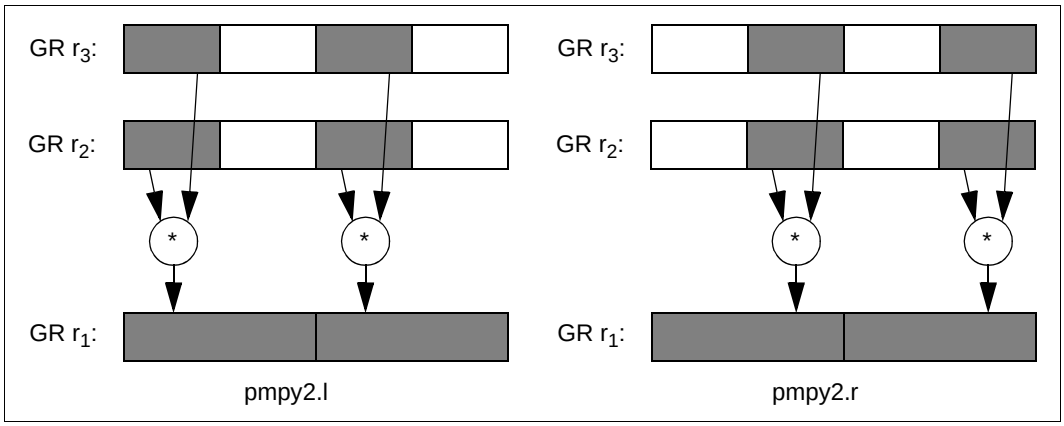
Parallel Multiply

Format:

(*qp*) pmpy2.r $r_1 = r_2, r_3$	right_form	I2
(*qp*) pmpy2.l $r_1 = r_2, r_3$	left_form	I2

Description: Two signed 16-bit data elements of GR r_2 are multiplied by the corresponding two signed 16-bit data elements of GR r_3 as shown in Figure 2-34. The two 32-bit results are placed in GR r_1.

Figure 2-34. Parallel Multiply Operation



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (right_form) {
        GR[r1]{31:0} = sign_ext(GR[r2]{15:0}, 16) *
                       sign_ext(GR[r3]{15:0}, 16);
        GR[r1]{63:32} = sign_ext(GR[r2]{47:32}, 16) *
                        sign_ext(GR[r3]{47:32}, 16);
    } else {                                          // left_form
        GR[r1]{31:0} = sign_ext(GR[r2]{31:16}, 16) *
                       sign_ext(GR[r3]{31:16}, 16);
        GR[r1]{63:32} = sign_ext(GR[r2]{63:48}, 16) *
                        sign_ext(GR[r3]{63:48}, 16);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Parallel Multiply and Shift Right

Format:

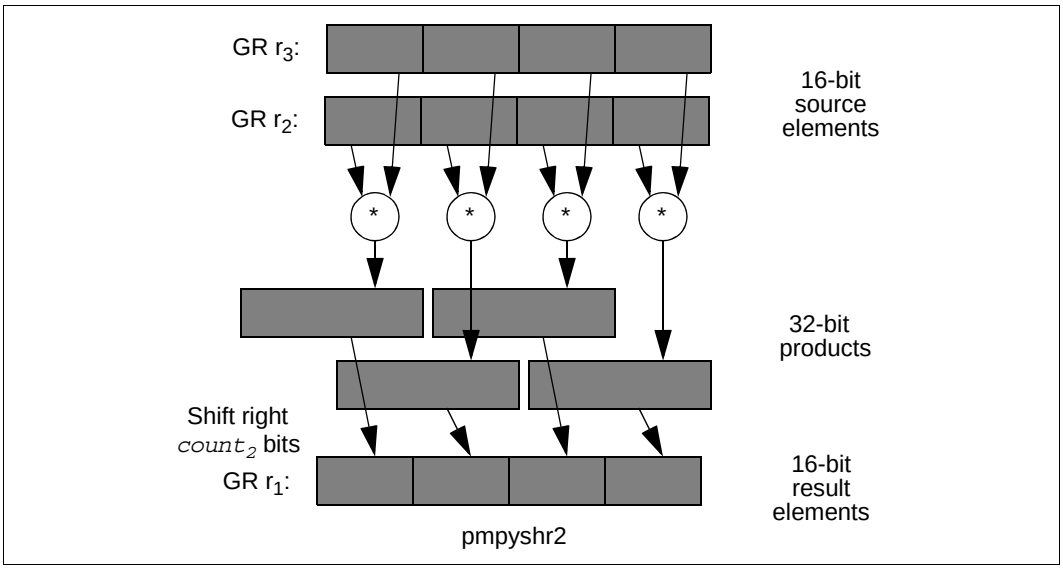
(*qp*) pmpyshr2 $r_1 = r_2, r_3, count_2$	signed_form	I1
(*qp*) pmpyshr2.u $r_1 = r_2, r_3, count_2$	unsigned_form	I1

Description: The four 16-bit data elements of GR r_2 are multiplied by the corresponding four 16-bit data elements of GR r_3 as shown in Figure 2-35. This multiplication can either be signed (`pmpyshr2`), or unsigned (`pmpyshr2.u`). Each product is then shifted to the right $count_2$ bits, and the least-significant 16-bits of each shifted product form 4 16-bit results, which are placed in GR r_1. A $count_2$ of 0 gives the 16 low bits of the results, a $count_2$ of 16 gives the 16 high bits of the results. The allowed values for $count_2$ are given in Table 2-44.

Table 2-44. PMPYSHR Shift Options

$count_2$	Selected Bit Field from Each 32-bit Product
0	15:0
7	22:7
15	30:15
16	31:16

Figure 2-35. Parallel Multiply and Shift Right Operation



Operation:

```
if (PR[qp]) {
    check_target_register(r1);
    x[0] = GR[r2]{15:0};    y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};   y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};   y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};   y[3] = GR[r3]{63:48};
    for (i = 0; i < 4; i++) {
        if (unsigned_form)                          // unsigned multiplication
            temp[i] = zero_ext(x[i], 16) * zero_ext(y[i], 16);
        else                                        // signed multiplication
            temp[i] = sign_ext(x[i], 16) * sign_ext(y[i], 16);

        res[i] = temp[i]{(count2 + 15):count2};
    }

    GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Population Count

Format: (*qp*) popcnt $r_1 = r_3$ I9

Description: The number of bits in GR r_3 having the value 1 is counted, and the resulting sum is placed in GR r_1.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    res = 0;
    // Count up all the one bits
    for (i = 0; i < 64; i++) {
        res += GR[r3]{i};
    }

    GR[r1] = res;
    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Probe Access

Format:

(*qp*) probe.r $r_1 = r_3, r_2$	read_form, register_form	M38
(*qp*) probe.w $r_1 = r_3, r_2$	write_form, register_form	M38
(*qp*) probe.r $r_1 = r_3, imm_2$	read_form, immediate_form	M39
(*qp*) probe.w $r_1 = r_3, imm_2$	write_form, immediate_form	M39
(*qp*) probe.r.fault r_3, imm_2	fault_form, read_form, immediate_form	M40
(*qp*) probe.w.fault r_3, imm_2	fault_form, write_form, immediate_form	M40
(*qp*) probe.rw.fault r_3, imm_2	fault_form, read_write_form, immediate_form	M40

Description:

This instruction determines whether read or write access, with a specified privilege level, to a given virtual address is permitted. GR r_1 is set to 1 if the specified access is allowed and to 0 otherwise. In the fault_form, if the specified access is allowed this instruction does nothing; if the specified access is not allowed, a fault is taken.

When PSR.dt is 1, the DTLB and the VHPT are queried for present translations to determine if access to the virtual address specified by GR r_3 bits {60:0} and the region register indexed by GR r_3 bits {63:61}, is permitted at the privilege level given by either GR r_2 bits{1:0} or imm_2. If PSR.pk is 1, protection key checks are also performed. The read or write form specifies whether the instruction checks for read or write access, or both.

When PSR.dt is 0, a non-faulting probe uses its address operand as a virtual address to query the DTLB only, because the VHPT walker is disabled. If the probed address is found in the DTLB, the non-faulting probe returns the appropriate value, if not an Alternate Data TLB fault is raised.

When PSR.dt is 0, a faulting probe treats its address operand as a physical address, and takes no TLB related faults.

A non-faulting probe to an unimplemented virtual address returns 0. A faulting probe to an unimplemented virtual address (when PSR.dt is 1) or unimplemented physical address (when PSR.dt is 0) takes an Unimplemented Data Address fault.

If this instruction faults, then it will set the non-access bit in the ISR and set the ISR read or write bits depending on the completer. The following faults are taken by the faulting form of the probe instruction only (the non-faulting form of the instruction does not take them): Unimplemented Data Address fault, Data Key Permissions fault, Data Access Rights fault, Data Dirty Bit fault, Data Access Bit fault, and Data Debug fault.

This instruction can only probe with equal or lower privilege levels. If the specified privilege level is higher (lower number), then the probe is performed with the current privilege level.

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS;
    itype |= (read_write_form) ? READ|WRITE : ((write_form) ? WRITE : READ);
    itype |= (fault_form) ? PROBE_FAULT : PROBE;

    if (!fault_form)
        check_target_register(r1);

    if (GR[r3].nat || (register_form ? GR[r2].nat : 0))
        register_nat_consumption_fault(itype);

    tmp_pl = (register_form) ? GR[r2]{1:0} : imm2;
    if (tmp_pl < PSR.cpl)
        tmp_pl = PSR.cpl;

    if (fault_form) {
        tlb_translate(GR[r3], 1, itype, tmp_pl, &mattr, &defer);
    } else {
        GR[r1] = tlb_grant_permission(GR[r3], itype, tmp_pl);
        GR[r1].nat = 0;
    }
}
```

Interruptions:

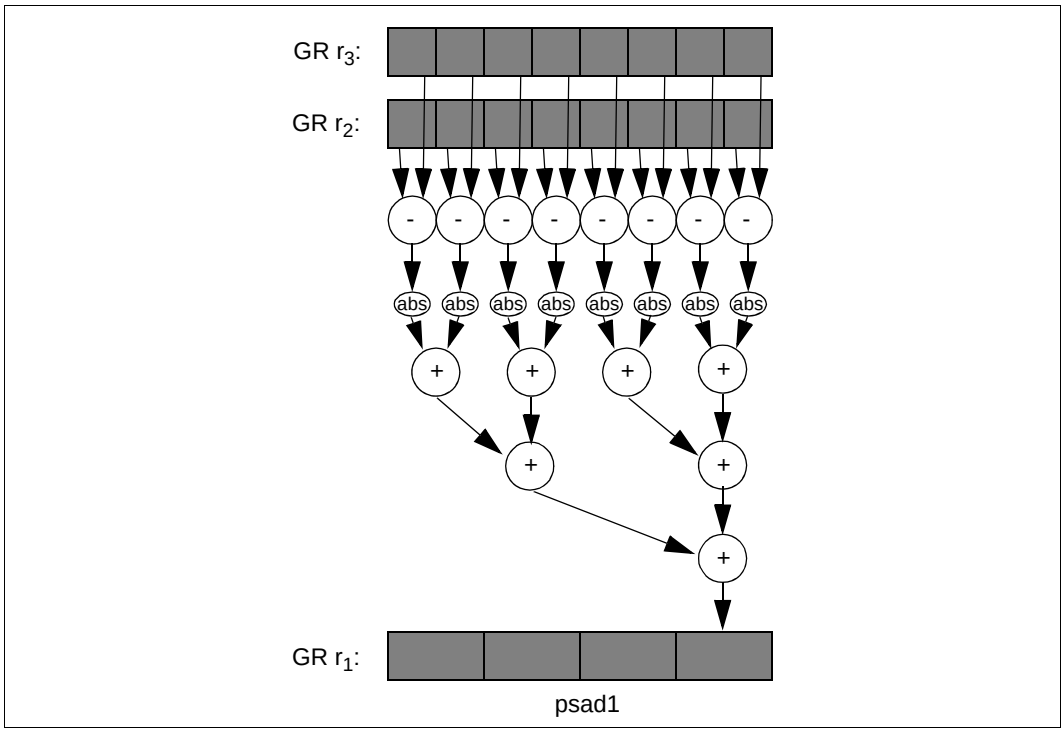
Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault

Parallel Sum of Absolute Difference

Format: (*qp*) psad1 $r_1 = r_2, r_3$ I2

Description: The unsigned 8-bit elements of GR r_2 are subtracted from the unsigned 8-bit elements of GR r_3. The absolute value of each difference is accumulated across the elements and placed in GR r_1.

Figure 2-36. Parallel Sum of Absolute Difference Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
    x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
    x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
    x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
    x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
    x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
    x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
    x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

    GR[r1] = 0;
    for (i = 0; i < 8; i++) {
        temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
        if (temp[i] < 0)
            temp[i] = -temp[i];
        GR[r1] += temp[i];
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

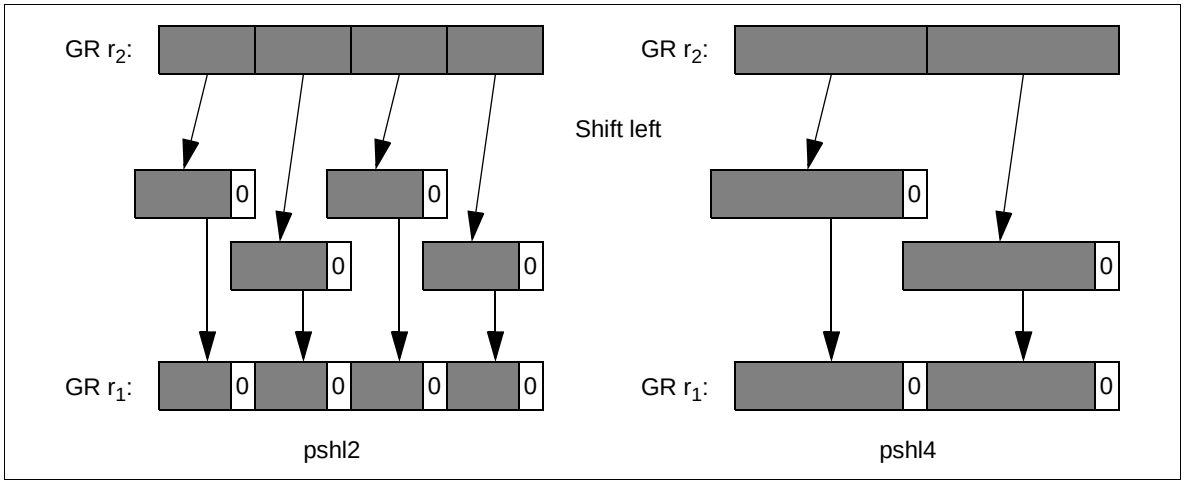
Interruptions: Illegal Operation fault

Parallel Shift Left

Format:	(*qp*) pshl2 $r_1 = r_2, r_3$	two_byte_form, variable_form	I7
	(*qp*) pshl2 $r_1 = r_2, count_5$	two_byte_form, fixed_form	I8
	(*qp*) pshl4 $r_1 = r_2, r_3$	four_byte_form, variable_form	I7
	(*qp*) pshl4 $r_1 = r_2, count_5$	four_byte_form, fixed_form	I8

Description: The data elements of GR r_2 are each independently shifted to the left by the scalar shift count in GR r_3, or in the immediate field $count_5$. The low-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 15 (for 16-bit quantities) or 31 (for 32-bit quantities) yield all zero results. The results are placed in GR r_1.

Figure 2-37. Parallel Shift Left Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    shift_count = (variable_form ? GR[r3] : count5);
    tmp_nat = (variable_form ? GR[r3].nat : 0);

    if (two_byte_form) {                              // two_byte_form
        if (shift_count u> 16)
            shift_count = 16;
        GR[r1]{15:0}  = GR[r2]{15:0}  << shift_count;
        GR[r1]{31:16} = GR[r2]{31:16} << shift_count;
        GR[r1]{47:32} = GR[r2]{47:32} << shift_count;
        GR[r1]{63:48} = GR[r2]{63:48} << shift_count;
    } else {                                          // four_byte_form
        if (shift_count u> 32)
            shift_count = 32;
        GR[r1]{31:0}  = GR[r2]{31:0}  << shift_count;
        GR[r1]{63:32} = GR[r2]{63:32} << shift_count;
    }

    GR[r1].nat = GR[r2].nat || tmp_nat;
}
```

Interruptions: Illegal Operation fault

Parallel Shift Left and Add

Format: (*qp*) pshladd2 $r_1 = r_2, count_2, r_3$ A10

Description: The four signed 16-bit data elements of GR r_2 are each independently shifted to the left by $count_2$ bits (shifting zeros into the low-order bits), and added to the four signed 16-bit data elements of GR r_3. Both the left shift and the add operations are saturating: if the result of either the shift or the add is not representable as a signed 16-bit value, the final result is saturated. The four signed 16-bit results are placed in GR r_1. The first operand can be shifted by 1, 2 or 3 bits.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

    max = sign_ext(0x7fff, 16);
    min = sign_ext(0x8000, 16);

    for (i = 0; i < 4; i++) {
        temp[i] = sign_ext(x[i], 16) << count2;

        if (temp[i] > max)
            res[i] = max;
        else if (temp[i] < min)
            res[i] = min;
        else {
            res[i] = temp[i] + sign_ext(y[i], 16);
            if (res[i] > max)
                res[i] = max;
            if (res[i] < min)
                res[i] = min;
        }
    }

    GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Parallel Shift Right

Format:

(*qp*) pshr2 $r_1 = r_3, r_2$	signed_form, two_byte_form, variable_form	I5
(*qp*) pshr2 $r_1 = r_3, count_5$	signed_form, two_byte_form, fixed_form	I6
(*qp*) pshr2.u $r_1 = r_3, r_2$	unsigned_form, two_byte_form, variable_form	I5
(*qp*) pshr2.u $r_1 = r_3, count_5$	unsigned_form, two_byte_form, fixed_form	I6
(*qp*) pshr4 $r_1 = r_3, r_2$	signed_form, four_byte_form, variable_form	I5
(*qp*) pshr4 $r_1 = r_3, count_5$	signed_form, four_byte_form, fixed_form	I6
(*qp*) pshr4.u $r_1 = r_3, r_2$	unsigned_form, four_byte_form, variable_form	I5
(*qp*) pshr4.u $r_1 = r_3, count_5$	unsigned_form, four_byte_form, fixed_form	I6

Description: The data elements of GR r_3 are each independently shifted to the right by the scalar shift count in GR r_2, or in the immediate field $count_5$. The high-order bits of each element are filled with either the initial value of the sign bits of the data elements in GR r_3 (arithmetic shift) or zeros (logical shift). The shift count is interpreted as unsigned. Shift counts greater than 15 (for 16-bit quantities) or 31 (for 32-bit quantities) yield all zero or all one results depending on the initial values of the sign bits of the data elements in GR r_3 and whether a signed or unsigned shift is done. The results are placed in GR r_1.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    shift_count = (variable_form ? GR[r2] : count5);
    tmp_nat = (variable_form ? GR[r2].nat : 0);

    if (two_byte_form) {                                  // two_byte_form
        if (shift_count u> 16)
            shift_count = 16;
        if (unsigned_form) {                              // unsigned shift
            GR[r1]{15:0}  = shift_right_unsigned(zero_ext(GR[r3]{15:0},  16),
                                                 shift_count);
            GR[r1]{31:16} = shift_right_unsigned(zero_ext(GR[r3]{31:16}, 16),
                                                 shift_count);
            GR[r1]{47:32} = shift_right_unsigned(zero_ext(GR[r3]{47:32}, 16),
                                                 shift_count);
            GR[r1]{63:48} = shift_right_unsigned(zero_ext(GR[r3]{63:48}, 16),
                                                 shift_count);
        } else {                                              // signed shift
            GR[r1]{15:0}  = shift_right_signed(sign_ext(GR[r3]{15:0},  16),
                                               shift_count);
            GR[r1]{31:16} = shift_right_signed(sign_ext(GR[r3]{31:16}, 16),
                                               shift_count);
            GR[r1]{47:32} = shift_right_signed(sign_ext(GR[r3]{47:32}, 16),
                                               shift_count);
            GR[r1]{63:48} = shift_right_signed(sign_ext(GR[r3]{63:48}, 16),
                                               shift_count);
        }
    } else {                                              // four_byte_form
        if (shift_count > 32)
            shift_count = 32;
        if (unsigned_form) {                              // unsigned shift
            GR[r1]{31:0}  = shift_right_unsigned(zero_ext(GR[r3]{31:0},  32),
                                                 shift_count);
            GR[r1]{63:32} = shift_right_unsigned(zero_ext(GR[r3]{63:32}, 32),
                                                 shift_count);
        } else {                                              // signed shift
            GR[r1]{31:0}  = shift_right_signed(sign_ext(GR[r3]{31:0},  32),
                                               shift_count);
```

```
            GR[r1]{63:32} = shift_right_signed(sign_ext(GR[r3]{63:32}, 32),
                                                    shift_count);
        }
    }

    GR[r1].nat = GR[r3].nat || tmp_nat;
}
```

Interruptions: Illegal Operation fault

Parallel Shift Right and Add

Format: (*qp*) pshradd2 $r_1 = r_2, count_2, r_3$ A10

Description: The four signed 16-bit data elements of GR r_2 are each independently shifted to the right by $count_2$ bits, and added to the four signed 16-bit data elements of GR r_3. The right shift operation fills the high-order bits of each element with the initial value of the sign bits of the data elements in GR r_2. The add operation is performed with signed saturation. The four signed 16-bit results of the add are placed in GR r_1. The first operand can be shifted by 1, 2 or 3 bits.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

    max = sign_ext(0x7fff, 16);
    min = sign_ext(0x8000, 16);

    for (i = 0; i < 4; i++) {
        temp[i] = shift_right_signed(sign_ext(x[i], 16), count2);

        res[i] = temp[i] + sign_ext(y[i], 16);
        if (res[i] > max)
            res[i] = max;
        if (res[i] < min)
            res[i] = min;
    }

    GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Parallel Subtract

Format:

(*qp*) psub1 $r_1 = r_2, r_3$	one_byte_form, modulo_form	A9
(*qp*) psub1.sss $r_1 = r_2, r_3$	one_byte_form, sss_saturation_form	A9
(*qp*) psub1.uus $r_1 = r_2, r_3$	one_byte_form, uus_saturation_form	A9
(*qp*) psub1.uuu $r_1 = r_2, r_3$	one_byte_form, uuu_saturation_form	A9
(*qp*) psub2 $r_1 = r_2, r_3$	two_byte_form, modulo_form	A9
(*qp*) psub2.sss $r_1 = r_2, r_3$	two_byte_form, sss_saturation_form	A9
(*qp*) psub2.uus $r_1 = r_2, r_3$	two_byte_form, uus_saturation_form	A9
(*qp*) psub2.uuu $r_1 = r_2, r_3$	two_byte_form, uuu_saturation_form	A9
(*qp*) psub4 $r_1 = r_2, r_3$	four_byte_form, modulo_form	A9

Description: The sets of elements from the two source operands are subtracted, and the results placed in GR r_1.

If the difference between two elements cannot be represented in the result element and a saturation completer is specified, then saturation clipping is performed. The saturation can either be signed or unsigned, as given in Table 2-45. If the difference of two elements is larger than the upper limit value, the result is the upper limit value. If it is smaller than the lower limit value, the result is the lower limit value. The saturation limits are given in Table 2-46.

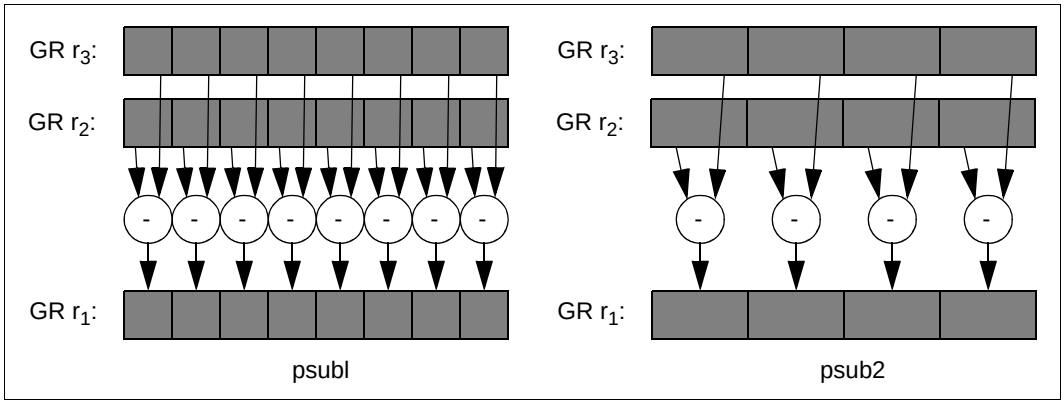
Table 2-45. Parallel Subtract Saturation Completers

Completer	Result r_1 Treated As	Source r_2 Treated As	Source r_3 Treated As
sss	signed	signed	signed
uus	unsigned	unsigned	signed
uuu	unsigned	unsigned	unsigned

Table 2-46. Parallel Subtract Saturation Limits

Size	Element Width	Result r_1 Signed		Result r_1 Unsigned	
		Upper Limit	Lower Limit	Upper Limit	Lower Limit
1	8 bit	0x7f	0x80	0xff	0x00
2	16 bit	0x7fff	0x8000	0xffff	0x0000

Figure 2-38. Parallel Subtract Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                  // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (sss_saturation_form) {                        // sss_saturation_form
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);
            for (i = 0; i < 8; i++) {
                temp[i] = sign_ext(x[i], 8) - sign_ext(y[i], 8);
            }
        } else if (uus_saturation_form) {                 // uus_saturation_form
            max = 0xff;
            min = 0x00;
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - sign_ext(y[i], 8);
            }
        } else if (uuu_saturation_form) {                 // uuu_saturation_form
            max = 0xff;
            min = 0x00;
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            }
        } else {                                          // modulo_form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            }
        }

        if (sss_saturation_form || uus_saturation_form ||
            uuu_saturation_form) {
            for (i = 0; i < 8; i++) {
                if (temp[i] > max)
                    temp[i] = max;
                if (temp[i] < min)
                    temp[i] = min;
            }
        }

        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                              temp[3], temp[2], temp[1], temp[0]);
    } else if (two_byte_form) {                           // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (sss_saturation_form) {                        // sss_saturation_form
            max = sign_ext(0x7fff, 16);
            min = sign_ext(0x8000, 16);
            for (i = 0; i < 4; i++) {
                temp[i] = sign_ext(x[i], 16) - sign_ext(y[i], 16);
            }
        } else if (uus_saturation_form) {                 // uus_saturation_form
```

```
            max = 0xffff;
            min = 0x0000;
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - sign_ext(y[i], 16);
            }
        } else if (uuu_saturation_form) {           // uuu_saturation_form
            max = 0xffff;
            min = 0x0000;
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            }
        } else {                                    // modulo_form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            }
        }

        if (sss_saturation_form || uus_saturation_form ||
            uuu_saturation_form) {
            for (i = 0; i < 4; i++) {
                if (temp[i] > max)
                    temp[i] = max;
                if (temp[i] < min)
                    temp[i] = min;
            }
        }

        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);
    } else {                                        // four-byte elements
        x[0] = GR[r2]{31:0};    y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};   y[1] = GR[r3]{63:32};

        for (i = 0; i < 2; i++) {                   // modulo_form
            temp[i] = zero_ext(x[i], 32) - zero_ext(y[i], 32);
        }

        GR[r1] = concatenate2(temp[1], temp[0]);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Purge Translation Cache Entry

Format: (*qp*) ptc.e r_3 M28

Description: One or more translation entries are purged from the local processor's instruction and data translation cache. Translation Registers and the VHPT are not modified.

The number of translation cache entries purged is implementation specific. Some implementations may purge all levels of the translation cache hierarchy with one iteration of `ptc.e`, while other implementations may require several iterations to flush all levels, sets and associativities of both instruction and data translation caches. GR r_3 specifies an implementation specific parameter associated with each iteration.

The following loop is defined to flush the entire translation cache for all processor models. Software can acquire parameters through a processor dependent layer that is accessed through a procedural interface. The selected region registers must remain unchanged during the loop.

```
disable_interrupts();
addr = base;
for (i = 0; i < count1; i++) {
    for (j = 0; j < count2; j++) {
           ptc.e(addr);
           addr += stride2;
       }
    addr += stride1;
}
enable_interrupts();
```

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat)
        register_nat_consumption_fault(0);
    tlb_purge_translation_cache(GR[r3]);
}
```

Interruptions: Privileged Operation fault Register NaT Consumption fault

Serialization: Software must issue a data serialization operation to ensure the purge is complete before issuing a data access or non-access reference dependent upon the purge. Software must issue instruction serialize operation before fetching an instruction dependent upon the purge.

Purge Global Translation Cache

Format:

(*qp*) ptc.g r_3, r_2	global_form	M45
(*qp*) ptc.ga r_3, r_2	global_alat_form	M45

Description: The instruction and data translation cache for each processor in the local TLB coherence domain are searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. These entries are removed.

The purge virtual address is specified by GR r_3 bits{60:0} and the purge region identifier is selected by GR r_3 bits {63:61}. GR r_2 specifies the address range of the purge as 1<<GR[r_2]{7:2} bytes in size.

Based on the processor model, the translation cache may be also purged of more translations than specified by the purge parameters up to and including removal of all entries within the translation cache.

`ptc.g` has release semantics and is guaranteed to be made visible after all previous data memory accesses are made visible. The memory fence instruction forces all processors to complete the purge prior to any subsequent memory operations. Serialization is still required to observe the side-effects of a translation being removed.

`ptc.g` must be the last instruction in an instruction group; otherwise, its behavior (including its ordering semantics) is undefined.

The behavior of the `ptc.ga` instruction is similar to `ptc.g`. In addition to the behavior specified for `ptc.g` the `ptc.ga` instruction encodes an extra bit of information in the broadcast transaction. This information specifies the purge is due to a page remapping as opposed to a protection change or page tear down. The remote processors within the coherency domain will then take what ever additional action is necessary to make their ALAT consistent. The local ALAT is not purged.

This instruction can only be executed at the most privileged level.

Only one global purge transaction may be issued at a time by all processors, the operation is undefined otherwise. Software is responsible for enforcing this restriction.

Propagation of `ptc.g` between multiple local TLB coherence domains is platform dependent, and must be handled by software. It is expected that the local TLB coherence domain covers at least the processors on the same local bus.

Operation:

```
if (PR[qp]) {
    if (!followed_by_stop())
        undefined_behavior();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3]))
        unimplemented_data_address_fault(0);

    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);
    tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);

    if (global_alat_form) tmp_ptc_type = GLOBAL_ALAT_FORM;
    else tmp_ptc_type = GLOBAL_FORM;

    tlb_broadcast_purge(tmp_rid, tmp_va, tmp_size, tmp_ptc_type);
}
```

Interruptions: Machine Check abort
Privileged Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault

Serialization: The broadcast purge TC is not synchronized with the instruction stream on a remote processor. Software cannot depend on any such synchronization with the instruction stream. Hardware on the remote machine cannot reload an instruction from memory or cache after acknowledging a broadcast purge TC without first retranslating the I-side access in the TLB. Hardware may continue to use a valid private copy of the instruction stream data (possibly in an I-buffer) obtained prior to acknowledging a broadcast purge TC to a page containing the i-stream data. Hardware must retranslate access to an instruction page upon an interruption or any explicit or implicit instruction serialization event (e.g. `srlz.i`, `rfi`).

Software must issue the appropriate data and/or instruction serialization operation to ensure the purge is completed before a local data access, non-access reference, or local instruction fetch access dependent upon the purge.

Purge Local Translation Cache

Format: (*qp*) ptc.l r_3, r_2 M45

Description: The instruction and data translation cache of the local processor is searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. All these entries are removed.

The purge virtual address is specified by GR r_3 bits{60:0} and the purge region identifier is selected by GR r_3 bits {63:61}. GR r_2 specifies the address range of the purge as 1<<GR[r_2]{7:2} bytes in size.

The processor ensures that all entries matching the purging parameters are removed. However, based on the processor model, the translation cache may be also purged of more translations than specified by the purge parameters up to and including removal of all entries within the translation cache.

This instruction can only be executed at the most privileged level.

This is a local operation, no purge broadcast to other processors occurs in a multiprocessor system.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3]))
        unimplemented_data_address_fault(0);

    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);
    tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
}
```

Interruptions:

Machine Check abort
Privileged Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault

Serialization: Software must issue the appropriate data and/or instruction serialization operation to ensure the purge is completed before a data access, non-access reference, or instruction fetch access dependent upon the purge.

Purge Translation Register

Format:

(*qp*) ptr.d r_3, r_2	data_form	M45
(*qp*) ptr.i r_3, r_2	instruction_form	M45

Description: In the data form of this instruction, the data translation registers and caches are searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. All these entries are removed. Entries in the instruction translation registers are unaffected by the data form of the purge.

In the instruction form, the instruction translation registers and caches are searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. All these entries are removed. Entries in the data translation registers are unaffected by the instruction form of the purge.

In addition, in both forms, the instruction and data translation cache may be purged of more translations than specified by the purge parameters up to and including removal of all entries within the translation cache.

The purge virtual address is specified by GR r_3 bits{60:0} and the purge region identifier is selected by GR r_3 bits {63:61}. GR r_2 specifies the address range of the purge as 1<<GR[r_2]{7:2} bytes in size.

This instruction can only be executed at the most privileged level.

This is a local operation, no purge broadcast to other processors occurs in a multiprocessor system.

As described in "Translation Cache (TC)" on page 4-5 in Volume 2, the processor may use the translation caches to cache virtual address mappings held by translation registers. The `ptr.i` and `ptr.d` instructions purge the processor's translation registers as well as cached translation register copies that may be contained in the respective translation caches.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3]))
        unimplemented_data_address_fault(0);

    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    if (data_form) {
        tlb_must_purge_dtr_entries(tmp_rid, tmp_va, tmp_size);
        tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
    } else {                                        // instruction_form
        tlb_must_purge_itr_entries(tmp_rid, tmp_va, tmp_size);
        tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    }
}
```

Interruptions: Privileged Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault

Serialization: For the data form, software must issue a data serialization operation to ensure the purge is completed before issuing an instruction dependent upon the purge. For the instruction form, software must issue an instruction serialization operation to ensure the purge is completed before fetching an instruction dependent on that purge.

Return From Interruption

Format: rfi B8

Description: The machine context prior to an interruption is restored. PSR is restored from IPSR, IPSR is unmodified, and IP is restored from IIP. Execution continues at the bundle address loaded into the IP, and the instruction slot loaded into PSR.ri.

This instruction must be immediately followed by a stop. Otherwise, an Illegal Operation fault is taken. This instruction switches to the register bank specified by IPSR.bn. Instructions in the same instruction group that access GR16 to GR31 reference the previous register bank. Subsequent instruction groups reference the new register bank.

This instruction performs instruction serialization, which ensures:

- Prior modifications to processor register resources that affect fetching of subsequent instruction groups are observed.
- Prior modifications to processor register resources that affect subsequent execution or data memory accesses are observed.
- Prior memory synchronization (`sync.i`) operations have taken effect on the local processor instruction cache.
- Subsequent instruction group fetches (including the target instruction group) are re-initiated after `rfi` completes.

The `rfi` instruction must be in an instruction group after the instruction group containing the operation that is to be serialized.

This instruction can only be executed at the most privileged level. This instruction can not be predicated.

Execution of this instruction is undefined if PSR.ic or PSR.i are 1. Software must ensure that an interruption cannot occur that could modify IIP, IPSR, or IFS between when they are written and the subsequent `rfi`.

This instruction does not take Lower Privilege Transfer, Taken Branch or Single Step traps.

If the target is a bundle containing a `movl` instruction and if this instruction sets PSR.ri to 2, then an Illegal Operation fault will be taken on the target bundle.

If IPSR.is is 1, control is resumed in the IA-32 instruction set at the virtual linear address specified by IIP{31:0}. PSR.di does not inhibit instruction set transitions for this instruction. If PSR.dfh is 1 after `rfi` completes execution, a Disabled FP Register fault is raised on the target IA-32 instruction.

If IPSR.is is 1 and an Unimplemented Instruction Address trap is taken, IIP will contain the original 64-bit target IP. (The value will not have been zero extended from 32 bits.)

All stacked general purpose integer registers are left in an undefined state. Software can not rely on the value of these registers across an instruction set transition. The current stack frame is set to zero. Software must ensure ar[BSPSTORE]==ar[BSP], otherwise undefined behavior may result.

Software must issue a `mf` instruction before this instruction if memory ordering is required between IA-32 processor-consistent and IA-64 unordered memory references. The processor does not ensure IA-64-instruction-set-generated writes into the instruction stream are seen by subsequent IA-32 instructions.

Software must ensure the code segment descriptor and selector are loaded before issuing this instruction. If the target EIP value exceeds the code segment limit or has a code segment privilege

violation, an IA-32_Exception(GPFault) exception is raised on the target IA-32 instruction. For entry into 16-bit IA-32 code, if IIP is not within 64K-bytes of CSD.base a GPFault is raised on the target instruction.

EFLAG.rf and PSR.id are unmodified until the successful completion of the target IA-32 instruction. PSR.da, PSR.dd, PSR.ia and PSR.ed are cleared to zero before the target IA-32 instruction begins execution.

IA-32 instruction set execution leaves the contents of the ALAT undefined. Software can not rely on ALAT state across an instruction set transition. On entry to IA-32 code, existing entries in the ALAT are ignored.

Operation:

```
if (!followed_by_stop())
    illegal_operation_fault();

unimplemented_address = 0;
if (PSR.cpl != 0)
    privileged_operation_fault(0);

taken_rfi = 1;

PSR = CR[IPSR];
if (CR[IPSR].is == 1) {         //resume IA-32 instruction set
    tmp_IP = CR[IIP];
    if ((CR[IPSR].it && unimplemented_virtual_address(tmp_IP))
        || (!CR[IPSR].it && unimplemented_physical_address(tmp_IP)))
        unimplemented_address = 1;
                                //compute effective instruction pointer
    EIP{31:0}= CR[IIP]{31:0} - AR[CSD].Base;
                                //force zero-sized restored frame
    rse_restore_frame(0, 0, CFM.sof);
    CFM.sof = 0;
    CFM.sol = 0;
    CFM.sor = 0;
    CFM.rrb.gr = 0;
    CFM.rrb.fr = 0;
    CFM.rrb.pr = 0;
    rse_invalidate_non_current_regs();
    //The register stack engine is disabled during IA-32
    //instruction set execution.
} else {                        //return to IA-64 instruction set
    tmp_IP = CR[IIP] & ~0xf;
    slot = CR[IPSR].ri;
    if ((CR[IPSR].it && unimplemented_virtual_address(tmp_IP))
        || (!CR[IPSR].it && unimplemented_physical_address(tmp_IP)))
        unimplemented_address = 1;
    if (CR[IFS].v) {
        tmp_growth = -CFM.sof;
        alat_frame_update(-CR[IFS].ifm.sof, 0);
        rse_restore_frame(CR[IFS].ifm.sof, tmp_growth, CFM.sof);
        CFM = CR[IFS].ifm;
    }
    rse_enable_current_frame_load();
}
IP = tmp_IP;
instruction_serialize();
if (unimplemented_address)
    unimplemented_instruction_address_trap(0, tmp_IP);
```

Interruptions: Illegal Operation fault Privileged Operation fault
Unimplemented Instruction Address trap

Additional Faults on IA-32 target instructions
IA-32_Exception(GPFault)
Disabled FP Reg Fault if PSR.dfh is 1

Serialization: An implicit instruction and data serialization operation is performed.

Reset System Mask

Format: (*qp*) rsm *imm*$_{24}$ M44

Description: The complement of the *imm*$_{24}$ operand is ANDed with the system mask (PSR{23:0}) and the result is placed in the system mask.

The PSR system mask can only be written at the most privileged level.

When the current privilege level is zero (PSR.cpl is 0), an `rsm` instruction whose mask includes PSR.i may cause external interrupts to be disabled for an implementation-dependent number of instructions, even if the qualifying predicate for the `rsm` instruction is false. Architecturally, the extents of this external interrupt disabling "window" are defined as follows:

- External interrupts may be disabled for any instructions in the same instruction group as the `rsm`, including those that precede the `rsm` in sequential program order, regardless of the value of the qualifying predicate of the `rsm` instruction.
- If the qualifying predicate of the `rsm` is true, then external interrupts are disabled immediately following the `rsm` instruction.
- If the qualifying predicate of the `rsm` is false, then external interrupts may be disabled until the next data serialization operation that follows the `rsm` instruction.

The external interrupt disable window is guaranteed to be no larger than defined by the above criteria, but it may be smaller, depending on the processor implementation.

When the current privilege level is non-zero (PSR.cpl is not 0), an `rsm` instruction whose mask includes PSR.i may briefly disable external interrupts, regardless of the value of the qualifying predicate of the `rsm` instruction. However, processor implementations guarantee that non-privileged code cannot lock out external interrupts indefinitely (e.g. via an arbitrarily long sequence of `rsm` instructions with zero-valued qualifying predicates).

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (is_reserved_field(PSR_TYPE, PSR_SM, imm24))
        reserved_register_field_fault();

    if (imm24{1})     PSR{1} = 0;
    if (imm24{2})     PSR{2} = 0;
    if (imm24{3})     PSR{3} = 0;
    if (imm24{4})     PSR{4} = 0;
    if (imm24{5})     PSR{5} = 0;
    if (imm24{13})    PSR{13} = 0;
    if (imm24{14})    PSR{14} = 0;
    if (imm24{15})    PSR{15} = 0;
    if (imm24{17})    PSR{17} = 0;
    if (imm24{18})    PSR{18} = 0;
    if (imm24{19})    PSR{19} = 0;
    if (imm24{20})    PSR{20} = 0;
    if (imm24{21})    PSR{21} = 0;
    if (imm24{22})    PSR{22} = 0;
    if (imm24{23})    PSR{23} = 0;
}
```

Interruptions: Privileged Operation fault Reserved Register/Field fault

Serialization: Software must use a data serialize or instruction serialize operation before issuing instructions dependent upon the altered PSR bits – except the PSR.i bit. The PSR.i bit is implicitly serialized and the processor ensures that external interrupts are masked by the time the next instruction executes.

Reset User Mask

Format: (*qp*) rum imm_{24} M44

Description: The complement of the imm_{24} operand is ANDed with the user mask (PSR{5:0}) and the result is placed in the user mask.

PSR.up is only cleared if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Operation:

```
if (PR[qp]) {
    if (is_reserved_field(PSR_TYPE, PSR_UM, imm24))
        reserved_register_field_fault();

    if (imm24{1})     PSR{1} = 0;
    if (imm24{2} && PSR.sp == 0)       //non-secure perf monitor
                      PSR{2} = 0;
    if (imm24{3})     PSR{3} = 0;
    if (imm24{4})     PSR{4} = 0;
    if (imm24{5})     PSR{5} = 0;
}
```

Interruptions: Reserved Register/Field fault

Serialization: All user mask modifications are observed by the next instruction group.

Set Floating-Point Value, Exponent, or Significand

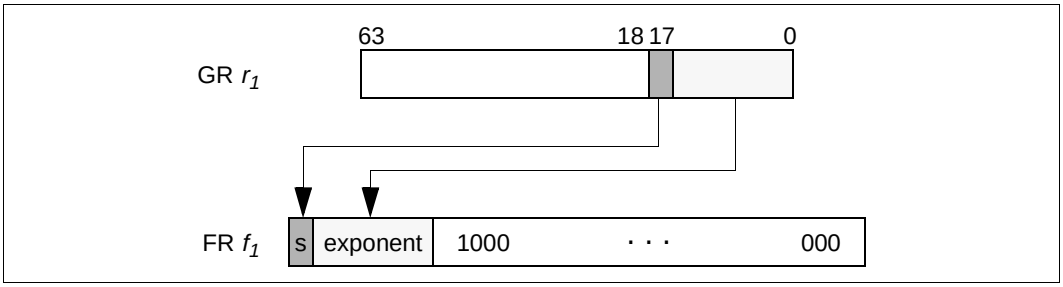
Format:

(*qp*) setf.s $f_1 = r_2$	single_form	M18
(*qp*) setf.d $f_1 = r_2$	double_form	M18
(*qp*) setf.exp $f_1 = r_2$	exponent_form	M18
(*qp*) setf.sig $f_1 = r_2$	significand_form	M18

Description: In the single and double forms, GR r_2 is treated as a single precision (in the single_form) or double precision (in the double_form) memory representation, converted into floating-point register format, and placed in FR f_1.

In the exponent_form, bits 16:0 of GR r_2 are copied to the exponent field of FR f_1 and bit 17 of GR r_2 is copied to the sign bit of FR f_1. The significand field of FR f_1 is set to one (0x800...000).

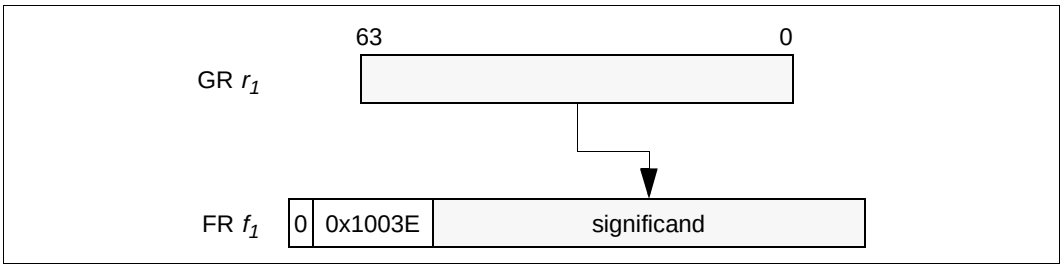
Figure 2-39. Function of setf.exp



In the significand_form, the value in GR r_2 is copied to the significand field of FR f_1.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

Figure 2-40. Function of setf.sig



For all forms, if the NaT bit corresponding to r_2 is equal to 1, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (!GR[r2].nat) {
        if (single_form)
            FR[f1] = fp_mem_to_fr_format(GR[r2], 4, 0);
        else if (double_form)
            FR[f1] = fp_mem_to_fr_format(GR[r2], 8, 0);
        else if (significand_form) {
            FR[f1].significand = GR[r2];
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = 0;
        } else {                                        // exponent_form
            FR[f1].significand = 0x8000000000000000;
            FR[f1].exp = GR[r2]{16:0};
            FR[f1].sign = GR[r2]{17};
        }
    } else
        FR[f1] = NATVAL;

    fp_update_psr(f1);
}
```

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

Shift Left

Format:
(*qp*) shl $r_1 = r_2, r_3$ I7
(*qp*) shl $r_1 = r_2, count_6$ pseudo-op of: (*qp*) dep.z $r_1 = r_2, count_6, 64–count_6$

Description: The value in GR r_2 is shifted to the left, with the vacated bit positions filled with zeroes, and placed in GR r_1. The number of bit positions to shift is specified by the value in GR r_3 or by an immediate value $count_6$. The shift count is interpreted as an unsigned number. If the value in GR r_3 is greater than 63, then the result is all zeroes.

See “Deposit” on p. 2-37 for the immediate form.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    count = GR[r3];
    GR[r1] = (count > 63) ? 0: GR[r2] << count;

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Shift Left and Add

Format: (*qp*) shladd $r_1 = r_2, count_2, r_3$ A2

Description: The first source operand is shifted to the left by $count_2$ bits and then added to the second source operand and the result placed in GR r_1. The first operand can be shifted by 1, 2, 3, or 4 bits.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = (GR[r2] << count2) + GR[r3];
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

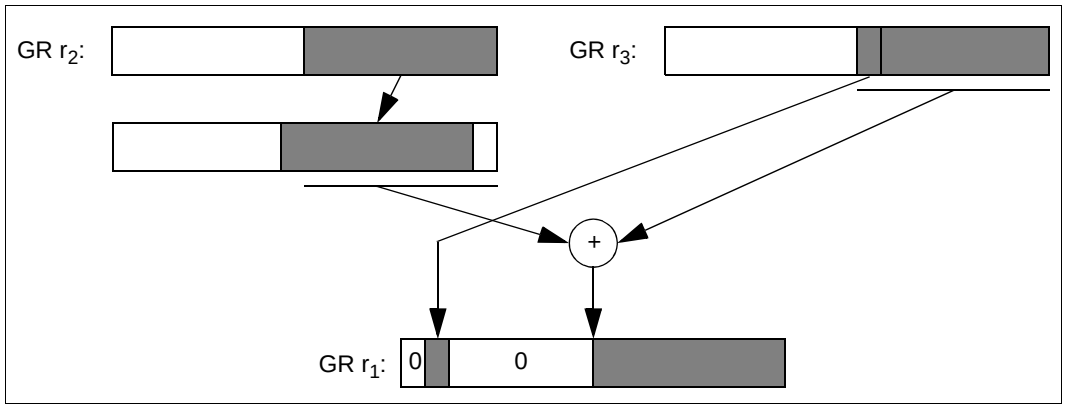
Interruptions: Illegal Operation fault

Shift Left and Add Pointer

Format: (*qp*) shladdp4 $r_1 = r_2, count_2, r_3$ A2

Description: The first source operand is shifted to the left by $count_2$ bits and then is added to the second source operand. The upper 32 bits of the result are forced to zero, and then bits {31:30} of GR r_3 are copied to bits {62:61} of the result. This result is placed in GR r_1. The first operand can be shifted by 1, 2, 3, or 4 bits.

Figure 2-41. Shift Left and Add Pointer



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_res = (GR[r2] << count2) + GR[r3];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[r3]{31:30};
    GR[r1] = tmp_res;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Shift Right

Format:

(*qp*) shr $r_1 = r_3, r_2$	signed_form	I5
(*qp*) shr.u $r_1 = r_3, r_2$	unsigned_form	I5
(*qp*) shr $r_1 = r_3, count_6$	pseudo-op of: (*qp*) extr $r_1 = r_3, count_6, 64–count_6$	
(*qp*) shr.u $r_1 = r_3, count_6$	pseudo-op of: (*qp*) extr.u $r_1 = r_3, count_6, 64–count_6$	

Description: The value in GR r_3 is shifted to the right and placed in GR r_1. In the signed_form the vacated bit positions are filled with bit 63 of GR r_3; in the unsigned_form the vacated bit positions are filled with zeroes. The number of bit positions to shift is specified by the value in GR r_2 or by an immediate value $count_6$. The shift count is interpreted as an unsigned number. If the value in GR r_2 is greater than 63, then the result is all zeroes (for the unsigned_form, or if bit 63 of GR r_3 was 0) or all ones (for the signed_form if bit 63 of GR r_3 was 1).

If the .u completer is specified, the shift is unsigned (logical), otherwise it is signed (arithmetic).

See “Extract” on p. 2-40 for the immediate forms.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (signed_form) {
        count = (GR[r2] > 63) ? 63 : GR[r2];
        GR[r1] = shift_right_signed(GR[r3], count);
    } else {
        count = GR[r2];
        GR[r1] = (count > 63) ? 0 : shift_right_unsigned(GR[r3], count);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

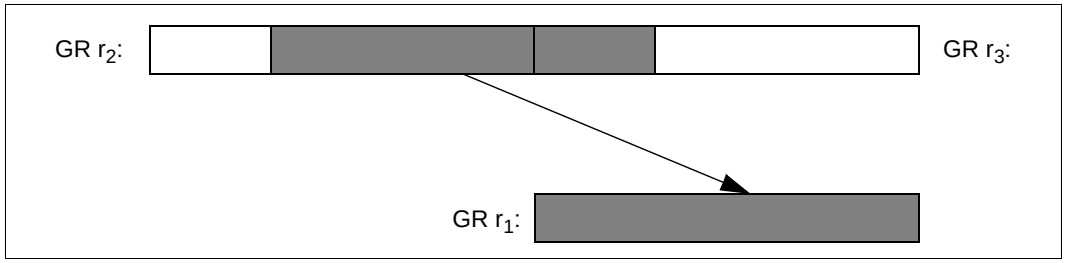
Shift Right Pair

Format: (*qp*) shrp $r_1 = r_2, r_3, count_6$ I10

Description: The two source operands, GR r_2 and GR r_3, are concatenated to form a 128-bit value and shifted to the right $count_6$ bits. The least-significant 64 bits of the result are placed in GR r_1.

The immediate value $count_6$ can be any number in the range 0 to 63.

Figure 2-42. Shift Right Pair



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    temp1 = shift_right_unsigned(GR[r3], count6);
    temp2 = GR[r2] << (64 - count6);
    GR[r1] = zero_ext(temp1, 64 - count6) | temp2;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Serialize

Format:

(*qp*) srlz.i	instruction_form	M24
(*qp*) srlz.d	data_form	M24

Description: Instruction serialization (`srlz.i`) ensures:

- Prior modifications to processor register resources that affect fetching of subsequent instruction groups are observed,
- Prior modifications to processor register resources that affect subsequent execution or data memory accesses are observed,
- Prior memory synchronization (`sync.i`) operations have taken effect on the local processor instruction cache,
- Subsequent instruction group fetches are re-initiated after `srlz.i` completes.

The `srlz.i` instruction must be in an instruction group after the instruction group containing the operation that is to be serialized. Operations dependent on the serialization must be in an instruction group after the instruction group containing the `srlz.i`.

Data serialization (`srlz.d`) ensures:

- Prior modifications to processor register resources that affect subsequent execution or data memory accesses are observed.

The `srlz.d` instruction must be in an instruction group after the instruction group containing the operation that is to be serialized. Operations dependent on the serialization must follow the `srlz.d`, but they can be in the same instruction group as the `srlz.d`.

A `srlz` cannot be used to stall processor data memory references until prior data memory references, or memory fences are visible or “accepted” by the external platform.

The following processor resources require a serialize to ensure side-effects are observed; CRs, PSR, DBRs, IBRs, PMDs, PMCs, RRs, PKRs, TRs and TCs (refer to Volume 2 for details).

Operation:

```
if (PR[qp]) {
    if (instruction_form)
        instruction_serialize();
    else       // data_form
        data_serialize();
}
```

Interruptions: None

Set System Mask

Format: (*qp*) ssm imm_{24} M44

Description: The imm_{24} operand is ORed with the system mask (PSR{23:0}) and the result is placed in the system mask.

The PSR system mask can only be written at the most privileged level.

The contents of the interruption resources (that are overwritten when the PSR.ic bit is 1), are undefined if an interruption occurs between the enabling of the PSR.ic bit and a subsequent instruction serialize operation.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (is_reserved_field(PSR_TYPE, PSR_SM, imm24))
        reserved_register_field_fault();

    if (imm24{1})     PSR{1} = 1;
    if (imm24{2})     PSR{2} = 1;
    if (imm24{3})     PSR{3} = 1;
    if (imm24{4})     PSR{4} = 1;
    if (imm24{5})     PSR{5} = 1;
    if (imm24{13})    PSR{13} = 1;
    if (imm24{14})    PSR{14} = 1;
    if (imm24{15})    PSR{15} = 1;
    if (imm24{17})    PSR{17} = 1;
    if (imm24{18})    PSR{18} = 1;
    if (imm24{19})    PSR{19} = 1;
    if (imm24{20})    PSR{20} = 1;
    if (imm24{21})    PSR{21} = 1;
    if (imm24{22})    PSR{22} = 1;
    if (imm24{23})    PSR{23} = 1;
}
```

Interruptions: Privileged Operation fault Reserved Register/Field fault

Serialization: Software must issue a data serialize or instruction serialize operation before issuing instructions dependent upon the altered PSR bits from the `ssm` instruction. Unlike with the `rsm` instruction, setting the PSR.i bit is not treated specially. Refer to Volume 2 for a description of serialization.

Store

Format:

(*qp*) st*sz*.*sttype*.*sthint* [r_3] = r_2	normal_form, no_base_update_form	M4
(*qp*) st*sz*.*sttype*.*sthint* [r_3] = r_2, imm_9	normal_form, imm_base_update_form	M5
(*qp*) st8.spill.*sthint* [r_3] = r_2	spill_form, no_base_update_form	M4
(*qp*) st8.spill.*sthint* [r_3] = r_2, imm_9	spill_form, imm_base_update_form	M5

Description: A value consisting of the least significant *sz* bytes of the value in GR r_2 is written to memory starting at the address specified by the value in GR r_3. The values of the *sz* completer are given in Table 2-30 on page 2-126. The *sttype* completer specifies special store operations, which are described in Table 2-47. If the NaT bit corresponding to GR r_3 is 1 (or in the normal_form, if the NaT bit corresponding to GR r_2 is 1), a Register NaT Consumption fault is taken.

In the spill_form, an 8-byte value is stored, and the NaT bit corresponding to GR r_2 is copied to a bit in the UNAT application register. This instruction is used for spilling a register/NaT pair.0 See Volume 1 for details.

In the imm_base_update form, the value in GR r_3 is added to a signed immediate value (imm_9) and the result is placed back in GR r_3. This base register update is done after the store, and does not affect the store address, nor the value stored (for the case where r_2 and r_3 specify the same register).

Table 2-47. Store Types

sttype Completer	Interpretation	Special Store Operation
none	Normal store	
rel	Ordered store	An ordered store is performed with release semantics.

For more details on ordered stores see Volume 1.

The ALAT is queried using the physical memory address and the access size, and all overlapping entries are invalidated.

The value of the *sthint* completer specifies the locality of the memory access. The values of the *sthint* completer are given in Table 2-48. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *sthint*. For more details, refer to Volume 1.

Table 2-48. Store Hints

sthint Completer	Interpretation
none	Temporal locality, level 1
nta	Non-temporal locality, all levels

Operation:

```
if (PR[qp]) {
    size = spill_form ? 8 : sz;
    otype = (sttype == 'rel') ? RELEASE : UNORDERED;

    if (imm_base_update_form)
        check_target_register(r3);
    if (GR[r3].nat || (normal_form && GR[r2].nat))
        register_nat_consumption_fault(WRITE);

    paddr = tlb_translate(GR[r3], size, WRITE, PSR.cpl, &mattr,
                          &tmp_unused);
    if (spill_form && GR[r2].nat)
        natd_gr_write(GR[r2], paddr, size, UM.be, mattr, otype, sthint);
    else
        mem_write(GR[r2], paddr, size, UM.be, mattr, otype, sthint);

    if (spill_form) {
        bit_pos = GR[r3]{8:3};
        AR[UNAT]{bit_pos} = GR[r2].nat;
    }

    alat_inval_multiple_entries(paddr, size);

    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = 0;
        mem_implicit_prefetch(GR[r3], sthint, WRITE);
    }
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault

Floating-Point Store

Format:	(*qp*) stf*fsz*.*sthint* [r_3] = f_2	normal_form, no_base_update_form	M9
	(*qp*) stf*fsz*.*sthint* [r_3] = f_2, imm_9	normal_form, imm_base_update_form	M10
	(*qp*) stf8.*sthint* [r_3] = f_2	integer_form, no_base_update_form	M9
	(*qp*) stf8.*sthint* [r_3] = f_2, imm_9	integer_form, imm_base_update_form	M10
	(*qp*) stf.spill.*sthint* [r_3] = f_2	spill_form, no_base_update_form	M9
	(*qp*) stf.spill.*sthint* [r_3] = f_2, imm_9	spill_form, imm_base_update_form	M10

Description: A value, consisting of *fsz* bytes, is generated from the value in FR f_2 and written to memory starting at the address specified by the value in GR r_3. In the normal_form, the value in FR f_2 is converted to the memory format and then stored. In the integer_form, the significand of FR f_2 is stored. The values of the *fsz* completer are given in Table 2-33 on page 2-130. In the normal_form or the integer_form, if the NaT bit corresponding to GR r_3 is 1 or if FR f_2 contains NaTVal, a Register NaT Consumption fault is taken. See Volume 1 for details on conversion from floating-point register format.

In the spill_form, a 16-byte value from FR f_2 is stored without conversion. This instruction is used for spilling a register. See Volume 1 for details.

In the imm_base_update form, the value in GR r_3 is added to a signed immediate value (imm_9) and the result is placed back in GR r_3. This base register update is done after the store, and does not affect the store address.

The ALAT is queried using the physical memory address and the access size, and all overlapping entries are invalidated.

The value of the *sthint* completer specifies the locality of the memory access. The values of the *sthint* completer are given in Table 2-48 on page 2-213. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *sthint*. For more details, refer to Volume 1.

Hardware support for `stfe` (10-byte) instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `stfe` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.

Operation:

```
if (PR[qp]) {
    if (imm_base_update_form)
        check_target_register(r3);
    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, WRITE);

    if (GR[r3].nat || (!spill_form && (FR[f2] == NATVAL)))
        register_nat_consumption_fault(WRITE);

    size = spill_form ? 16 : (integer_form ? 8 : fsz);

    paddr = tlb_translate(GR[r3], size, WRITE, PSR.cpl, &mattr, &tmp_unused);
    val = fp_fr_to_mem_format(FR[f2], size, integer_form);
    mem_write(val, paddr, size, UM.be, mattr, UNORDERED, sthint);

    alat_inval_multiple_entries(paddr, size);

    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = 0;
        mem_implicit_prefetch(GR[r3], sthint, WRITE);
    }
}
```

Interruptions:

Illegal Operation fault
Disabled Floating-point Register fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

Subtract

Format:

(*qp*) sub $r_1 = r_2, r_3$	register_form	A1
(*qp*) sub $r_1 = r_2, r_3, 1$	minus1_form, register_form	A1
(*qp*) sub $r_1 = imm_8, r_3$	imm8_form	A3

Description: The second source operand (and an optional constant 1) are subtracted from the first operand and the result placed in GR r_1. In the register form the first operand is GR r_2; in the immediate form the first operand is taken from the sign-extended imm_8 encoding field.

The minus1_form is available only in the register_form (although the equivalent effect can be achieved by adjusting the immediate).

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    if (minus1_form)
        GR[r1] = tmp_src - GR[r3] - 1;
    else
        GR[r1] = tmp_src - GR[r3];

    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Set User Mask

Format: (*qp*) sum imm_{24} M44

Description: The imm_{24} operand is ORed with the user mask (PSR{5:0}) and the result is placed in the user mask.

PSR.up can only be set if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Operation:

```
if (PR[qp]) {
    if (is_reserved_field(PSR_TYPE, PSR_UM, imm24))
        reserved_register_field_fault();

    if (imm24{1})     PSR{1} = 1;
    if (imm24{2} && PSR.sp == 0)       //non-secure perf monitor
                      PSR{2} = 1;
    if (imm24{3})     PSR{3} = 1;
    if (imm24{4})     PSR{4} = 1;
    if (imm24{5})     PSR{5} = 1;
}
```

Interruptions: Reserved Register/Field fault

Serialization: All user mask modifications are observed by the next instruction group.

Sign Extend

Format: (*qp*) sxt*xsz* $r_1 = r_3$ I29

Description: The value in GR r_3 is sign extended from the bit position specified by *xsz* and the result is placed in GR r_1. The mnemonic values for *xsz* are given in Table 2-49.

Table 2-49. *xsz* Mnemonic Values

xsz Mnemonic	Bit Position
1	7
2	15
4	31

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = sign_ext(GR[r3],xsz * 8);
    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Memory Synchronization

Format: (*qp*) sync.i M24

Description: `sync.i` ensures that when previously initiated Flush Cache (`fc`) operations issued by the local processor become visible to local data memory references, prior Flush Cache operations are also observed by the local processor instruction fetch stream. `sync.i` also ensures that at the time previously initiated Flush Cache (`fc`) operations are observed on a remote processor by data memory references they are also observed by instruction memory references on the remote processor. `sync.i` is ordered with respect to all cache flush operations as observed by another processor. A `sync.i` and a previous `fc` must be in separate instruction groups. If semantically required, the programmer must explicitly insert ordered data references (acquire, release or fence type) to appropriately constrain `sync.i` (and hence `fc`) visibility to the data stream on other processors.

`sync.i` is used to maintain an ordering relationship between instruction and data caches on local and remote processors. An instruction serialize operation must be used to ensure synchronization initiated by `sync.i` on the local processor has been observed by a given point in program execution.

An example of self-modifying code (local processor):

```
    st  [L1] = data   //store into local instruction stream
    fc  L1            //flush stale datum from instruction/data cache
    ;;                //require instruction boundary between fc and sync.i
    sync.i            //ensure local and remote data/inst caches
                      //are synchronized
    ;;
    srlz.i            //ensure sync has been observed by the local processor,
    ;;                //ensure subsequent instructions observe
                      //modified memory
L1: target            //instruction modified
```

Operation:

```
if (PR[qp]) {
    instruction_synchronize();
}
```

Interruptions: None

Translation Access Key

Format: (*qp*) tak $r_1 = r_3$ M46

Description: The protection key for a given virtual address is obtained and placed in GR r_1.

When PSR.dt is 1, the DTLB and the VHPT are searched for the virtual address specified by GR r_3 and the region register indexed by GR r_3 bits {63:61}. If a matching present translation is found the protection key of the translation is placed in GR r_1. If a matching present translation is not found or if an unimplemented virtual address is specified by GR r_3, the value 1 is returned.

When PSR.dt is 0, tak searches the DTLB only, because the VHPT walker is disabled. If no matching present translation is found in the DTLB, the value 1 is returned.

A translation with the NaTPage attribute is not treated differently and returns its key field.

This instruction can only be executed at the most privileged level.

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS|TAK;
    check_target_register(r1);

    if (PSR.cpl != 0)
        privileged_operation_fault(itype);

    if (GR[r3].nat)
        register_nat_consumption_fault(itype);

    GR[r1] = tlb_access_key(GR[r3], itype);
    GR[r1].nat = 0;
}
```

Interruptions: Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault

Test Bit

Format: (*qp*) tbit.*trel*.*ctype* $p_1, p_2 = r_3, pos_6$ I16

Description: The bit specified by the pos_6 immediate is selected from GR r_3. The selected bit forms a single bit result either complemented or not depending on the *trel* completer. This result is written to the two predicate register destinations p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-14 on page 2-26.

The *trel* completer values .nz and .z indicate non-zero and zero sense of the test. For normal and unc types, only the .z value is directly implemented in hardware; the .nz value is actually a pseudo-op. For it, the assembler simply switches the predicate target specifiers and uses the implemented relation. For the parallel types, both relations are implemented in hardware.

Table 2-50. Test Bit Relations for Normal and unc tbits

trel	Test Relation	Pseudo-op of	
nz	selected bit == 1	z	$p_1 \leftrightarrow p_2$
z	selected bit == 0		

Table 2-51. Test Bit Relations for Parallel tbits

trel	Test Relation
nz	selected bit == 1
z	selected bit == 0

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is set, or if the compare type is unc.

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (trel == 'nz')                                   // 'nz' - test for 1
        tmp_rel = GR[r3]{pos6};
    else                                                // 'z' - test for 0
        tmp_rel = !GR[r3]{pos6};

    switch (ctype) {
        case 'and':                                     // and-type compare
            if (GR[r3].nat || !tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                                      // or-type compare
            if (!GR[r3].nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                                // or.andcm-type compare
            if (!GR[r3].nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
```

```
            }
            break;
        case 'unc':                                     // unc-type compare
        default:                                        // normal compare
            if (GR[r3].nat) {
                PR[p1] = 0;
                PR[p2] = 0;
            } else {
                PR[p1] = tmp_rel;
                PR[p2] = !tmp_rel;
            }
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Interruptions: Illegal Operation fault

Translation Hashed Entry Address

Format: (*qp*) thash $r_1 = r_3$ M46

Description: A Virtual Hashed Page Table (VHPT) entry address is generated based upon the specified virtual address and the result is placed in GR r_1. The virtual address is specified by GR r_3 and the region register selected by GR r_3 bits {63:61}.

If `thash` is given a NaT input argument or an unimplemented virtual address as an input, the resulting target register value is undefined, and its NaT bit is set to one.

When the processor is configured to use the region-based short format VHPT (PTA.vf=0), the value returned by `thash` is defined by the architected short format hash function. See "Region-based VHPT Short Format" on p. 4-16 in Volume 2.

When the processor is configured to use the long format VHPT (PTA.vf=1), `thash` performs an implementation-specific long format hash function on the virtual address to generate a hash index into the long format VHPT.

In the long format, a translation in the VHPT must be uniquely identified by its hash index generated by this instruction and the hash tag produced from the `ttag` instruction.

The hash function must use all implemented region bits and only virtual address bits {60:0} to determine the offset into the VHPT. Virtual address bits {63:61} are used only by the short format hash to determine the region of the VHPT.

This instruction must be implemented on all processor models, even processor models that do not implement a VHPT walker.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (GR[r3].nat || unimplemented_virtual_address(GR[r3])) {
        GR[r1] = undefined();
        GR[r1].nat = 1;
    } else {
        tmp_vr = GR[r3]{63:61};
        tmp_va = GR[r3]{60:0};
        GR[r1] = tlb_vhpt_hash(tmp_vr, tmp_va, RR[tmp_vr].rid,
                               RR[tmp_vr].ps);
        GR[r1].nat = 0;
    }
}
```

Interruptions: Illegal Operation fault

Test NaT

Format: (*qp*) tnat.*trel*.*ctype* $p_1, p_2 = r_3$ I17

Description: The NaT bit from GR r_3 forms a single bit result, either complemented or not depending on the *trel* completer. This result is written to the two predicate register destinations, p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-14 on page 2-26.

The *trel* completer values .nz and .z indicate non-zero and zero sense of the test. For normal and unc types, only the .z value is directly implemented in hardware; the .nz value is actually a pseudo-op. For it, the assembler simply switches the predicate target specifiers and uses the implemented relation. For the parallel types, both relations are implemented in hardware.

Table 2-52. Test NaT Relations for Normal and unc tnats

trel	Test Relation	Pseudo-op of	
nz	selected bit == 1	z	$p_1 \leftrightarrow p_2$
z	selected bit == 0		

Table 2-53. Test NaT Relations for Parallel tnats

trel	Test Relation
nz	selected bit == 1
z	selected bit == 0

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is set, or if the compare type is unc.

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (trel == 'nz')                                   // 'nz' - test for 1
        tmp_rel = GR[r3].nat;
    else                                                // 'z' - test for 0
        tmp_rel = !GR[r3].nat;

    switch (ctype) {
        case 'and':                                     // and-type compare
            if (!tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                                      // or-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                                // or.andcm-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
```

```
            break;
        case 'unc':                                     // unc-type compare
        default:                                        // normal compare
            PR[p1] = tmp_rel;
            PR[p2] = !tmp_rel;
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Interruptions: Illegal Operation fault

Translate to Physical Address

Format: (*qp*) tpa $r_1 = r_3$ M46

Description: The physical address for the virtual address specified by GR r_3 is obtained and placed in GR r_1.

When PSR.dt is 1, the DTLB and the VHPT are searched for the virtual address specified by GR r_3 and the region register indexed by GR r_3 bits {63:61}. If a matching present translation is found the physical address of the translation is placed in GR r_1. If a matching present translation is not found the appropriate TLB fault is taken.

When PSR.dt is 0, tak searches the DTLB only, because the VHPT walker is disabled. If no matching present translation is found in the DTLB, an Alternate Data TLB fault is raised.

If this instruction faults, then it will set the non-access bit in the ISR. The ISR read and write bits are not set.

This instruction can only be executed at the most privileged level.

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS|TPA;
    check_target_register(r1);

    if (PSR.cpl != 0)
        privileged_operation_fault(itype);

    if (GR[r3].nat)
        register_nat_consumption_fault(itype);

    GR[r1] = tlb_translate_nonaccess(GR[r3], itype);
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault

Translation Hashed Entry Tag

Format: (*qp*) ttag $r_1 = r_3$ M46

Description: A tag used for matching during searches of the long format Virtual Hashed Page Table (VHPT) is generated and placed in GR r_1. The virtual address is specified by GR r_3 and the region register selected by GR r_3 bits {63:61}.

If `ttag` is given a NaT input argument or an unimplemented virtual address as an input, the resulting target register value is undefined, and its NaT bit is set to one.

The tag generation function generates an implementation-specific long format VHPT tag. The tag generation function must use all implemented region bits and only virtual address bits {60:0}. PTA.vf is ignored by this instruction.

A translation in the long format VHPT must be uniquely identified by its hash index generated by the `thash` instruction and the tag produced from this instruction.

This instruction must be implemented on all processor models, even processor models that do not implement a VHPT walker.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (GR[r3].nat || unimplemented_virtual_address(GR[r3])) {
        GR[r1] = undefined();
        GR[r1].nat = 1;
    } else {
        tmp_vr = GR[r3]{63:61};
        tmp_va = GR[r3]{60:0};
        GR[r1] = tlb_vhpt_tag(tmp_va, RR[tmp_vr].rid, RR[tmp_vr].ps);
        GR[r1].nat = 0;
    }
}
```

Interruptions: Illegal Operation fault

Unpack

Format:

(*qp*) unpack1.h $r_1 = r_2, r_3$	one_byte_form, high_form	I2
(*qp*) unpack2.h $r_1 = r_2, r_3$	two_byte_form, high_form	I2
(*qp*) unpack4.h $r_1 = r_2, r_3$	four_byte_form, high_form	I2
(*qp*) unpack1.l $r_1 = r_2, r_3$	one_byte_form, low_form	I2
(*qp*) unpack2.l $r_1 = r_2, r_3$	two_byte_form, low_form	I2
(*qp*) unpack4.l $r_1 = r_2, r_3$	four_byte_form, low_form	I2

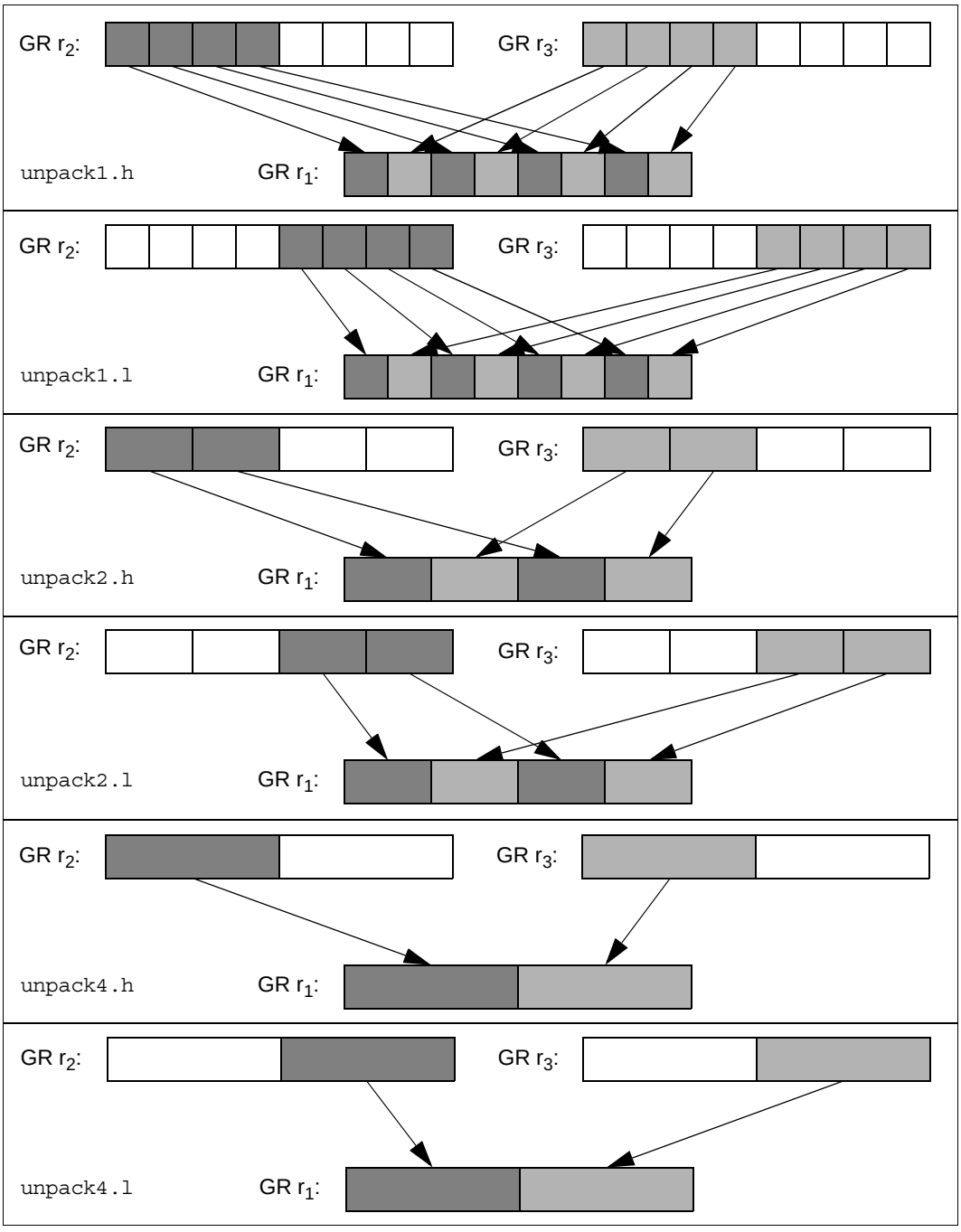
Description: The data elements of GR r_2 and r_3 are unpacked, and the result placed in GR r_1. In the high_form, the most significant elements of each source register are selected, while in the low_form the least significant elements of each source register are selected. Elements are selected alternately from the source registers.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (high_form)
            GR[r1] = concatenate8(  x[7], y[7], x[6], y[6],
                                    x[5], y[5], x[4], y[4]);
        else // low_form
            GR[r1] = concatenate8(  x[3], y[3], x[2], y[2],
                                    x[1], y[1], x[0], y[0]);
    } else if (two_byte_form) {                         // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (high_form)
            GR[r1] = concatenate4(x[3], y[3], x[2], y[2]);
        else // low_form
            GR[r1] = concatenate4(x[1], y[1], x[0], y[0]);
    } else {                                            // four-byte elements
        x[0] = GR[r2]{31:0};     y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};    y[1] = GR[r3]{63:32};

        if (high_form)
            GR[r1] = concatenate2(x[1], y[1]);
        else // low_form
            GR[r1] = concatenate2(x[0], y[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Figure 2-43. Unpack Operation



Exchange

Format: (*qp*) xchg*sz.ldhint* $r_1 = [r_3], r_2$ M16

Description: A value consisting of *sz* bytes is read from memory starting at the address specified by the value in GR r_3. The least significant *sz* bytes of the value in GR r_2 are written to memory starting at the address specified by the value in GR r_3. The value read from memory is then zero extended and placed in GR r_1 and the NaT bit corresponding to GR r_1 is cleared. The values of the *sz* completer are given in Table 2-54.

If the address specified by the value in GR r_3 is not naturally aligned to the size of the value being accessed in memory, an Unaligned Data Reference fault is taken independent of the state of the User Mask alignment checking bit, UM.ac (PSR.ac in the Processor Status Register).

Both read and write access privileges for the referenced page are required.

Table 2-54. Memory Exchange Size

sz Completer	Bytes Accessed
1	1 byte
2	2 bytes
4	4 bytes
8	8 bytes

The exchange is performed with acquire semantics, i.e. the memory read/write is made visible prior to all subsequent data memory accesses. See Volume 1 and Volume 2 for details on memory ordering.

The memory read and write are guaranteed to be atomic.

This instruction is only supported to cacheable pages with write-back write policy. Accesses to NaTPages cause a Data NaT Page Consumption fault. Accesses to pages with other memory attributes cause an Unsupported Data Reference fault.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 2-32 on page 2-127. Locality hints do not affect program functionality and may be ignored by the implementation.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(SEMAPHORE);

    paddr = tlb_translate(GR[r3], sz, SEMAPHORE, PSR.cpl, &mattr,
                          &tmp_unused);

    if (!ma_supports_semaphores(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

    val = mem_xchg(GR[r2], paddr, sz, UM.be, mattr, ACQUIRE, ldhint);

    alat_inval_multiple_entries(paddr, sz);

    GR[r1] = zero_ext(val, sz * 8);
    GR[r1].nat = 0;
}
```

Interruptions: Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

Fixed-Point Multiply Add

Format:

(*qp*) xma.l $f_1 = f_3, f_4, f_2$	low_form	F2
(*qp*) xma.lu $f_1 = f_3, f_4, f_2$	pseudo-op of: (*qp*) xma.l $f_1 = f_3, f_4, f_2$	
(*qp*) xma.h $f_1 = f_3, f_4, f_2$	high_form	F2
(*qp*) xma.hu $f_1 = f_3, f_4, f_2$	high_unsigned_form	F2

Description: Two source operands (FR f_3 and FR f_4) are treated as either signed or unsigned integers and multiplied. The third source operand (FR f_2) is zero extended and added to the product. The upper or lower 64 bits of the resultant sum are selected and placed in FR f_1.

In the high_unsigned_form, the significand fields of FR f_3 and FR f_4 are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The significand field of FR f_2 is zero extended and added to the product. The most significant 64-bits of the resultant sum are placed in the significand field of FR f_1.

In the high_form, the significand fields of FR f_3 and FR f_4 are treated as signed integers and multiplied to produce a full 128-bit signed result. The significand field of FR f_2 is zero extended and added to the product. The most significant 64-bits of the resultant sum are placed in the significand field of FR f_1.

In the other forms, the significand fields of FR f_3 and FR f_4 are treated as signed integers and multiplied to produce a full 128-bit signed result. The significand field of FR f_2 is zero extended and added to the product. The least significant 64-bits of the resultant sum are placed in the significand field of FR f_1.

In all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

Note: f1 as an operand is not an integer 1; it is just the register file format's 1.0 value.

In all forms, if any of FR f_3 , FR f_4 , or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
    } else {
        if (low_form || high_form)
            tmp_res_128 =
                fp_I64_x_I64_to_I128(FR[f3].significand, FR[f4].significand);
        else // high_unsigned_form
            tmp_res_128 =
                fp_U64_x_U64_to_U128(FR[f3].significand, FR[f4].significand);

        tmp_res_128 =
            fp_U128_add(tmp_res_128, fp_U64_to_U128(FR[f2].significand));

        if (high_form || high_unsigned_form)
            FR[f1].significand = tmp_res_128.hi;
        else // low_form
            FR[f1].significand = tmp_res_128.lo;

        FR[f1].exponent = FP_INTEGER_EXP;
```

```
            FR[f1].sign = FP_SIGN_POSITIVE;
        }

        fp_update_psr(f1);
    }
```

Interruptions: Disabled Floating-point Register fault

Fixed-Point Multiply

Format:

(*qp*) xmpy.l $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.l $f_1 = f_3, f_4$, f0
(*qp*) xmpy.lu $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.l $f_1 = f_3, f_4$, f0
(*qp*) xmpy.h $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.h $f_1 = f_3, f_4$, f0
(*qp*) xmpy.hu $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.hu $f_1 = f_3, f_4$, f0

Description: Two source operands (FR f_3 and FR f_4) are treated as either signed or unsigned integers and multiplied. The upper or lower 64 bits of the resultant product are selected and placed in FR f_1.

In the high_unsigned_form, the significand fields of FR f_3 and FR f_4 are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The most significant 64-bits of the resultant product are placed in the significand field of FR f_1.

In the high_form, the significand fields of FR f_3 and FR f_4 are treated as signed integers and multiplied to produce a full 128-bit signed result. The most significant 64-bits of the resultant product are placed in the significand field of FR f_1.

In the other forms, the significand fields of FR f_3 and FR f_4 are treated as signed integers and multiplied to produce a full 128-bit signed result. The least significant 64-bits of the resultant product are placed in the significand field of FR f_1.

In all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

Note: f1 as an operand is not an integer 1; it is just the register file format's 1.0 value.

Operation: See "Fixed-Point Multiply Add" on p. 2-233.

Exclusive Or

Format:

(*qp*) xor $r_1 = r_2, r_3$	register_form	A1
(*qp*) xor $r_1 = imm_8, r_3$	imm8_form	A3

Description: The two source operands are logically XORed and the result placed in GR r_1. In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the imm_8 encoding field.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src ^ GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

Zero Extend

Format: (*qp*) zxt*xsz* $r_1 = r_3$ I29

Description: The value in GR r_3 is zero extended above the bit position specified by *xsz* and the result is placed in GR r_1. The mnemonic values for *xsz* are given in Table 2-49 on page 2-219.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = zero_ext(GR[r3],xsz * 8);
    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

IA-64 Pseudo-Code Functions 3

This chapter contains a table of all pseudo-code functions used on the IA-64 instruction pages.

Table 3-1. Pseudo-Code Functions (Sheet 1 of 11)

Function	Operation
*xxx*_fault(parameters ...)	There are several fault functions. Each fault function accepts parameters specific to the fault, e.g. exception code values, virtual addresses, etc. If the fault is deferred for speculative load exceptions the fault function will return with a deferral indication. Otherwise, fault routines do not return and terminate the instruction sequence.
*xxx*_trap(parameters ...)	There are several trap functions. Each trap function accepts parameters specific to the trap, e.g. trap code values, virtual addresses, etc. Trap routines do not return.
acceptance_fence()	Ensures prior data memory references to uncached ordered-sequential memory pages are “accepted”, before subsequent data memory references are performed by the processor.
alat_cmp(rtype, raddr)	Returns a one if the implementation finds an ALAT entry which matches the register type specified by `rtype` and the register address specified by `raddr`, else returns zero. This function is implementation specific. Note that an implementation may optionally choose to return zero (indicating no match) even if a matching entry exists in the ALAT. This provides implementation flexibility in designing fast ALAT lookup circuits.
alat_frame_update(delta_bof, delta_sof)	Notifies the ALAT of a change in the bottom of frame and/or size of frame. This allows management of the ALAT’s tag bits or other management functions it might need.
alat_inval()	Invalidate all entries in the ALAT.
alat_inval_multiple_entries(paddr , size)	The ALAT is queried using the physical memory address specified by `paddr` and the access size specified by `size`. All matching ALAT entries are invalidated. No value is returned.
alat_inval_single_entry(rtype, rega)	The ALAT is queried using the register type specified by `rtype` and the register address specified by `rega`. At most one matching ALAT entry is invalidated. No value is returned.
alat_write(rtype, raddr, paddr, size)	Allocates a new ALAT entry using the register type specified by `rtype`, the register address specified by `raddr`, the physical memory address specified by `paddr`, and the access size specified by `size`. No value is returned. This function guarantees that only one ALAT entry exists for a given `raddr`. If a `ld.c.nc`, `ldf.c.nc`, or `ldfp.c.nc` instruction’s `raddr` matches an existing ALAT entry’s register tag, but the instruction’s `size` and/or `paddr` are different than that of the existing entry’s; then this function may either preserve the existing entry, or invalidate it and write a new entry with the instruction’s specified `size` and `paddr`.

Table 3-1. Pseudo-Code Functions (Sheet 2 of 11)

Function	Operation
align_to_size_boundary(vaddr, size)	Returns `vaddr` aligned to the boundary specified by `size`.
branch_predict(wh, ih, ret, target, tag)	Implementation-dependent routine which updates the processor's branch prediction structures.
check_branch_implemented(check_type)	Implementation-dependent routine which returns TRUE or FALSE, depending on whether a failing check instruction causes a branch (TRUE), or a Speculative Operation fault (FALSE). The result may be different for different types of check instructions: `CHKS_GENERAL, CHKS_FLOAT, CHKA_GENERAL, CHKA_FLOAT.`
check_target_register(r1)	If r1 targets an out-of-frame stacked register (as defined by CFM), an illegal operation fault is delivered, and this function does not return.
check_target_register_sof(r1, newsof)	If r1 targets an out-of-frame stacked register (as defined by the newsof parameter), an illegal operation fault is delivered, and this function does not return.
concatenate2(x1, x2)	Concatenates the lower 32 bits of the 2 arguments, and returns the 64-bit result.
concatenate4(x1, x2, x3, x4)	Concatenates the lower 16 bits of the 4 arguments, and returns the 64-bit result.
concatenate8(x1, x2, x3, x4, x5, x6, x7, x8)	Concatenates the lower 8 bits of the 8 arguments, and returns the 64-bit result.
data_serialize()	Ensures all prior register updates with side-effects are observed before subsequent execution and data memory references are performed.
deliver_unmasked_pending_interrupt()	This implementation-specific function checks whether any unmasked external interrupts are pending, and if so, transfers control to the external interrupt vector.
fadd(fp_dp, fr2)	Adds a floating-point register value to the infinitely precise product and return the infinitely precise sum, ready for rounding.
fcmp_exception_fault_check(f2, f3, frel, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fcmp` instruction.
fcvt_fx_exception_fault_check(fr2, signed_form, trunc_form, sf *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fcvt.fx`, `fcvt.fxu`, `fcvt.fx.trunc` and `fcvt.fxu.trunc` instructions. It propagates NaNs.
fma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fma` instruction. It propagates NaNs and special IEEE results.
fminmax_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `famax`, `famin`, `fmax`, and `fmin` instructions.
fms_fnma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fms` and `fnma` instructions. It propagates NaNs and special IEEE results.
fmul(fr3, fr4)	Performs an infinitely precise multiply of two floating-point register values.
followed_by_stop()	Returns TRUE if the current instruction is followed by a stop; otherwise, returns FALSE.

Table 3-1. Pseudo-Code Functions (Sheet 3 of 11)

Function	Operation
fp_check_target_register(f1)	If the specified floating-point register identifier is 0 or 1, this function causes an illegal operation fault.
fp_decode_fault(tmp_fp_env)	Returns floating-point exception fault code values for ISR.code.
fp_decode_traps(tmp_fp_env)	Returns floating-point trap code values for ISR.code.
fp_is_nan_or_inf(freg)	Returns true if the floating-point exception_fault_check functions returned a IEEE fault disabled default result or a propagated NaN.
fp_equal(fr1, fr2)	IEEE standard equality relationship test.
fp_ieee_recip(num, den)	Returns the true quotient for special sets of operands, or an approximation to the reciprocal of the divisor to be used in the software divide algorithm.
fp_ieee_recip_sqrt(root)	Returns the true square root result for special operands, or an approximation to the reciprocal square root to be used in the software square root algorithm.
fp_is_nan(freg)	Returns true when floating register contains a NaN.
fp_is_natval(freg)	Returns true when floating register contains a NaTVal.
fp_is_normal(freg)	Returns true when floating register contains a normal number.
fp_is_pos_inf(freg)	Returns true when floating register contains a positive infinity.
fp_is_qnan(freg)	Returns true when floating register contains a quiet NaN.
fp_is_snan(freg)	Returns true when floating register contains a signalling NaN.
fp_is_unorm(freg)	Returns true when floating register contains an unnormalized number.
fp_is_unsupported(freg)	Returns true when floating register contains an unsupported format.
fp_less_than(fr1, fr2)	IEEE standard less-than relationship test.
fp_lesser_or_equal(fr1, fr2)	IEEE standard less-than or equal-to relationship test.
fp_normalize(fr1)	Normalizes an unnormalized fp value. This function flushes to zero any unnormal values which can not be represented in the register file.
fp_raise_fault(tmp_fp_env)	Checks the local instruction state for any faulting conditions which require an interruption to be raised.
fp_raise_traps(tmp_fp_env)	Checks the local instruction state for any trapping conditions which require an interruption to be raised.
fp_reg_bank_conflict(f1, f2)	Returns true if the two specified FRs are in the same bank.
fp_reg_disabled(f1, f2, f3, f4)	Check for possible disabled floating-point register faults.
fp_reg_read(freg)	Reads the FR and gives canonical double-extended denormals (and pseudo-denormals) their true mathematical exponent. Other classes of operands are unaltered.
fp_unordered(fr1, fr2)	IEEE standard unordered relationship.
fp_fr_to_mem_format(freg, size)	Converts a floating-point value in register format to floating-point memory format. It assumes that the floating-point value in the register has been previously rounded to the correct precision which corresponds with the `size` parameter.

Table 3-1. Pseudo-Code Functions (Sheet 4 of 11)

Function	Operation
fpcmp_exception_fault_check(f2, f3, frel, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpcmp` instruction.
fpcvt_exception_fault_check(f2, signed_form, trunc_form, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpcvt.fx`, `fpcvt.fxu`, `fpcvt.fx.trunc`, and `fpcvt.fxu.trunc` instructions. It propagates NaNs.
fpma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpma` instruction. It propagates NaNs and special IEEE results.
fpminmax_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpmin`, `fpmax`, `fpamin` and `fpamax` instructions.
fpms_fpnma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpms` and `fpnma` instructions. It propagates NaNs and special IEEE results.
fprcpa_exception_fault_check(f2, f3, sf, *tmp_fp_env, *limits_check)	Checks for all floating-point faulting conditions for the `fprcpa` instruction. It propagates NaNs and special IEEE results. It also indicates operand limit violations.
fprsqrta_exception_fault_check(f3, sf, *tmp_fp_env, *limits_check)	Checks for all floating-point faulting conditions for the `fprsqrta` instruction. It propagates NaNs and special IEEE results. It also indicates operand limit violations.
frcpa_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `frcpa` instruction. It propagates NaNs and special IEEE results.
frsqrta_exception_fault_check(f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `frsqrta` instruction. It propagates NaNs and special IEEE results
ignored_field_mask(regclass, reg, value)	Boolean function that returns value with bits cleared to 0 corresponding to ignored bits for the specified register and register type.
instruction_serialize()	Ensures all prior register updates with side-effects are observed before subsequent instruction and data memory references are performed. Also ensures prior `sync.i` operations have been observed by the instruction cache.
instruction_synchronize	Synchronizes the instruction and data stream for Flush Cache operations. This function ensures that when prior `fc` operations are observed by the local data cache they are observed by the local instruction cache, and when prior `fc` operations are observed by another processor's data cache they are observed within the same processor's instruction cache.
is_finite(freg)	Returns true when floating register contains a finite number.
is_ignored_reg(regnum)	Boolean function that returns true if `regnum` is an ignored application register, otherwise false.
is_inf(freg)	Returns true when floating register contains an infinite number.
is_interruption_cr(regnum)	Boolean function that returns true if `regnum` is one of the Interruption Control registers (see "Interruption Control Registers" on page 3-18 in Volume 2), otherwise false.
is_kernel_reg(ar_addr)	Returns a one if `ar_addr` is the address of a kernel register application register.

Table 3-1. Pseudo-Code Functions (Sheet 5 of 11)

Function	Operation
is_read_only_reg(rtype, raddr)	Returns a one if the register addressed by `raddr` in the register bank of type `rtype` is a read only register.
is_reserved_field(regclass, arg2, arg3)	Returns true if the specified data would write a one in a reserved field.
is_reserved_reg(regclass, regnum)	Returns true if register `regnum` is reserved in the `regclass` register file.
long_branch_implemented()	Implementation-dependent routine which returns TRUE or FALSE, depending on whether long branches are implemented.
mem_flush(paddr)	The line addressed by the physical address `paddr` is invalidated in all levels of the memory hierarchy above memory and written back to memory if it is inconsistent with memory.
mem_flush_pending_stores()	The processor is instructed to start draining pending stores in write coalescing and write buffers. This operation is a “hint”. There is no indication when prior stores have actually been drained.
mem_implicit_prefetch(vaddr, hint, type)	Moves the line addressed by `vaddr` to the location of the memory hierarchy specified by `hint`. This function is implementation dependent and can be ignored. The `type` allows the implementation to distinguish prefetches for different instruction types.
mem_promote(paddr, mtype, hint)	Moves the line addressed by `paddr` to the highest level of the memory hierarchy conditioned by the access hints specified by `hint`. Implementation dependent and can be ignored.
mem_read(paddr, size, border, mattr, otype, hint)	Returns the `size` bytes starting at the physical memory location specified by `paddr` with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or ACQUIRE.
fp_mem_to_fr_format(mem, size)	Converts a floating-point value in memory format to floating-point register format.
mem_write(value, paddr, size, border, mattr, otype, hint)	Writes the least significant `size` bytes of `value` into memory starting at the physical memory address specified by `paddr` with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or RELEASE. No value is returned.
mem_xchg(data, paddr, size, byte_order, mattr, otype, hint)	Returns size bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. After the read, the least significant `size` bytes of data are written to `size` bytes in memory starting at the physical address specified by `paddr`. The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and must be ACQUIRE.

Table 3-1. Pseudo-Code Functions (Sheet 6 of 11)

Function	Operation
mem_xchg_add(add_val, paddr, size, byte_order, mattr, otype, hint)	Returns `size` bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. The least significant `size` bytes of the sum of the value read from memory and `add_val` is then written to `size` bytes in memory starting at the physical address specified by `paddr`. The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
mem_xchg_cond(cmp_val, data, paddr, size, byte_order, mattr, otype, hint)	Returns `size` bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. If the value read from memory is equal to `cmp_val,` then the least significant `size` bytes of data are written to `size bytes` in memory starting at the physical address specified by `paddr`. If the write is performed, the read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
ordering_fence()	Ensures prior data memory references are made visible before future data memory references are made visible by the processor.
pr_phys_to_virt(phys_id)	Returns the virtual register id of the predicate from the physical register id, `phys_id` of the predicate.
rotate_regs()	Decrements the Register Rename Base registers, effectively rotating the register files. CFM.rrb.gr is decremented only if CFM.sor is non-zero.
rse_enable_current_frame_load( )	If the RSE load pointer (RSE.BSPLoad) is greater than AR[BSP], the `RSE.CFLE` bit is set to indicate that mandatory RSE loads are allowed to restore registers in the current frame (in no other case does the RSE spill or fill registers in the current frame). This function does not perform mandatory RSE loads. This procedure does not cause any interruptions.
rse_ensure_regs_loaded(number_of_bytes)	All registers and NaT collections between `AR[BSP]` and `(AR[BSP]-number_of_bytes)` which are not already in stacked registers are loaded into the register stack with mandatory RSE loads. If the number of registers to be loaded is greater than `RSE.N_STACK_PHYS` an Illegal Operation fault is raised. All registers starting with backing store address (AR[BSP] - 8) and decrementing down to and including backing store address (AR[BSP] - number_of_bytes) are made part of the dirty partition. With exception of the current frame, all other stacked registers are made part of the invalid partition. Note that `number_of_bytes` may be zero. The resulting sequence of RSE loads may be interrupted. Mandatory RSE loads may cause an interruption; see Table 6-6 on page 6-12 in Volume 2.

Table 3-1. Pseudo-Code Functions (Sheet 7 of 11)

Function	Operation
rse_invalidate_non_current_regs ()	All registers outside the current frame are invalidated.
rse_load(type)	Restores a register or NaT collection from the backing store (load_address = RSE.BspLoad - 8). If load_address{8:3} is equal to 0x3f then a NaT collection is loaded into a NaT dispersal register. (dispersal register may not be the same as AR[RNAT].) If load_address{8:3} is not equal to 0x3f then the register RSE.LoadReg - 1 is loaded and the NaT bit for that register is set to dispersal_register{load_address{8:3}}. If the load is successful RSE.BspLoad is decremented by 8. If the load is successful and a register was loaded RSE.LoadReg is decremented by 1 (possibly wrapping in the stacked registers). The load moves a register from the invalid partition to the current frame if RSE.CFLE is 1, or to the clean partition if RSE.CFLE is 0. For mandatory RSE loads, type is MANDATORY. Mandatory RSE loads may cause interruptions. See Table 6-6 on page 6-12 in Volume 2.
rse_new_frame(current_frame_size, new_frame_size)	A new frame is defined without changing any register renaming. The new frame size is completely defined by the new_frame_size parameter (successive calls are not cumulative). If new_frame_size is larger than current_frame_size and the number of registers in the invalid and clean partitions is less than the size of frame growth then mandatory RSE stores are issued until enough registers are available. The resulting sequence of RSE stores may be interrupted. Mandatory RSE stores may cause interruptions; see Table 6-6 on page 6-12 in Volume 2.
rse_preserve_frame(preserved_frame_size)	The number of registers specified by preserved_frame_size are marked to be preserved by the RSE. Register renaming causes the preserved_frame_size registers after GR[32] to be renamed to GR[32]. AR[BSP] is updated to contain the backing store address where the new GR[32] will be stored.
rse_restore_frame(preserved_sol, growth, current_frame_size)	The first two parameters define how the current frame is about to be updated by a branch return or rfi: preserved_sol defines how many registers need to be restored below RSE.BOF; growth defines by how many registers the top of the current frame will grow (growth will generally be negative). The number of registers specified by preserved_sol are marked to be restored. Register renaming causes the preserved_sol registers before GR[32] to be renamed to GR[32]. AR[BSP] is updated to contain the backing store address where the new GR[32] will be stored. If the number of dirty and clean registers is less than preserved_sol then mandatory RSE loads must be issued before the new current frame is considered valid. This function does not perform mandatory RSE loads. This function returns TRUE if the preserved frame grows beyond the invalid and clean regions into the dirty region. In this case the third argument, current_frame_size, is used to force the returned to frame to zero (see "Bad PFS used by Branch Return" on page 6-11 in Volume 2).

Table 3-1. Pseudo-Code Functions (Sheet 8 of 11)

Function	Operation
rse_store(type)	Saves a register or NaT collection to the backing store (store_address = AR[BSPSTORE]). If store_address{8:3} is equal to 0x3f then the NaT collection AR[RNAT] is stored. If store_address{8:3} is not equal to 0x3f then the register RSE.StoreReg is stored and the NaT bit from that register is deposited in AR[RNAT]{store_address{8:3}}. If the store is successful AR[BSPSTORE] is incremented by 8. If the store is successful and a register was stored RSE.StoreReg is incremented by 1 (possibly wrapping in the stacked registers). This store moves a register from the dirty partition to the clean partition. For mandatory RSE stores, `type` is MANDATORY. Mandatory RSE stores may cause interruptions. See Table 6-6 on page 6-12 in Volume 2.
rse_update_internal_stack_pointers(new_store_pointer)	Given a new value for `AR[BSPSTORE]` (`new_store_pointer`) this function computes the new value for `AR[BSP]`. This value is equal to `new_store_pointer` plus the number of dirty registers plus the number of intervening NaT collections. This means that the size of the dirty partition is the same before and after a write to `AR[BSPSTORE]`. All clean registers are moved to the invalid partition.
sign_ext(value, pos)	Returns a 64 bit number with bits `pos`-1 through 0 taken from `value` and bit `pos`-1 of `value` replicated in bit positions `pos` through 63. If `pos` is greater than or equal to 64, `value` is returned.
tlb_access_key(vaddr)	This function returns the access key from the TLB for the entry corresponding to `vaddr`.
tlb_broadcast_purge(rid, vaddr, size, type)	Sends a broadcast purge DTC and ITC transaction to other processors in the multi-processor coherency domain, where the region identifier (`rid`), virtual address (`vaddr`) and page size (`size`) specify the translation entry to purge. The operation waits until all processors that receive the purge have completed the purge operation. The purge type (`type`) specifies whether the ALAT on other processors should also be purged in conjunction with the TC.
tlb_enter_privileged_code()	This function determines the new privilege level for `epc` from the TLB entry for the page containing this instruction. If the page containing the epc instruction has execute-only page access rights and the privilege level assigned to the page is higher than (numerically less than) the current privilege level, then the current privilege level is set to the privilege level field in the translation for the page containing the `epc` instruction.
tlb_grant_permission(vaddr, type, pl)	Returns a boolean indicating if read, write access is granted for the specified virtual memory address (`vaddr`) and privilege level (`pl`). The access type (`type`) specifies either read or write. The following faults are checked; VHPT Translation, TLB Miss, Nested TLB, Page Not Present, NaT Page Consumption, and Key Miss. If a fault is generated, this function does not return.

Table 3-1. Pseudo-Code Functions (Sheet 9 of 11)

Function	Operation
tlb_insert_data(slot, pte0, pte1, vaddr, rid, tr)	Inserts an entry into the DTLB, at the specified `slot` number. `pte0`, `pte1` compose the translation. `vaddr` and `rid` specify the virtual address and region identifier for the translation. If `tr` is true the entry is placed in the TR section, otherwise the TC section.
tlb_insert_inst(slot, pte0, pte1, vaddr, rid, tr)	Inserts an entry into the ITLB, at the specified `slot` number. `pte0`, `pte1` compose the translation. `vaddr` and `rid` specify the virtual address and region identifier for the translation. If `tr` is true, the entry is placed in the TR section, otherwise the TC section.
tlb_may_purge_dtc_entries(rid, vaddr, size)	May locally purge DTC entries that match the specified virtual address (`vaddr`), region identifier (`rid`) and page size (`size`). May also invalidate entries that partially overlap the parameters. The extent of purging is implementation dependent. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_may_purge_itc_entries(rid, vaddr, size)	May locally purge ITC entries that match the specified virtual address (`vaddr`), region identifier (`rid`) and page size (`size`). May also invalidate entries that partially overlap the parameters. The extent of purging is implementation dependent. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_must_purge_dtc_entries(rid, vaddr, size)	Purges all local, possibly overlapping, DTC entries matching the specified region identifier (`rid`), virtual address (`vaddr`) and page size (`size`). `vaddr{63:61}` (VRN) is ignored in the purge, i.e all entries that match `vaddr`{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache. If the specified purge values overlap with an existing DTR translation, an implementation may generate a machine check abort.
tlb_must_purge_itc_entries(rid, vaddr, size)	Purges all local, possibly overlapping, ITC entry matching the specified region identifier (`rid`), virtual address (`vaddr`) and page size (`size`). `vaddr`{63:61} (VRN) is ignored in the purge, i.e all entries that match `vaddr`{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache. If the specified purge values overlap with an existing ITR translation, an implementation may generate a machine check abort.

Table 3-1. Pseudo-Code Functions (Sheet 10 of 11)

Function	Operation
tlb_must_purge_dtr_entries(rid, vaddr, size)	Purges all local, possibly overlapping, DTR entries matching the specified region identifier (`rid`), virtual address (`vaddr`) and page size (`size`). `vaddr{63:61}` (VRN) is ignored in the purge, i.e all entries that match `vaddr`{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_must_purge_itr_entries(rid, vaddr, size)	Purges all local, possibly overlapping, ITR entry matching the specified region identifier (`rid`), virtual address (`vaddr`) and page size (`size`). `vaddr`{63:61} (VRN) is ignored in the purge, i.e all entries that match `vaddr`{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_purge_translation_cache(loop)	Removes 1 to N translations from the local processor's ITC and DTC. The number of entries removed is implementation specific. The parameter `loop` is used to generate an implementation specific purge parameter.
tlb_replacement_algorithm(tlb)	Returns the next ITC or DTC slot number to replace. Replacement algorithms are implementation specific. `tlb` specifies to perform the algorithm on the ITC or DTC.
tlb_search_pkr(key)	Searches for a valid protection key register with a matching protection `key`. The search algorithm is implementation specific. Returns the PKR register slot number if found, otherwise returns Not Found.
tlb_translate(vaddr, size, type, cpl, *attr, *defer)	Returns the translated data physical address for the specified virtual memory address (`vaddr`) when translation enabled; otherwise, returns `vaddr`. `size` specifies the size of the access, `type` specifies the type of access (e.g. read, write, advance, spec). `cpl` specifies the privilege level for access checking purposes. `*attr` returns the mapped physical memory attribute. If any fault conditions are detected and deferred, tlb_translate returns with `*defer` set. If a fault is generated but the fault is not deferred, tlb_translate does not return. tlb_translate checks the following faults: • VHPT Data fault Data Nested TLB fault Data TLB fault Alternate Data TLB fault Data page not present fault Data NaT page consumption fault Data key miss fault Data key permission fault Data access rights fault Data dirty bit fault Data access bit fault Data debug fault

Table 3-1. Pseudo-Code Functions (Sheet 11 of 11)

Function	Operation
tlb_translate_nonaccess(vaddr, type)	Returns the translated data physical address for the specified virtual memory address (`vaddr`). `type` specifies the type of access (e.g. `FC`, `TPA`). If a fault is generated, tlb_translate_nonaccess does not return. The following faults are checked: • VHPT data fault Data TLB fault Data Nested TLB fault Data page not present fault Data NaT page consumption fault
tlb_vhpt_hash(vrn, vaddr61, rid, size)	Generates a VHPT entry address for the specified virtual region number (`vrn`) and 61-bit virtual offset (`vaddr61`), region identifier (`rid`) and page size (`size`). Tlb_vhpt_hash hashes `vaddr`, `rid` and `size` parameters to produce a hash index. The hash index is then masked based on PTA.size and concatenated with PTA.base to generate the VHPT entry address. The long format hash is implementation specific.
tlb_vhpt_tag(vaddr, rid, size)	Generates a VHPT tag identifier for the specified virtual address (`vaddr`), region identifier (`rid`) and page size (`size`). Tlb_vhpt_tag hashes the `vaddr`, rid and `size` parameters to produce translation identifier. The tag in conjunction with the hash index is used to uniquely identify translations in the VHPT. Tag generation is implementation specific. All processor models tag function must guarantee that bit 63 of the generated tag is zero (ti bit).
unimplemented_physical_address(paddr)	Return TRUE if the presented physical address is unimplemented on this processor model; FALSE otherwise. This function is model-specific.
undefined()	Returns an undefined 64-bit value.
undefined_behavior()	Causes undefined processor behavior.
unimplemented_virtual_address(vaddr)	Return TRUE if the presented virtual address is unimplemented on this processor model; FALSE otherwise. This function is model-specific.
fp_update_fpsr(sf, tmp_fp_env)	Copies a floating-point instruction's local state into the global FPSR.
fp_update_psr(dest_freg)	Conditionally sets PSR.mfl or PSR.mfh based on dest_freg.
zero_ext(value, pos)	Returns a 64 bit unsigned number with bits `pos`-1 through 0 taken from `value` and zeroes in bit positions `pos` through 63. If `pos` is greater than or equal to 64, `value` is returned.

IA-64 Instruction Formats 4

Each IA-64 instruction is categorized into one of six types; each instruction type may be executed on one or more execution unit types. Table 4-1 lists the instruction types and the execution unit type on which they are executed.

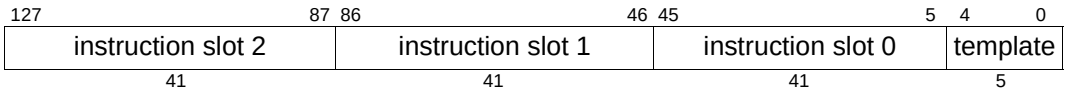
Table 4-1. Relationship between Instruction Type and Execution Unit Type

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit/B-unit[a]

a. L+X Major Opcodes 0 - 7 execute on an I-unit. L+X Major Opcodes 8 - F execute on a B-unit.

Three instructions are grouped together into 128-bit sized and aligned containers called **bundles**. Each bundle contains three 41-bit **instruction slots** and a 5-bit template field. The format of a bundle is depicted in Figure 4-1.

Figure 4-1. Bundle Format



The template field specifies two properties: stops within the current bundle and the mapping of instruction slots to execution unit types. Not all combinations of these two properties are allowed — Table 4-2 indicates the defined combinations. The three rightmost columns correspond to the three instruction slots in a bundle; listed within each column is the execution unit type controlled by that instruction slot for each encoding of the template field. A double line to the right of an instruction slot indicates that a stop occurs at that point within the current bundle. See "Instruction Encoding Overview" on page 3-15 in Volume 1 for the definition of a stop. Within a bundle, execution order proceeds from slot 0 to slot 2. Unused template values (appearing as empty rows in Table 4-2) are reserved and cause an Illegal Operation fault.

Extended instructions, used for long immediate integer and long branch instructions, occupy two instruction slots. Depending on the major opcode, extended instructions execute on a B-unit (long branch/call) or an I-unit (all other L+X instructions).

Table 4-2. Template Field Encoding and Instruction Slot Mapping

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	L-unit	X-unit[a]
05	M-unit	L-unit	X-unit[a]
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

a. The MLX template was formerly called MLI, and for compatibility, the X slot may encode `break.i` and `nop.i` in addition to any X-unit instruction.

4.1 Format Summary

All instructions in the instruction set are 41 bits in length. The leftmost 4 bits (40:37) of each instruction are the major opcode. Table 4-3 shows the major opcode assignments for each of the five instruction types — ALU (A), Integer (I), Memory (M), Floating-point (F), and Branch (B). Bundle template bits are used to distinguish among the four columns, so the same major op values can be reused in each column.

Unused major ops (appearing as blank entries in Table 4-3) behave in one of four ways:

- Ignored major ops (white entries in Table 4-3) execute as `nop` instructions.
- Reserved major ops (light gray in the gray scale version of Table 4-3) cause an Illegal Operation fault.
- Reserved if PR[qp] is 1 major ops (dark gray in the gray scale version of Table 4-3) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0.
- Reserved if PR[qp] is 1 B-unit major ops (medium gray in the gray scale version of Table 4-3) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0. These differ from the Reserved if PR[qp] is 1 major ops only in their RAW dependency behavior (see "RAW Dependency Table" on page A-4 in Volume 2).

Table 4-3. Major Opcode Assignments

Major Op (Bits 40:37)	Instruction Type				
	I/A	M/A	F	B	L+X
0	Misc 0	Sys/Mem Mgmt 0	FP Misc 0	Misc/Indirect Branch 0	Misc 0
1	1	Sys/Mem Mgmt 1	FP Misc 1	Indirect Call 1	1
2	2	2	2	Indirect Predict/Nop 2	2
3	3	3	3	3	3
4	Deposit 4	Int Ld +Reg/getf 4	FP Compare 4	IP-relative Branch 4	4
5	Shift/Test Bit 5	Int Ld/St +Imm 5	FP Class 5	IP-rel Call 5	5
6	6	FP Ld/St +Reg/setf 6	6	6	movl 6
7	MM Mpy/Shift 7	FP Ld/St +Imm 7	7	IP-relative Predict 7	7
8	ALU/MM ALU 8	ALU/MM ALU 8	fma 8	e 8	8
9	Add Imm_{22} 9	Add Imm_{22} 9	fma 9	e 9	9
A	A	A	fms A	e A	A
B	B	B	fms B	e B	B
C	Compare C	Compare C	fnma C	e C	Long Branch C
D	Compare D	Compare D	fnma D	e D	Long Call D
E	Compare E	Compare E	fselect/xma E	e E	E
F	F	F	F	e F	F

Table 4-4 summarizes all the instruction formats. The instruction fields are color-coded for ease of identification, as described in Table 4-5 on page 4-7.

The instruction field names, used throughout this chapter, are described in Table 4-6 on page 4-8. The set of special notations (such as whether an instruction is privileged) are listed in Table 4-7 on page 4-9. These notations appear in the “Instruction” column of the opcode tables.

Most instruction containing immediates encode those immediates in more than one instruction field. For example, the 14-bit immediate in the Add Imm_{14} instruction (format A4) is formed from the imm_{7b}, imm_{6d}, and s fields. Table 4-70 on page 4-84 shows how the immediates are formed from the instruction fields for each instruction which has an immediate.

Table 4-4. Instruction Format Summary

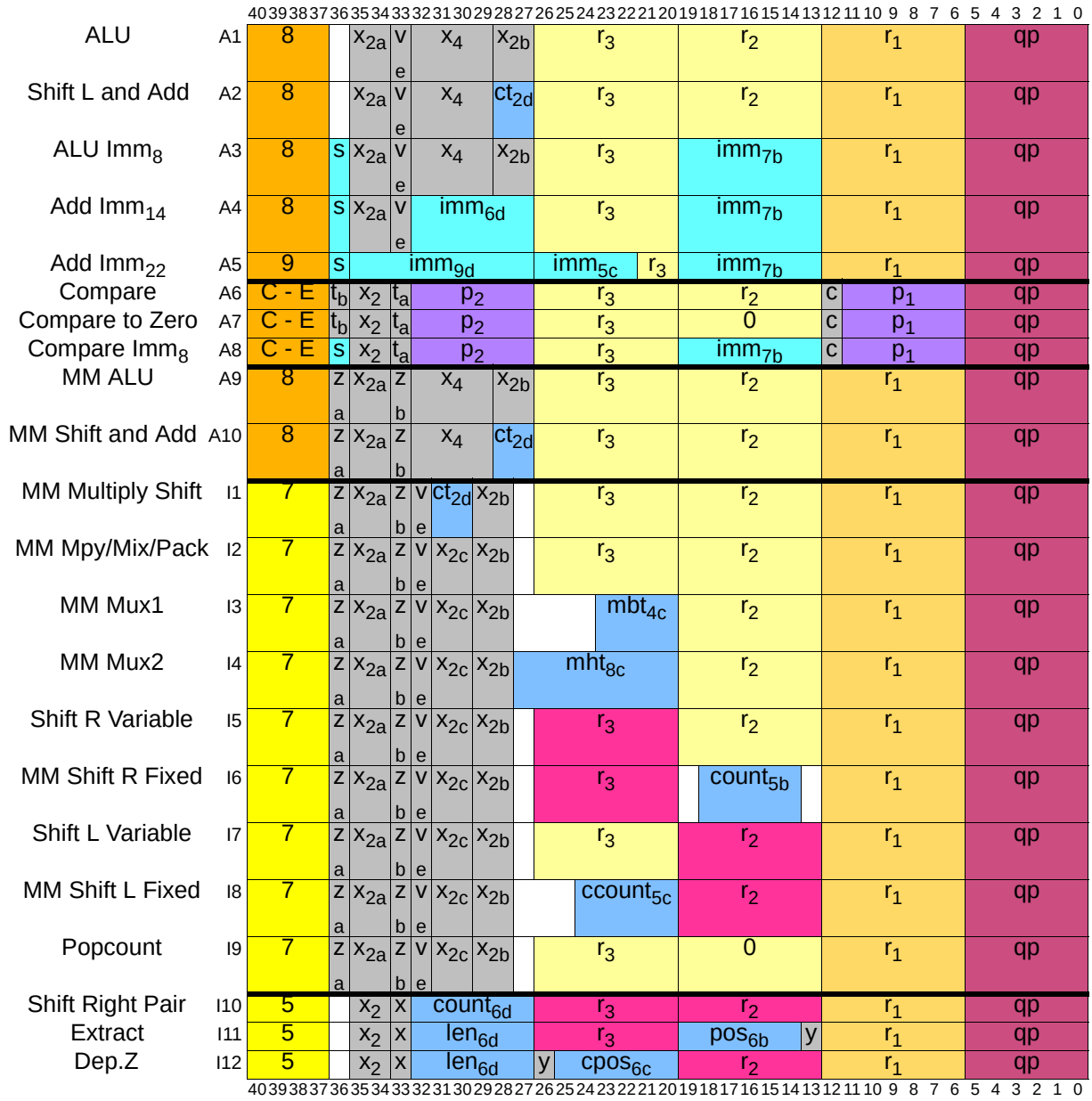


Table 4-4. Instruction Format Summary (Continued)

		40 39 38 37	36	35 34 33	32 31 30 29 28 27	26	25 24 23 22 21 20	19 18 17 16 15 14 13	12 11 10 9 8 7 6	5 4 3 2 1 0
Dep.Z Imm_8	I13	5	s	x_2 x	len_{6d}	y	$cpos_{6c}$	imm_{7b}	r_1	qp
Deposit Imm_1	I14	5	s	x_2 x	len_{6d}	r_3		$cpos_{6b}$	r_1	qp
Deposit	I15	4	$cpos_{6d}$		len_{4d}	r_3		r_2	r_1	qp
Test Bit	I16	5	t_b	x_2 t_a	p_2	r_3		pos_{6b}	y c p_1	qp
Test NaT	I17	5	t_b	x_2 t_a	p_2	r_3			y c p_1	qp
Break/Nop	I19	0	i	x_3	x_6		imm_{20a}			qp
Int Spec Check	I20	0	s	x_3	imm_{13c}			r_2	imm_{7a}	qp
Move to BR	I21	0		x_3	$timm_{9c}$	ih	x wh	r_2		b_1 qp
Move from BR	I22	0		x_3	x_6			b_2	r_1	qp
Move to Pred	I23	0	s	x_3		$mask_{8c}$		r_2	$mask_{7a}$	qp
Move to Pred Imm_{44}	I24	0	s	x_3	imm_{27a}					qp
Move from Pred/IP	I25	0		x_3	x_6				r_1	qp
Move to AR	I26	0		x_3	x_6	ar_3		r_2		qp
Move to AR Imm_8	I27	0	s	x_3	x_6	ar_3		imm_{7b}		qp
Move from AR	I28	0		x_3	x_6	ar_3			r_1	qp
Sxt/Zxt/Czx	I29	0		x_3	x_6	r_3			r_1	qp
Int Load	M1	4	m	x_6	hint x	r_3			r_1	qp
Int Load +Reg	M2	4	m	x_6	hint x	r_3		r_2	r_1	qp
Int Load +Imm	M3	5	s	x_6	hint i	r_3		imm_{7b}	r_1	qp
Int Store	M4	4	m	x_6	hint x	r_3		r_2		qp
Int Store +Imm	M5	5	s	x_6	hint i	r_3		r_2	imm_{7a}	qp
FP Load	M6	6	m	x_6	hint x	r_3			f_1	qp
FP Load +Reg	M7	6	m	x_6	hint x	r_3		r_2	f_1	qp
FP Load +Imm	M8	7	s	x_6	hint i	r_3		imm_{7b}	f_1	qp
FP Store	M9	6	m	x_6	hint x	r_3		f_2		qp
FP Store +Imm	M10	7	s	x_6	hint i	r_3		f_2	imm_{7a}	qp
FP Load Pair	M11	6	m	x_6	hint x	r_3		f_2	f_1	qp
FP Load Pair +Imm	M12	6	m	x_6	hint x	r_3		f_2	f_1	qp
Line Prefetch	M13	6	m	x_6	hint x	r_3				qp
Line Prefetch +Reg	M14	6	m	x_6	hint x	r_3		r_2		qp
Line Prefetch +Imm	M15	7	s	x_6	hint i	r_3		imm_{7b}		qp
(Cmp &) Exchg	M16	4	m	x_6	hint x	r_3		r_2	r_1	qp
Fetch & Add	M17	4	m	x_6	hint x	r_3		s i_{2b}	r_1	qp
Set FR	M18	6	m	x_6	x			r_2	f_1	qp
Get FR	M19	4	m	x_6	x			f_2	r_1	qp
Int Spec Check	M20	1	s	x_3	imm_{13c}			r_2	imm_{7a}	qp
FP Spec Check	M21	1	s	x_3	imm_{13c}			f_2	imm_{7a}	qp
Int ALAT Check	M22	0	s	x_3	imm_{20b}				r_1	qp
FP ALAT Check	M23	0	s	x_3	imm_{20b}				f_1	qp
Sync/Srlz/ALAT	M24	0		x_3	x_2 x_4					qp
RSE Control	M25	0		x_3	x_2 x_4					0
Int ALAT Inval	M26	0		x_3	x_2 x_4				r_1	qp
FP ALAT Inval	M27	0		x_3	x_2 x_4				f_1	qp
Flush Cache/Ptc.e	M28	1		x_3	x_6	r_3				qp
Move to AR	M29	1		x_3	x_6	ar_3		r_2		qp
Move to AR Imm_8	M30	0	s	x_3	x_2 x_4	ar_3		imm_{7b}		qp
Move from AR	M31	1		x_3	x_6	ar_3			r_1	qp
Move to CR	M32	1		x_3	x_6	cr_3		r_2		qp
Move from CR	M33	1		x_3	x_6	cr_3			r_1	qp

Table 4-4. Instruction Format Summary (Continued)

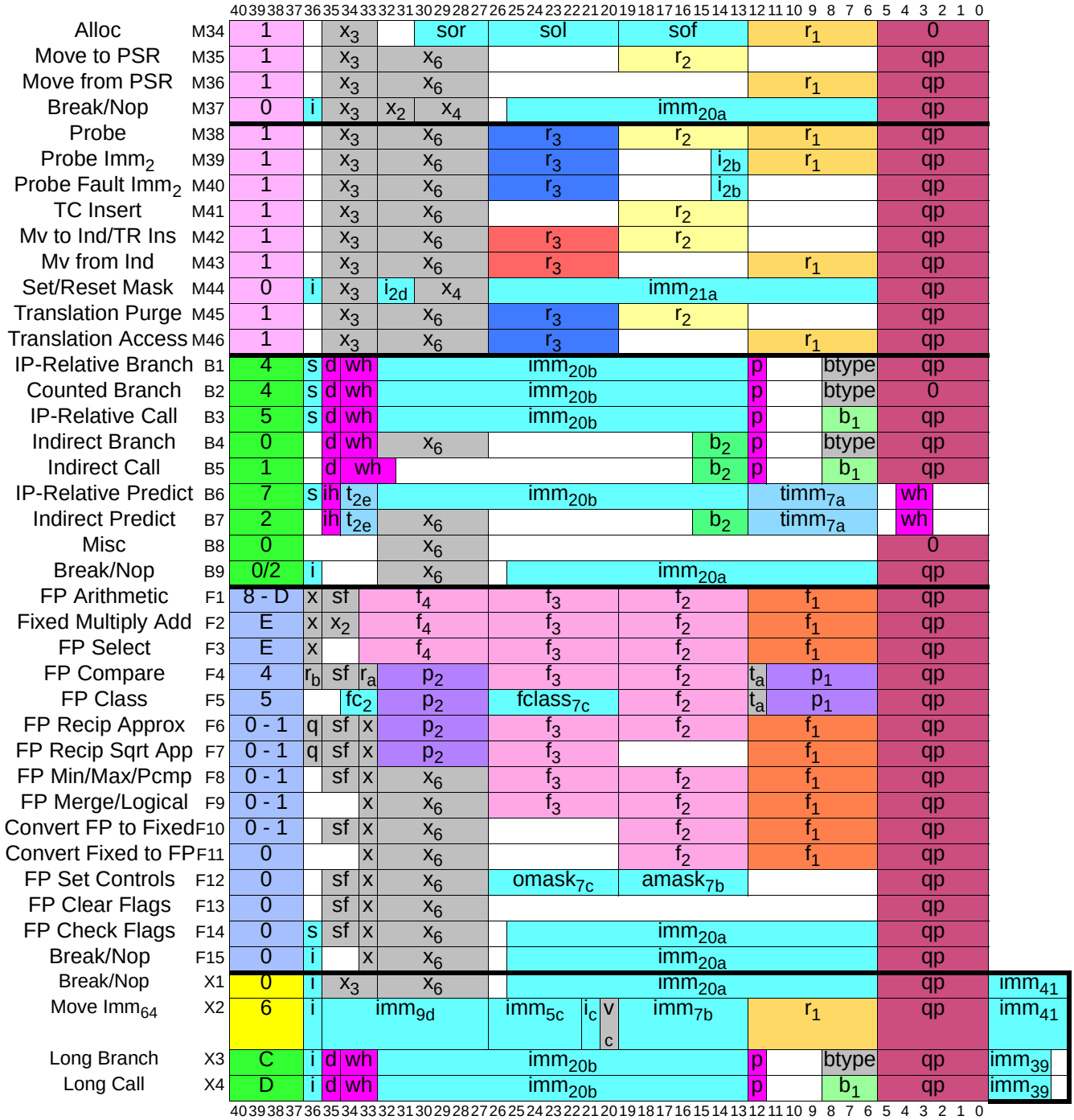


Table 4-5. Instruction Field Color Key

Field & Color	
ALU Instruction	Opcode Extension
Integer Instruction	Opcode Hint Extension
Memory Instruction	Immediate
Branch Instruction	Indirect Source
Floating-point Instruction	Predicate Destination
Integer Source	Integer Destination
Memory Source	Memory Source & Destination
Shift Source	Shift Immediate
Special Register Source	Special Register Destination
Floating-point Source	Floating-point Destination
Branch Source	Branch Destination
Address Source	Branch Tag Immediate
Qualifying Predicate	Reserved Instruction
Ignored Field/Instruction	Reserved Inst if PR[qp] is 1
	Reserved B-type Inst if PR[qp] is 1

The remaining sections of this chapter present the detailed encodings of all instructions. The "A-Unit Instruction encodings" are presented first, followed by the "I-Unit Instruction Encodings" on page 4-20, "M-Unit Instruction Encodings" on page 4-34, "B-Unit Instruction Encodings" on page 4-63, "F-Unit Instruction Encodings" on page 4-71, and "X-Unit Instruction Encodings" on page 4-81.

Within each section, the instructions are grouped by function, and appear with their instruction format in the same order as in Table 4-4, "Instruction Format Summary," on page 4-4. The opcode extension fields are briefly described and tables present the opcode extension assignments. Unused instruction encodings (appearing as blank entries in the opcode extensions tables) behave in one of four ways:

- Ignored instructions (white entries in the tables) execute as `nop` instructions.
- Reserved instructions (light gray color in the gray scale version of the tables) cause an Illegal Operation fault.
- Reserved if PR[qp] is 1 instructions (dark gray in the gray scale version of the tables) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0.
- Reserved if PR[qp] is 1 B-unit instructions (medium gray in the gray scale version of the tables) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0. These differ from the Reserved if PR[qp] is 1 instructions only in their RAW dependency behavior (see "RAW Dependency Table" on page A-4 in Volume 2).

Table 4-6. Instruction Field Names

Field Name	Description
ar_3	application register source/target
b_1, b_2	branch register source/target
btype	branch type opcode extension
c	complement compare relation opcode extension
$ccount_{5c}$	multimedia shift left complemented shift count immediate
$count_{5b}$, $count_{6d}$	multimedia shift right/shift right pair shift count immediate
$cpos_x$	deposit complemented bit position immediate
cr_3	control register source/target
ct_{2d}	multimedia multiply shift/shift and add shift count immediate
d	branch cache deallocation hint opcode extension
f_n	floating-point register source/target
fc_2, $fclass_{7c}$	floating-point class immediate
hint	memory reference hint opcode extension
i, i_{2b}, $i_{2d,}$ imm_x	immediate of length 1, 2, or *x*
ih	branch importance hint opcode extension
len_{4d}, len_{6d}	extract/deposit length immediate
m	memory reference post-modify opcode extension
$mask_x$	predicate immediate mask
mbt_{4c}, mht_{8c}	multimedia mux1/mux2 immediate
p	sequential prefetch hint opcode extension
p_1, p_2	predicate register target
pos_{6b}	test bit/extract bit position immediate
q	floating-point reciprocal/reciprocal square-root opcode extension
qp	qualifying predicate register source
r_n	general register source/target
s	immediate sign bit
sf	floating-point status field opcode extension
sof, sol, sor	alloc size of frame, size of locals, size of rotating immediates
t_a, t_b	compare type opcode extension
t_{2e}, $timm_x$	branch predict tag immediate
v_x	reserved opcode extension field
wh	branch whether hint opcode extension
x, x_n	opcode extension of length 1 or *n*
y	extract/deposit/test bit/test NaT opcode extension
z_a, z_b	multimedia operand size opcode extension

Table 4-7. Special Instruction Notations

Notation	Description
e	instruction ends an instruction group when taken, or for Reserved if PR[qp] is 1 encodings and non-branch instructions with a qualifying predicate, when its PR[qp] is 1, or for Reserved encodings, unconditionally
f	instruction must be the first in an instruction group
i	instruction is allowed in the I slot of an MLI template
l	instruction must be the last in an instruction group
p	privileged instruction
t	instruction is only allowed in instruction slot 2

Some processors may implement the Reserved if PR[qp] is 1 and Reserved if PR[qp] is 1 B-unit encodings in the L+X opcode space as Reserved. These encodings appear in the L+X column of Table 4-3 on page 4-3, and in Table 4-66 on page 4-81, Table 4-67 on page 4-81, Table 4-68 on page 4-82, and Table 4-69 on page 4-83. On processors which implement these encodings as Reserved, the operating system is required to provide an Illegal Operation fault handler which emulates them as Reserved if PR[qp] is 1 by decoding the reserved opcodes, checking the qualifying predicate, and returning to the next instruction if PR[qp] is 0.

Constant 0 fields in instructions must be 0 or undefined operation results. The undefined operation may include checking that the constant field is 0 and causing an Illegal Operation fault if it is not. If an instruction having a constant 0 field also has a qualifying predicate (qp field), the fault or other undefined operation must not occur if PR[qp] is 0. For constant 0 fields in instruction bits 5:0 (normally used for qp), the fault or other undefined operation may or may not depend on the PR addressed by those bits.

Ignored (white space) fields in instructions should be coded as 0. Although ignored in this revision of the architecture, future architecture revisions may define these fields as hint extensions. These hint extensions will be defined such that the 0 value in each field corresponds to the default hint. It is expected that assemblers will automatically set these fields to zero by default.

4.2 A-Unit Instruction Encodings

4.2.1 Integer ALU

All integer ALU instructions are encoded within major opcode 8 using a 2-bit opcode extension field in bits 35:34 (x_{2a}) and most have a second 2-bit opcode extension field in bits 28:27 (x_{2b}), a 4-bit opcode extension field in bits 32:29 (x_4), and a 1-bit reserved opcode extension field in bit 33 (v_e). Table 4-8 shows the 2-bit x_{2a} and 1-bit v_e assignments, Table 4-9 shows the integer ALU 4-bit+2-bit assignments, and Table 4-12 on page 4-17 shows the multimedia ALU 1-bit+2-bit assignments (which also share major opcode 8).

Table 4-8. Integer ALU 2-bit+1-bit Opcode Extensions

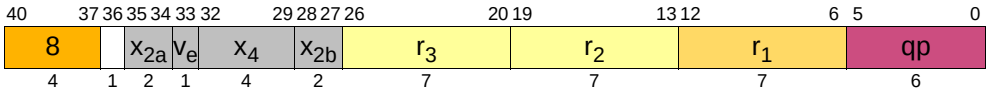
Opcode Bits 40:37	x_{2a} Bits 35:34	v_e Bit 33	
		0	1
8	0	Integer ALU 4-bit+2-bit Ext (Table 4-9)	
	1	Multimedia ALU 1-bit+2-bit Ext (Table 4-12)	
	2	adds – imm_{14} A4	
	3	addp4 – imm_{14} A4	

Table 4-9. Integer ALU 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	v_e Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
				0	1	2	3
8	0	0	0	add A1	add +1 A1		
			1	sub –1 A1	sub A1		
			2	addp4 A1			
			3	and A1	andcm A1	or A1	xor A1
			4	shladd A2			
			5				
			6	shladdp4 A2			
			7				
			8				
			9		sub – imm_8 A3		
			A				
			B	and – imm_8 A3	andcm – imm_8 A3	or – imm_8 A3	xor – imm_8 A3
			C				
			D				
			E				
			F				

4.2.1.1 Integer ALU – Register-Register

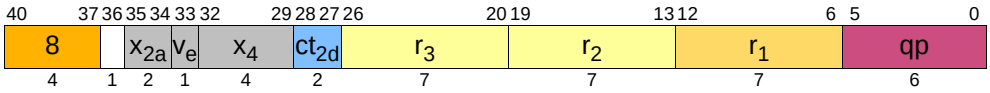
A1



Instruction	Operands	Opcode	Extension			
			x_{2a}	v_e	x_4	x_{2b}
add	$r_1 = r_2, r_3$	8	0	0	0	0
	$r_1 = r_2, r_3, 1$					1
sub	$r_1 = r_2, r_3$				1	1
	$r_1 = r_2, r_3, 1$					0
addp4	$r_1 = r_2, r_3$				2	0
and					3	0
andcm						1
or						2
xor						3

4.2.1.2 Shift Left and Add

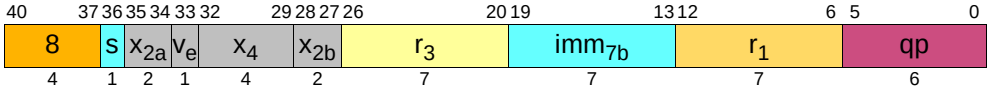
A2



Instruction	Operands	Opcode	Extension		
			x_{2a}	v_e	x_4
shladd	$r_1 = r_2, count_2, r_3$	8	0	0	4
shladdp4					6

4.2.1.3 Integer ALU – Immediate$_8$-Register

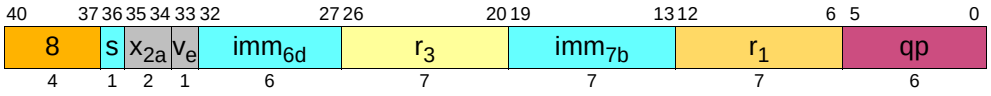
A3



Instruction	Operands	Opcode	Extension			
			x_{2a}	v_e	x_4	x_{2b}
sub	$r_1 = imm_8, r_3$	8	0	0	9	1
and					B	0
andcm						1
or						2
xor						3

4.2.1.4 Add Immediate$_{14}$

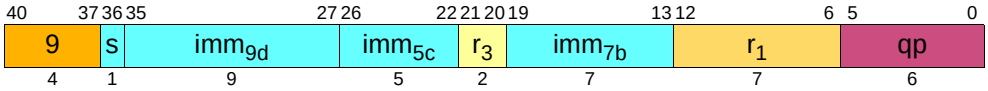
A4



Instruction	Operands	Opcode	Extension	
			x_{2a}	v_e
adds addp4	$r_1 = imm_{14}, r_3$	8	2 3	0

4.2.1.5 Add Immediate$_{22}$

A5



Instruction	Operands	Opcode
addl	$r_1 = imm_{22}, r_3$	9

4.2.2 Integer Compare

The integer compare instructions are encoded within major opcodes C – E using a 2-bit opcode extension field (x_2) in bits 35:34 and three 1-bit opcode extension fields in bits 33 (t_a), 36 (t_b), and 12 (c), as shown in Table 4-10. The integer compare immediate instructions are encoded within major opcodes C – E using a 2-bit opcode extension field (x_2) in bits 35:34 and two 1-bit opcode extension fields in bits 33 (t_a) and 12 (c), as shown in Table 4-11.

Table 4-10. Integer Compare Opcode Extensions

x_2 Bits 35:34	t_b Bit 36	t_a Bit 33	c Bit 12	Opcode Bits 40:37		
				C	D	E
0	0	0	0	cmp.lt A6	cmp.ltu A6	cmp.eq A6
			1	cmp.lt.unc A6	cmp.ltu.unc A6	cmp.eq.unc A6
		1	0	cmp.eq.and A6	cmp.eq.or A6	cmp.eq.or.andcm A6
			1	cmp.ne.and A6	cmp.ne.or A6	cmp.ne.or.andcm A6
	1	0	0	cmp.gt.and A7	cmp.gt.or A7	cmp.gt.or.andcm A7
			1	cmp.le.and A7	cmp.le.or A7	cmp.le.or.andcm A7
		1	0	cmp.ge.and A7	cmp.ge.or A7	cmp.ge.or.andcm A7
			1	cmp.lt.and A7	cmp.lt.or A7	cmp.lt.or.andcm A7

Table 4-10. Integer Compare Opcode Extensions (Continued)

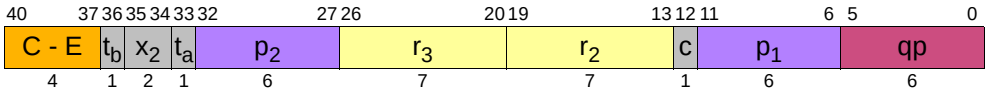
x_2 Bits 35:34	t_b Bit 36	t_a Bit 33	c Bit 12	Opcode Bits 40:37		
				C	D	E
1	0	0	0	cmp4.lt A6	cmp4.ltu A6	cmp4.eq A6
			1	cmp4.lt.unc A6	cmp4.ltu.unc A6	cmp4.eq.unc A6
		1	0	cmp4.eq.and A6	cmp4.eq.or A6	cmp4.eq.or.andcm A6
			1	cmp4.ne.and A6	cmp4.ne.or A6	cmp4.ne.or.andcm A6
	1	0	0	cmp4.gt.and A7	cmp4.gt.or A7	cmp4.gt.or.andcm A7
			1	cmp4.le.and A7	cmp4.le.or A7	cmp4.le.or.andcm A7
		1	0	cmp4.ge.and A7	cmp4.ge.or A7	cmp4.ge.or.andcm A7
			1	cmp4.lt.and A7	cmp4.lt.or A7	cmp4.lt.or.andcm A7

Table 4-11. Integer Compare Immediate Opcode Extensions

x_2 Bits 35:34	t_a Bit 33	c Bit 12	Opcode Bits 40:37		
			C	D	E
2	0	0	cmp.lt – imm_8 A8	cmp.ltu – imm_8 A8	cmp.eq – imm_8 A8
		1	cmp.lt.unc – imm_8 A8	cmp.ltu.unc – imm_8 A8	cmp.eq.unc – imm_8 A8
	1	0	cmp.eq.and – imm_8 A8	cmp.eq.or – imm_8 A8	cmp.eq.or.andcm – imm_8 A8
		1	cmp.ne.and – imm_8 A8	cmp.ne.or – imm_8 A8	cmp.ne.or.andcm – imm_8 A8
3	0	0	cmp4.lt – imm_8 A8	cmp4.ltu – imm_8 A8	cmp4.eq – imm_8 A8
		1	cmp4.lt.unc – imm_8 A8	cmp4.ltu.unc – imm_8 A8	cmp4.eq.unc – imm_8 A8
	1	0	cmp4.eq.and – imm_8 A8	cmp4.eq.or – imm_8 A8	cmp4.eq.or.andcm – imm_8 A8
		1	cmp4.ne.and – imm_8 A8	cmp4.ne.or – imm_8 A8	cmp4.ne.or.andcm – imm_8 A8

4.2.2.1 Integer Compare – Register-Register

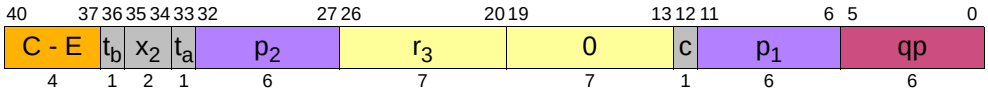
A6



Instruction	Operands	Opcode	Extension			
			x_2	t_b	t_a	c
cmp.lt	$p_1, p_2 = r_2, r_3$	C	0	0	0	0
cmp.ltu		D				
cmp.eq		E				
cmp.lt.unc		C				1
cmp.ltu.unc		D				
cmp.eq.unc		E				
cmp.eq.and		C			1	0
cmp.eq.or		D				
cmp.eq.or.andcm		E				
cmp.ne.and		C				1
cmp.ne.or		D				
cmp.ne.or.andcm		E				
cmp4.lt		C	1	0	0	0
cmp4.ltu		D				
cmp4.eq		E				
cmp4.lt.unc		C				1
cmp4.ltu.unc		D				
cmp4.eq.unc		E				
cmp4.eq.and		C			1	0
cmp4.eq.or		D				
cmp4.eq.or.andcm		E				
cmp4.ne.and		C				1
cmp4.ne.or		D				
cmp4.ne.or.andcm		E				

4.2.2.2 Integer Compare to Zero – Register

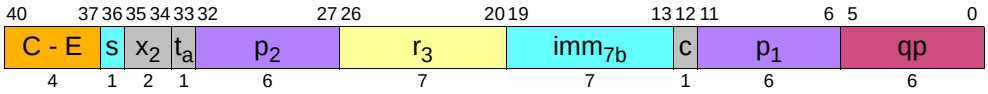
A7



Instruction	Operands	Opcode	Extension			
			x_2	t_b	t_a	c
cmp.gt.and	p_1, p_2 = r0, r_3	C	0	1	0	0
cmp.gt.or		D				
cmp.gt.or.andcm		E				
cmp.le.and		C				1
cmp.le.or		D				
cmp.le.or.andcm		E				
cmp.ge.and		C			1	0
cmp.ge.or		D				
cmp.ge.or.andcm		E				
cmp.lt.and		C				1
cmp.lt.or		D				
cmp.lt.or.andcm		E				
cmp4.gt.and		C	1		0	0
cmp4.gt.or		D				
cmp4.gt.or.andcm		E				
cmp4.le.and		C				1
cmp4.le.or		D				
cmp4.le.or.andcm		E				
cmp4.ge.and		C			1	0
cmp4.ge.or		D				
cmp4.ge.or.andcm		E				
cmp4.lt.and		C				1
cmp4.lt.or		D				
cmp4.lt.or.andcm		E				

4.2.2.3 Integer Compare – Immediate-Register

A8



Instruction	Operands	Opcode	Extension		
			x_2	t_a	c
cmp.lt	$p_1, p_2 = imm_8, r_3$	C	2	0	0
cmp.ltu		D			
cmp.eq		E			
cmp.lt.unc		C			1
cmp.ltu.unc		D			
cmp.eq.unc		E			
cmp.eq.and		C		1	0
cmp.eq.or		D			
cmp.eq.or.andcm		E			
cmp.ne.and		C			1
cmp.ne.or		D			
cmp.ne.or.andcm		E			
cmp4.lt		C	3	0	0
cmp4.ltu		D			
cmp4.eq		E			
cmp4.lt.unc		C			1
cmp4.ltu.unc		D			
cmp4.eq.unc		E			
cmp4.eq.and		C		1	0
cmp4.eq.or		D			
cmp4.eq.or.andcm		E			
cmp4.ne.and		C			1
cmp4.ne.or		D			
cmp4.ne.or.andcm		E			

4.2.3 Multimedia

All multimedia ALU instructions are encoded within major opcode 8 using two 1-bit opcode extension fields in bits 36 (z_a) and 33 (z_b) and a 2-bit opcode extension field in bits 35:34 (x_{2a}) as shown in Table 4-12. The multimedia ALU instructions also have a 4-bit opcode extension field in bits 32:29 (x_4), and a 2-bit opcode extension field in bits 28:27 (x_{2b}) as shown in Table 4-13 on page 4-17.

Table 4-12. Multimedia ALU 2-bit+1-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	
8	1	0	0	Multimedia ALU Size 1 (Table 4-13)
			1	Multimedia ALU Size 2 (Table 4-14)
		1	0	Multimedia ALU Size 4 (Table 4-15)
			1	

Table 4-13. Multimedia ALU Size 1 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	0	0	0	padd1 A9	padd1.sss A9	padd1.uuu A9	padd1.uus A9
				1	psub1 A9	psub1.sss A9	psub1.uuu A9	psub1.uus A9
				2			pavg1 A9	pavg1.raz A9
				3			pavgsub1 A9	
				4				
				5				
				6				
				7				
				8				
				9	pcmp1.eq A9	pcmp1.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

Table 4-14. Multimedia ALU Size 2 4-bit+2-bit Opcode Extensions

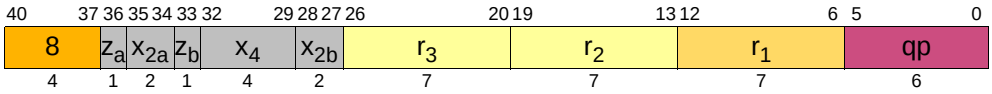
Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	0	1	0	padd2 A9	padd2.sss A9	padd2.uuu A9	padd2.uus A9
				1	psub2 A9	psub2.sss A9	psub2.uuu A9	psub2.uus A9
				2			pavg2 A9	pavg2.raz A9
				3			pavgsub2 A9	
				4	pshladd2 A10			
				5				
				6	pshradd2 A10			
				7				
				8				
				9	pcmp2.eq A9	pcmp2.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

Table 4-15. Multimedia ALU Size 4 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	1	0	0	padd4 A9			
				1	psub4 A9			
				2				
				3				
				4				
				5				
				6				
				7				
				8				
				9	pcmp4.eq A9	pcmp4.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

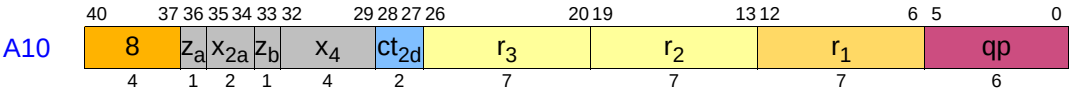
4.2.3.1 Multimedia ALU

A9



Instruction	Operands	Opcode	Extension				
			x_{2a}	z_a	z_b	x_4	x_{2b}
padd1	$r_1 = r_2, r_3$	8	1	0	0	0	0
padd2					1		
padd4				1	0		
padd1.sss				0	0		1
padd2.sss					1		
padd1.uuu				0	0		2
padd2.uuu					1		
padd1.uus				0	0		3
padd2.uus					1		
psub1				0	0	1	0
psub2					1		
psub4				1	0		
psub1.sss				0	0		1
psub2.sss					1		
psub1.uuu				0	0		2
psub2.uuu					1		
psub1.uus				0	0		3
psub2.uus					1		
pavg1				0	0	2	2
pavg2					1		
pavg1.raz				0	0		3
pavg2.raz					1		
pavgsub1				0	0	3	2
pavgsub2					1		
pcmp1.eq				0	0	9	0
pcmp2.eq					1		
pcmp4.eq				1	0		
pcmp1.gt				0	0		1
pcmp2.gt					1		
pcmp4.gt				1	0		

4.2.3.2 Multimedia Shift and Add



Instruction	Operands	Opcode	Extension			
			x_{2a}	z_a	z_b	x_4
pshladd2 pshradd2	$r_1 = r_2$, $count_2$, r_3	8	1	0	1	4 6

4.3 I-Unit Instruction Encodings

4.3.1 Multimedia and Variable Shifts

All multimedia multiply/shift/max/min/mix/mux/pack/unpack and variable shift instructions are encoded within major opcode 7 using two 1-bit opcode extension fields in bits 36 (z_a) and 33 (z_b) and a 1-bit reserved opcode extension in bit 32 (v_e) as shown in Table 4-16. They also have a 2-bit opcode extension field in bits 35:34 (x_{2a}) and a 2-bit field in bits 29:28 (x_{2b}) and most have a 2-bit field in bits 31:30 (x_{2c}) as shown in Table 4-17.

Table 4-16. Multimedia and Variable Shift 1-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	
			0	1
7	0	0	Multimedia Size 1 (Table 4-17)	
		1	Multimedia Size 2 (Table 4-18)	
	1	0	Multimedia Size 4 (Table 4-19)	
		1	Variable Shift (Table 4-20)	

Table 4-17. Multimedia Opcode 7 Size 1 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	0	0	0	0	0				
					1				
					2				
					3				
				1	0				
					1				
					2				
					3				
				2	0		unpack1.h I2	mix1.r I2	
					1	pmin1.u I2	pmax1.u I2		
					2		unpack1.l I2	mix1.l I2	
					3			psad1 I2	
				3	0				
					1				
					2			mux1 I3	
					3				

Table 4-18. Multimedia Opcode 7 Size 2 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	0	1	0	0	0	pshr2.u – var I5	pshl2 – var I7		
					1	pmpyshr2.u I1			
					2	pshr2 – var I5			
					3	pmpyshr2 I1			
				1	0				
					1	pshr2.u – fixed I6		popcnt I9	
					2				
					3	pshr2 – fixed I6			
				2	0	pack2.uss I2	unpack2.h I2	mix2.r I2	
					1				pmpy2.r I2
					2	pack2.sss I2	unpack2.l I2	mix2.l I2	
					3	pmin2 I2	pmax2 I2		pmpy2.l I2
				3	0				
					1		pshl2 – fixed I8		
					2			mux2 I4	
					3				

Table 4-19. Multimedia Opcode 7 Size 4 2-bit Opcode Extensions

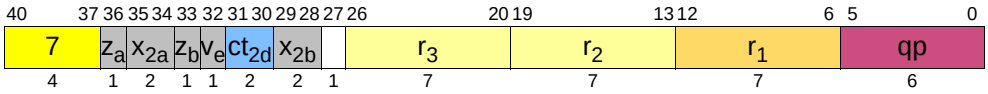
Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	1	0	0	0	0	pshr4.u – var I5	pshl4 – var I7		
					1				
					2	pshr4 – var I5			
					3				
				1	0				
					1	pshr4.u – fixed I6			
					2				
					3	pshr4 – fixed I6			
				2	0		unpack4.h I2	mix4.r I2	
					1				
					2	pack4.sss I2	unpack4.l I2	mix4.l I2	
					3				
				3	0				
					1		pshl4 – fixed I8		
					2				
					3				

Table 4-20. Variable Shift Opcode 7 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	1	1	0	0	0	shr.u – var I5	shl – var I7		
					1				
					2	shr – var I5			
					3				
				1	0				
					1				
					2				
					3				
				2	0				
					1				
					2				
					3				
				3	0				
					1				
					2				
					3				

4.3.1.1 Multimedia Multiply and Shift

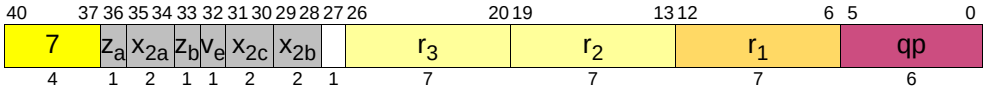
I1



Instruction	Operands	Opcode	Extension				
			z_a	z_b	v_e	x_{2a}	x_{2b}
pmpyshr2 pmpyshr2.u	$r_1 = r_2, r_3, count_2$	7	0	1	0	0	3 1

4.3.1.2 Multimedia Multiply/Mix/Pack/Unpack

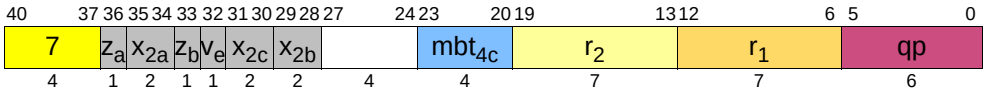
I2



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pmpy2.r pmpy2.l	$r_1 = r_2, r_3$	7	0	1	0	2	1 3	3
mix1.r mix2.r mix4.r			0 0 1	0 1 0			0	2
mix1.l mix2.l mix4.l			0 0 1	0 1 0			2	
pack2.uss			0	1			0	0
pack2.sss pack4.sss			0 1	1 0			2	
unpack1.h unpack2.h unpack4.h			0 0 1	0 1 0			0	1
unpack1.l unpack2.l unpack4.l			0 0 1	0 1 0			2	
pmin1.u pmax1.u			0	0			1	0 1
pmin2 pmax2			0	1			3	0 1
psad1			0	0			3	2

4.3.1.3 Multimedia Mux1

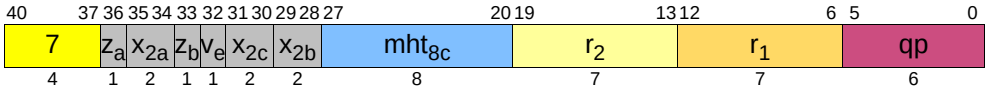
I3



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mux1	$r_1 = r_2$, $mbtype_4$	7	0	0	0	3	2	2

4.3.1.4 Multimedia Mux2

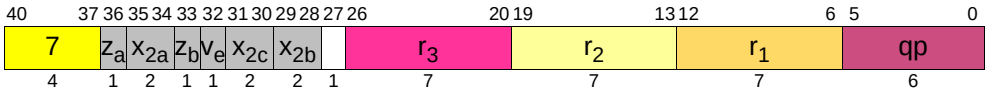
I4



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mux2	$r_1 = r_2$, $mhtype_8$	7	0	1	0	3	2	2

4.3.1.5 Shift Right – Variable

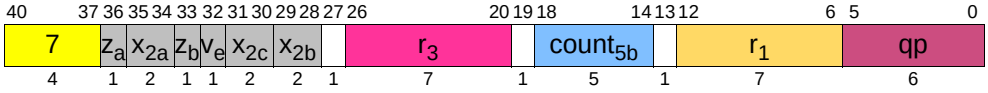
I5



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshr2	$r_1 = r_3$, r_2	7	0	1	0	0	2	0
pshr4			1	0				
shr			1	1				
pshr2.u			0	1			0	
pshr4.u			1	0				
shr.u			1	1				

4.3.1.6 Multimedia Shift Right – Fixed

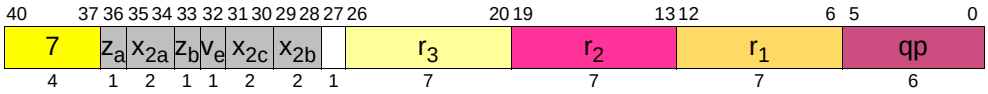
I6



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshr2	$r_1 = r_3$, $count_5$	7	0	1	0	1	3	0
pshr4			1	0				
pshr2.u			0	1			1	
pshr4.u			1	0				

4.3.1.7 Shift Left – Variable

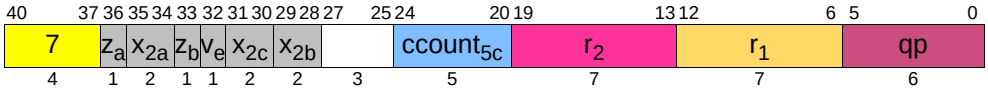
I7



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshl2	$r_1 = r_2, r_3$	7	0	1	0	0	0	1
pshl4			1	0				
shl			1	1				

4.3.1.8 Multimedia Shift Left – Fixed

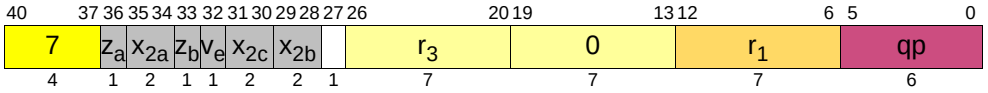
I8



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshl2	$r_1 = r_2, count_5$	7	0	1	0	3	1	1
pshl4			1	0				

4.3.1.9 Population Count

I9



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
popcnt	$r_1 = r_3$	7	0	1	0	1	1	2

4.3.2 Integer Shifts

The integer shift, test bit, and test NaT instructions are encoded within major opcode 5 using a 2-bit opcode extension field in bits 35:34 (x_2) and a 1-bit opcode extension field in bit 33 (x). The extract and test bit instructions also have a 1-bit opcode extension field in bit 13 (y). Table 4-21 shows the test bit, extract, and shift right pair assignments. Most deposit instructions also have a 1-bit opcode extension field in bit 26 (y). Table 4-22 shows these assignments.

Table 4-21. Integer Shift/Test Bit/Test NaT 2-bit Opcode Extensions

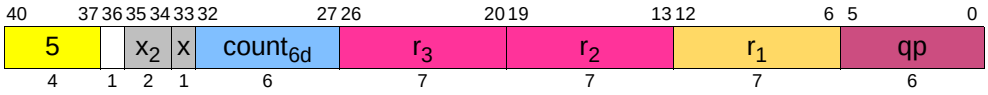
Opcode Bits 40:37	x_2 Bits 35:34	x Bit 33	y Bit 13	
			0	1
5	0	0	Test Bit (Table 4-23)	Test NaT (Table 4-23)
	1		extr.u I11	extr I11
	2			
	3		shrp I10	

Table 4-22. Deposit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	x Bit 33	y Bit 26	
			0	1
5	0	1	Test Bit/Test NaT (Table 4-23)	
	1		dep.z I12	dep.z – imm_8 I13
	2			
	3		dep – imm_1 I14	

4.3.2.1 Shift Right Pair

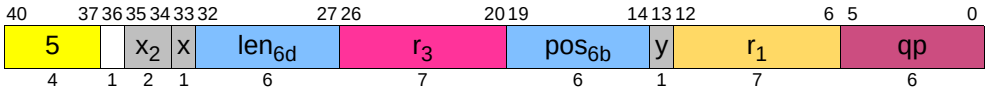
I10



Instruction	Operands	Opcode	Extension	
			x_2	x
shrp	$r_1 = r_2, r_3, count_6$	5	3	0

4.3.2.2 Extract

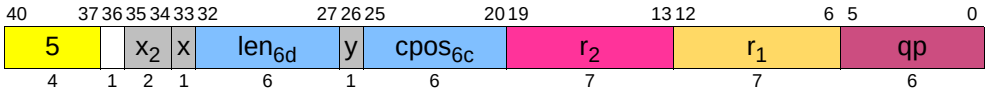
I11



Instruction	Operands	Opcode	Extension		
			x_2	x	y
extr.u extr	$r_1 = r_3, pos_6, len_6$	5	1	0	0 1

4.3.2.3 Zero and Deposit

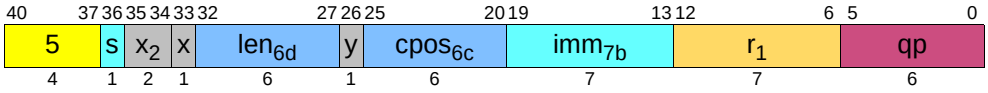
I12



Instruction	Operands	Opcode	Extension		
			x_2	x	y
dep.z	$r_1 = r_2, pos_6, len_6$	5	1	1	0

4.3.2.4 Zero and Deposit Immediate$_8$

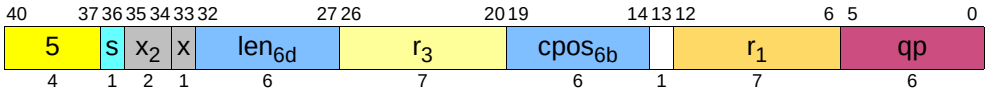
I13



Instruction	Operands	Opcode	Extension		
			x_2	x	y
dep.z	$r_1 = imm_8, pos_6, len_6$	5	1	1	1

4.3.2.5 Deposit Immediate$_1$

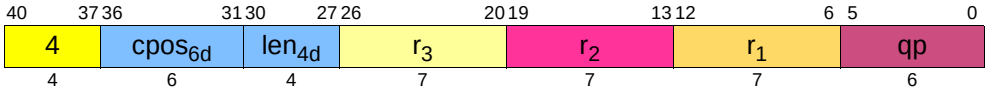
I14



Instruction	Operands	Opcode	Extension	
			x_2	x
dep	$r_1 = imm_1, r_3, pos_6, len_6$	5	3	1

4.3.2.6 Deposit

I15



Instruction	Operands	Opcode
dep	$r_1 = r_2, r_3, pos_6, len_4$	4

4.3.3 Test Bit

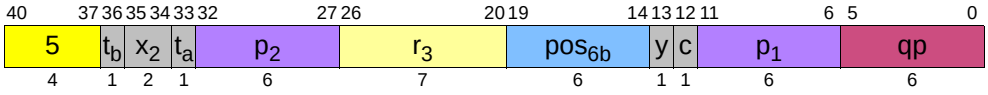
All test bit instructions are encoded within major opcode 5 using a 2-bit opcode extension field in bits 35:34 (x_2) plus four 1-bit opcode extension fields in bits 33 (t_a), 36 (t_b), 12 (c), and 19 (y). Table 4-23 summarizes these assignments.

Table 4-23. Test Bit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	t_a Bit 33	t_b Bit 36	c Bit 12	y Bit 13	
					0	1
5	0	0	0	0	tbit.z I16	tnat.z I17
				1	tbit.z.unc I16	tnat.z.unc I17
			1	0	tbit.z.and I16	tnat.z.and I17
				1	tbit.nz.and I16	tnat.nz.and I17
		1	0	0	tbit.z.or I16	tnat.z.or I17
				1	tbit.nz.or I16	tnat.nz.or I17
			1	0	tbit.z.or.andcm I16	tnat.z.or.andcm I17
				1	tbit.nz.or.andcm I16	tnat.nz.or.andcm I17

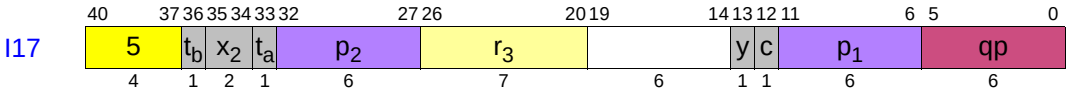
4.3.3.1 Test Bit

I16



Instruction	Operands	Opcode	Extension				
			x_2	t_a	t_b	y	c
tbit.z	$p_1, p_2 = r_3, pos_6$	5	0	0	0	0	0
tbit.z.unc							1
tbit.z.and					1		0
tbit.nz.and							1
tbit.z.or				1	0		0
tbit.nz.or							1
tbit.z.or.andcm					1		0
tbit.nz.or.andcm							1

4.3.3.2 Test NaT



Instruction	Operands	Opcode	Extension				
			x_2	t_a	t_b	y	c
tnat.z tnat.z.unc	$p_1, p_2 = r_3$	5	0	0	0	1	0 1
tnat.z.and tnat.nz.and					1		0 1
tnat.z.or tnat.nz.or				1	0		0 1
tnat.z.or.andcm tnat.nz.or.andcm					1		0 1

4.3.4 Miscellaneous I-Unit Instructions

The miscellaneous I-unit instructions are encoded in major opcode 0 using a 3-bit opcode extension field (x_3) in bits 35:33. Some also have a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-24 shows the 3-bit assignments and Table 4-25 summarizes the 6-bit assignments.

Table 4-24. Misc I-Unit 3-bit Opcode Extensions

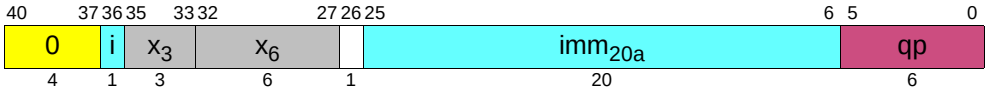
Opcode Bits 40:37	x_3 Bits 35:33	
0	0	6-bit Ext (Table 4-25)
	1	chk.s.i – int I20
	2	mov to pr.rot – imm_{44} I24
	3	mov to pr I23
	4	
	5	
	6	
	7	mov to b I21

Table 4-25. Misc I-Unit 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 30:27	Bits 32:31 = 0	Bits 32:31 = 1	Bits 32:31 = 2	Bits 32:31 = 3
0	0	0	break.i I19	zxt1 I29		mov from ip I25
		1	nop.i I19	zxt2 I29		mov from b I22
		2		zxt4 I29		mov.i from ar I28
		3				mov from pr I25
		4		sxt1 I29		
		5		sxt2 I29		
		6		sxt4 I29		
		7				
		8		czx1.l I29		
		9		czx2.l I29		
		A	mov.i to ar – imm_8 I27		mov.i to ar I26	
		B				
		C		czx1.r I29		
		D		czx2.r I29		
		E				
		F				

4.3.4.1 Break/Nop (I-Unit)

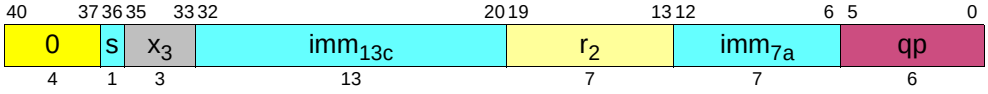
I19



Instruction	Operands	Opcode	Extension x_3	Extension x_6
break.i [i] nop.i [i]	imm_{21}	0	0	00 01

4.3.4.2 Integer Speculation Check (I-Unit)

I20



Instruction	Operands	Opcode	Extension x_3
chk.s.i	r_2, $target_{25}$	0	1

4.3.5 GR/BR Moves

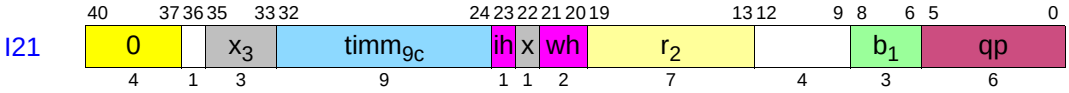
The GR/BR move instructions are encoded in major opcode 0. See "Miscellaneous I-Unit Instructions" on page 4-29 for a summary of the opcode extensions. The mov to BR instruction uses a 2-bit "whether" prediction hint field in bits 21:20 (wh) as shown in Table 4-26.

Table 4-26. Move to BR Whether Hint Completer

wh Bits 21:20	*mwh*
0	.sptk
1	*none*
2	.dptk
3	

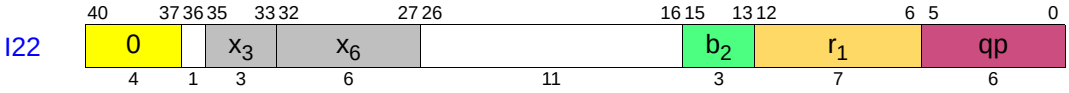
The mov to BR instruction also uses a 1-bit opcode extension field (x) in bit 22 to distinguish the return form from the normal form, and a 1-bit hint extension in bit 23 (ih) (see Table 4-54 on page 4-69).

4.3.5.1 Move to BR



Instruction	Operands	Opcode	Extension			
			x_3	x	ih	wh
mov.*mwh*.*ih* mov.ret.*mwh*.*ih*	$b_1 = r_2, tag_{13}$	0	7	0 1	See Table 4-54 on page 4-69	See Table 4-26 on page 4-31

4.3.5.2 Move from BR



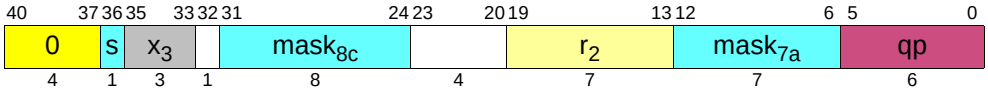
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	$r_1 = b_2$	0	0	31

4.3.6 GR/Predicate/IP Moves

The GR/Predicate/IP move instructions are encoded in major opcode 0. See "Miscellaneous I-Unit Instructions" on page 4-29 for a summary of the opcode extensions.

4.3.6.1 Move to Predicates – Register

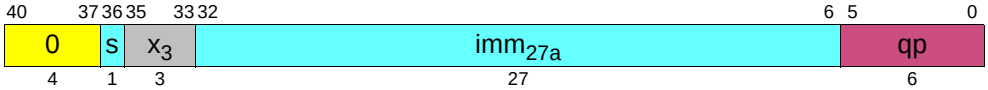
I23



Instruction	Operands	Opcode	Extension
			x_3
mov	pr = r_2, $mask_{17}$	0	3

4.3.6.2 Move to Predicates – Immediate$_{44}$

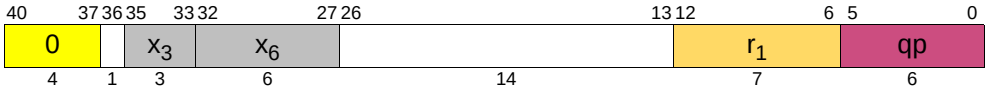
I24



Instruction	Operands	Opcode	Extension
			x_3
mov	pr.rot = imm_{44}	0	2

4.3.6.3 Move from Predicates/IP

I25



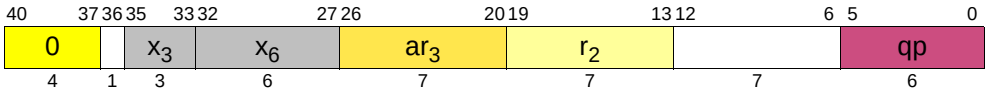
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	r_1 = ip r_1 = pr	0	0	30 33

4.3.7 GR/AR Moves (I-Unit)

The I-Unit GR/AR move instructions are encoded in major opcode 0. (Some ARs are accessed using system/memory management instructions on the M-unit. See "GR/AR Moves (M-Unit)" on page 4-55) See "Miscellaneous I-Unit Instructions" on page 4-29 for a summary of the I-Unit GR/AR opcode extensions.

4.3.7.1 Move to AR – Register (I-Unit)

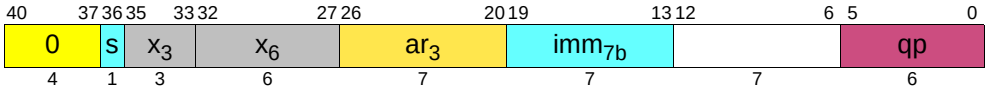
I26



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$ar_3 = r_2$	0	0	2A

4.3.7.2 Move to AR – Immediate$_8$ (I-Unit)

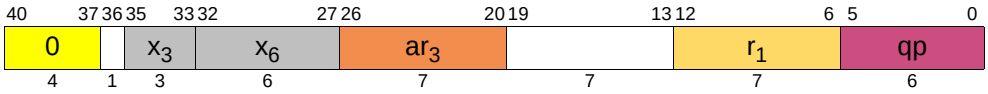
I27



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$ar_3 = imm_8$	0	0	0A

4.3.7.3 Move from AR (I-Unit)

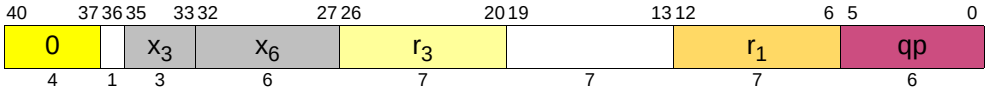
I28



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$r_1 = ar_3$	0	0	32

4.3.8 Sign/Zero Extend/Compute Zero Index

I29



Instruction	Operands	Opcode	Extension	
			x_3	x_6
zxt1	$r_1 = r_3$	0	0	10
zxt2				11
zxt4				12
sxt1				14
sxt2				15
sxt4				16
czx1.l				18
czx2.l				19
czx1.r				1C
czx2.r				1D

4.4 M-Unit Instruction Encodings

4.4.1 Loads and Stores

All load and store instructions are encoded within major opcodes 4, 5, 6, and 7 using a 6-bit opcode extension field in bits 35:30 (x_6). Instructions in major opcode 4 (integer load/store, semaphores, and get FR) use two 1-bit opcode extension fields in bit 36 (m) and bit 27 (x) as shown in Table 4-27. Instructions in major opcode 6 (floating-point load/store, load pair, and set FR) use two 1-bit opcode extension fields in bit 36 (m) and bit 27 (x) as shown in Table 4-28.

Table 4-27. Integer Load/Store/Semaphore/Get FR 1-bit Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	
4	0	0	Load/Store (Table 4-29)
	0	1	Semaphore/get FR (Table 4-32)
	1	0	Load +Reg (Table 4-30)
	1	1	

Table 4-28. Floating-point Load/Store/Load Pair/Set FR 1-bit Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	
6	0	0	FP Load/Store (Table 4-33)
	0	1	FP Load Pair/set FR (Table 4-36)
	1	0	FP Load +Reg (Table 4-34)
	1	1	FP Load Pair +Imm (Table 4-37)

The integer load/store opcode extensions are summarized in Table 4-29 on page 4-35, Table 4-30 on page 4-35, and Table 4-31 on page 4-36, and the semaphore and get FR opcode extensions in Table 4-32 on page 4-36. The floating-point load/store opcode extensions are summarized in Table 4-33 on page 4-37, Table 4-34 on page 4-37, and Table 4-35 on page 4-38, the floating-point load pair and set FR opcode extensions in Table 4-36 on page 4-38 and Table 4-37 on page 4-39.

Table 4-29. Integer Load/Store Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	0	0	ld1 M1	ld2 M1	ld4 M1	ld8 M1
			1	ld1.s M1	ld2.s M1	ld4.s M1	ld8.s M1
			2	ld1.a M1	ld2.a M1	ld4.a M1	ld8.a M1
			3	ld1.sa M1	ld2.sa M1	ld4.sa M1	ld8.sa M1
			4	ld1.bias M1	ld2.bias M1	ld4.bias M1	ld8.bias M1
			5	ld1.acq M1	ld2.acq M1	ld4.acq M1	ld8.acq M1
			6				ld8.fill M1
			7				
			8	ld1.c.clr M1	ld2.c.clr M1	ld4.c.clr M1	ld8.c.clr M1
			9	ld1.c.nc M1	ld2.c.nc M1	ld4.c.nc M1	ld8.c.nc M1
			A	ld1.c.clr.acq M1	ld2.c.clr.acq M1	ld4.c.clr.acq M1	ld8.c.clr.acq M1
			B				
			C	st1 M4	st2 M4	st4 M4	st8 M4
			D	st1.rel M4	st2.rel M4	st4.rel M4	st8.rel M4
			E				st8.spill M4
			F				

Table 4-30. Integer Load +Reg Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	1	0	0	ld1 M2	ld2 M2	ld4 M2	ld8 M2
			1	ld1.s M2	ld2.s M2	ld4.s M2	ld8.s M2
			2	ld1.a M2	ld2.a M2	ld4.a M2	ld8.a M2
			3	ld1.sa M2	ld2.sa M2	ld4.sa M2	ld8.sa M2
			4	ld1.bias M2	ld2.bias M2	ld4.bias M2	ld8.bias M2
			5	ld1.acq M2	ld2.acq M2	ld4.acq M2	ld8.acq M2
			6				ld8.fill M2
			7				
			8	ld1.c.clr M2	ld2.c.clr M2	ld4.c.clr M2	ld8.c.clr M2
			9	ld1.c.nc M2	ld2.c.nc M2	ld4.c.nc M2	ld8.c.nc M2
			A	ld1.c.clr.acq M2	ld2.c.clr.acq M2	ld4.c.clr.acq M2	ld8.c.clr.acq M2
			B				
			C				
			D				
			E				
			F				

Table 4-31. Integer Load/Store +Imm Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 35:32	Bits 31:30			
		0	1	2	3
5	0	ld1 M3	ld2 M3	ld4 M3	ld8 M3
	1	ld1.s M3	ld2.s M3	ld4.s M3	ld8.s M3
	2	ld1.a M3	ld2.a M3	ld4.a M3	ld8.a M3
	3	ld1.sa M3	ld2.sa M3	ld4.sa M3	ld8.sa M3
	4	ld1.bias M3	ld2.bias M3	ld4.bias M3	ld8.bias M3
	5	ld1.acq M3	ld2.acq M3	ld4.acq M3	ld8.acq M3
	6				ld8.fill M3
	7				
	8	ld1.c.clr M3	ld2.c.clr M3	ld4.c.clr M3	ld8.c.clr M3
	9	ld1.c.nc M3	ld2.c.nc M3	ld4.c.nc M3	ld8.c.nc M3
	A	ld1.c.clr.acq M3	ld2.c.clr.acq M3	ld4.c.clr.acq M3	ld8.c.clr.acq M3
	B				
	C	st1 M5	st2 M5	st4 M5	st8 M5
	D	st1.rel M5	st2.rel M5	st4.rel M5	st8.rel M5
	E				st8.spill M5
	F				

Table 4-32. Semaphore/Get FR Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	1	0	cmpxchg1.acq M16	cmpxchg2.acq M16	cmpxchg4.acq M16	cmpxchg8.acq M16
			1	cmpxchg1.rel M16	cmpxchg2.rel M16	cmpxchg4.rel M16	cmpxchg8.rel M16
			2	xchg1 M16	xchg2 M16	xchg4 M16	xchg8 M16
			3				
			4			fetchadd4.acq M17	fetchadd8.acq M17
			5			fetchadd4.rel M17	fetchadd8.rel M17
			6				
			7	getf.sig M19	getf.exp M19	getf.s M19	getf.d M19
			8				
			9				
			A				
			B				
			C				
			D				
			E				
			F				

Table 4-33. Floating-point Load/Store/Lfetch Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	0	0	0	ldfe M6	ldf8 M6	ldfs M6	ldfd M6
			1	ldfe.s M6	ldf8.s M6	ldfs.s M6	ldfd.s M6
			2	ldfe.a M6	ldf8.a M6	ldfs.a M6	ldfd.a M6
			3	ldfe.sa M6	ldf8.sa M6	ldfs.sa M6	ldfd.sa M6
			4				
			5				
			6				ldf.fill M6
			7				
			8	ldfe.c.clr M6	ldf8.c.clr M6	ldfs.c.clr M6	ldfd.c.clr M6
			9	ldfe.c.nc M6	ldf8.c.nc M6	ldfs.c.nc M6	ldfd.c.nc M6
			A				
			B	lfetch M13	lfetch.excl M13	lfetch.fault M13	lfetch.fault.excl M13
			C	stfe M9	stf8 M9	stfs M9	stfd M9
			D				
			E				stf.spill M9
			F				

Table 4-34. Floating-point Load/Lfetch +Reg Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	1	0	0	ldfe M7	ldf8 M7	ldfs M7	ldfd M7
			1	ldfe.s M7	ldf8.s M7	ldfs.s M7	ldfd.s M7
			2	ldfe.a M7	ldf8.a M7	ldfs.a M7	ldfd.a M7
			3	ldfe.sa M7	ldf8.sa M7	ldfs.sa M7	ldfd.sa M7
			4				
			5				
			6				ldf.fill M7
			7				
			8	ldfe.c.clr M7	ldf8.c.clr M7	ldfs.c.clr M7	ldfd.c.clr M7
			9	ldfe.c.nc M7	ldf8.c.nc M7	ldfs.c.nc M7	ldfd.c.nc M7
			A				
			B	lfetch M14	lfetch.excl M14	lfetch.fault M14	lfetch.fault.excl M14
			C				
			D				
			E				
			F				

Table 4-35. Floating-point Load/Store/Lfetch +Imm Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 35:32	Bits 31:30			
		0	1	2	3
7	0	ldfe M8	ldf8 M8	ldfs M8	ldfd M8
	1	ldfe.s M8	ldf8.s M8	ldfs.s M8	ldfd.s M8
	2	ldfe.a M8	ldf8.a M8	ldfs.a M8	ldfd.a M8
	3	ldfe.sa M8	ldf8.sa M8	ldfs.sa M8	ldfd.sa M8
	4				
	5				
	6				ldf.fill M8
	7				
	8	ldfe.c.clr M8	ldf8.c.clr M8	ldfs.c.clr M8	ldfd.c.clr M8
	9	ldfe.c.nc M8	ldf8.c.nc M8	ldfs.c.nc M8	ldfd.c.nc M8
	A				
	B	lfetch M15	lfetch.excl M15	lfetch.fault M15	lfetch.fault.excl M15
	C	stfe M10	stf8 M10	stfs M10	stfd M10
	D				
	E				stf.spill M10
	F				

Table 4-36. Floating-point Load Pair/Set FR Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	0	1	0		ldfp8 M11	ldfps M11	ldfpd M11
			1		ldfp8.s M11	ldfps.s M11	ldfpd.s M11
			2		ldfp8.a M11	ldfps.a M11	ldfpd.a M11
			3		ldfp8.sa M11	ldfps.sa M11	ldfpd.sa M11
			4				
			5				
			6				
			7	setf.sig M18	setf.exp M18	setf.s M18	setf.d M18
			8		ldfp8.c.clr M11	ldfps.c.clr M11	ldfpd.c.clr M11
			9		ldfp8.c.nc M11	ldfps.c.nc M11	ldfpd.c.nc M11
			A				
			B				
			C				
			D				
			E				
			F				

Table 4-37. Floating-point Load Pair +Imm Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	1	1	0		ldfp8 M12	ldfps M12	ldfpd M12
			1		ldfp8.s M12	ldfps.s M12	ldfpd.s M12
			2		ldfp8.a M12	ldfps.a M12	ldfpd.a M12
			3		ldfp8.sa M12	ldfps.sa M12	ldfpd.sa M12
			4				
			5				
			6				
			7				
			8		ldfp8.c.clr M12	ldfps.c.clr M12	ldfpd.c.clr M12
			9		ldfp8.c.nc M12	ldfps.c.nc M12	ldfpd.c.nc M12
			A				
			B				
			C				
			D				
			E				
			F				

The load and store instructions all have a 2-bit opcode extension field in bits 29:28 (hint) which encodes locality hint information. Table 4-38 and Table 4-39 summarize these assignments.

Table 4-38. Load Hint Completer

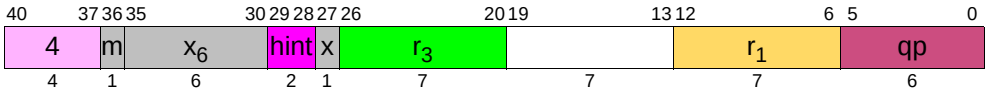
hint Bits 29:28	*ldhint*
0	*none*
1	.nt1
2	
3	.nta

Table 4-39. Store Hint Completer

hint Bits 29:28	*sthint*
0	*none*
1	
2	
3	.nta

4.4.1.1 Integer Load

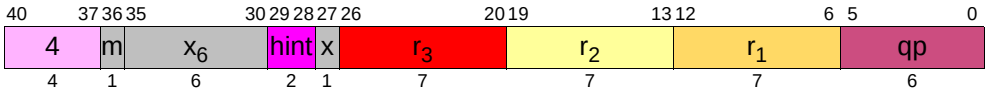
M1



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ld1.*ldhint*	$r_1 = [r_3]$	4	0	0	00	See Table 4-38 on page 4-39
ld2.*ldhint*					01	
ld4.*ldhint*					02	
ld8.*ldhint*					03	
ld1.s.*ldhint*					04	
ld2.s.*ldhint*					05	
ld4.s.*ldhint*					06	
ld8.s.*ldhint*					07	
ld1.a.*ldhint*					08	
ld2.a.*ldhint*					09	
ld4.a.*ldhint*					0A	
ld8.a.*ldhint*					0B	
ld1.sa.*ldhint*					0C	
ld2.sa.*ldhint*					0D	
ld4.sa.*ldhint*					0E	
ld8.sa.*ldhint*					0F	
ld1.bias.*ldhint*					10	
ld2.bias.*ldhint*					11	
ld4.bias.*ldhint*					12	
ld8.bias.*ldhint*					13	
ld1.acq.*ldhint*					14	
ld2.acq.*ldhint*					15	
ld4.acq.*ldhint*					16	
ld8.acq.*ldhint*					17	
ld8.fill.*ldhint*					1B	
ld1.c.clr.*ldhint*					20	
ld2.c.clr.*ldhint*					21	
ld4.c.clr.*ldhint*					22	
ld8.c.clr.*ldhint*					23	
ld1.c.nc.*ldhint*					24	
ld2.c.nc.*ldhint*					25	
ld4.c.nc.*ldhint*					26	
ld8.c.nc.*ldhint*					27	
ld1.c.clr.acq.*ldhint*					28	
ld2.c.clr.acq.*ldhint*					29	
ld4.c.clr.acq.*ldhint*					2A	
ld8.c.clr.acq.*ldhint*					2B	

4.4.1.2 Integer Load – Increment by Register

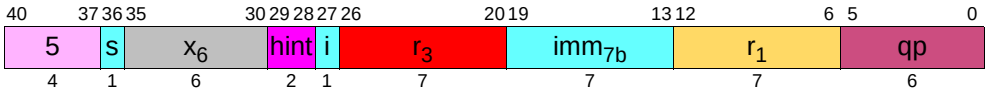
M2



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ld1.*ldhint*					00	
ld2.*ldhint*					01	
ld4.*ldhint*					02	
ld8.*ldhint*					03	
ld1.s.*ldhint*					04	
ld2.s.*ldhint*					05	
ld4.s.*ldhint*					06	
ld8.s.*ldhint*					07	
ld1.a.*ldhint*					08	
ld2.a.*ldhint*					09	
ld4.a.*ldhint*					0A	
ld8.a.*ldhint*					0B	
ld1.sa.*ldhint*					0C	
ld2.sa.*ldhint*					0D	
ld4.sa.*ldhint*					0E	
ld8.sa.*ldhint*					0F	
ld1.bias.*ldhint*					10	
ld2.bias.*ldhint*					11	See
ld4.bias.*ldhint*	$r_1 = [r_3], r_2$	4	1	0	12	Table 4-38 on page 4-39
ld8.bias.*ldhint*					13	
ld1.acq.*ldhint*					14	
ld2.acq.*ldhint*					15	
ld4.acq.*ldhint*					16	
ld8.acq.*ldhint*					17	
ld8.fill.*ldhint*					1B	
ld1.c.clr.*ldhint*					20	
ld2.c.clr.*ldhint*					21	
ld4.c.clr.*ldhint*					22	
ld8.c.clr.*ldhint*					23	
ld1.c.nc.*ldhint*					24	
ld2.c.nc.*ldhint*					25	
ld4.c.nc.*ldhint*					26	
ld8.c.nc.*ldhint*					27	
ld1.c.clr.acq.*ldhint*					28	
ld2.c.clr.acq.*ldhint*					29	
ld4.c.clr.acq.*ldhint*					2A	
ld8.c.clr.acq.*ldhint*					2B	

4.4.1.3 Integer Load – Increment by Immediate

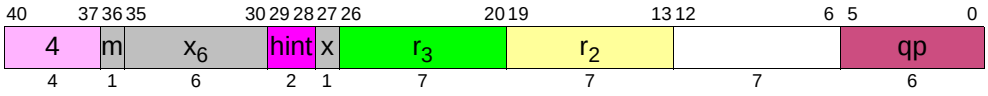
M3



Instruction	Operands	Opcode	Extension	
			x_6	hint
ld1.*ldhint*	$r_1 = [r_3]$, imm_9	5	00	See Table 4-38 on page 4-39
ld2.*ldhint*			01	
ld4.*ldhint*			02	
ld8.*ldhint*			03	
ld1.s.*ldhint*			04	
ld2.s.*ldhint*			05	
ld4.s.*ldhint*			06	
ld8.s.*ldhint*			07	
ld1.a.*ldhint*			08	
ld2.a.*ldhint*			09	
ld4.a.*ldhint*			0A	
ld8.a.*ldhint*			0B	
ld1.sa.*ldhint*			0C	
ld2.sa.*ldhint*			0D	
ld4.sa.*ldhint*			0E	
ld8.sa.*ldhint*			0F	
ld1.bias.*ldhint*			10	
ld2.bias.*ldhint*			11	
ld4.bias.*ldhint*			12	
ld8.bias.*ldhint*			13	
ld1.acq.*ldhint*			14	
ld2.acq.*ldhint*			15	
ld4.acq.*ldhint*			16	
ld8.acq.*ldhint*			17	
ld8.fill.*ldhint*			1B	
ld1.c.clr.*ldhint*			20	
ld2.c.clr.*ldhint*			21	
ld4.c.clr.*ldhint*			22	
ld8.c.clr.*ldhint*			23	
ld1.c.nc.*ldhint*			24	
ld2.c.nc.*ldhint*			25	
ld4.c.nc.*ldhint*			26	
ld8.c.nc.*ldhint*			27	
ld1.c.clr.acq.*ldhint*			28	
ld2.c.clr.acq.*ldhint*			29	
ld4.c.clr.acq.*ldhint*			2A	
ld8.c.clr.acq.*ldhint*			2B	

4.4.1.4 Integer Store

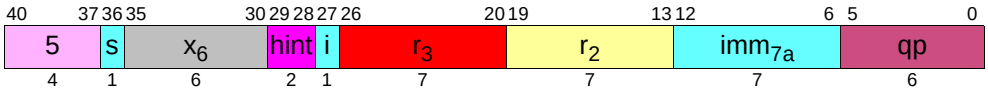
M4



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
st1.*sthint*	$[r_3] = r_2$	4	0	0	30	See Table 4-39 on page 4-39
st2.*sthint*					31	
st4.*sthint*					32	
st8.*sthint*					33	
st1.rel.*sthint*					34	
st2.rel.*sthint*					35	
st4.rel.*sthint*					36	
st8.rel.*sthint*					37	
st8.spill.*sthint*					3B	

4.4.1.5 Integer Store – Increment by Immediate

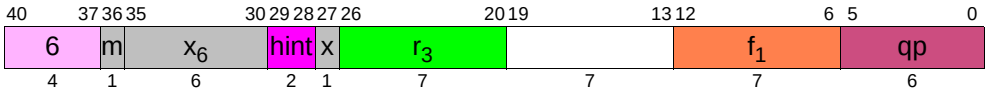
M5



Instruction	Operands	Opcode	Extension	
			x_6	hint
st1.*sthint*	$[r_3] = r_2, imm_9$	5	30	See Table 4-39 on page 4-39
st2.*sthint*			31	
st4.*sthint*			32	
st8.*sthint*			33	
st1.rel.*sthint*			34	
st2.rel.*sthint*			35	
st4.rel.*sthint*			36	
st8.rel.*sthint*			37	
st8.spill.*sthint*			3B	

4.4.1.6 Floating-point Load

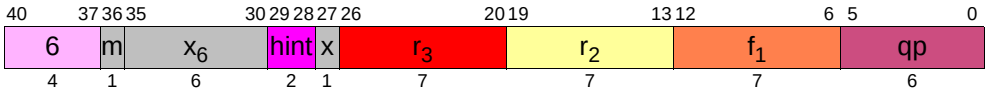
M6



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfs.*ldhint*	$f_1 = [r_3]$	6	0	0	02	See Table 4-38 on page 4-39
ldfd.*ldhint*					03	
ldf8.*ldhint*					01	
ldfe.*ldhint*					00	
ldfs.s.*ldhint*					06	
ldfd.s.*ldhint*					07	
ldf8.s.*ldhint*					05	
ldfe.s.*ldhint*					04	
ldfs.a.*ldhint*					0A	
ldfd.a.*ldhint*					0B	
ldf8.a.*ldhint*					09	
ldfe.a.*ldhint*					08	
ldfs.sa.*ldhint*					0E	
ldfd.sa.*ldhint*					0F	
ldf8.sa.*ldhint*					0D	
ldfe.sa.*ldhint*					0C	
ldf.fill.*ldhint*					1B	
ldfs.c.clr.*ldhint*					22	
ldfd.c.clr.*ldhint*					23	
ldf8.c.clr.*ldhint*					21	
ldfe.c.clr.*ldhint*					20	
ldfs.c.nc.*ldhint*					26	
ldfd.c.nc.*ldhint*					27	
ldf8.c.nc.*ldhint*					25	
ldfe.c.nc.*ldhint*					24	

4.4.1.7 Floating-point Load – Increment by Register

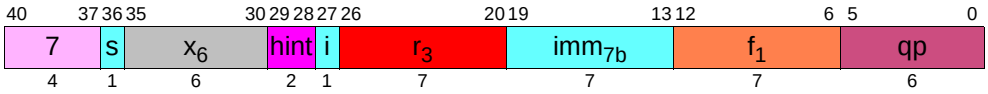
M7



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfs.*ldhint*	$f_1 = [r_3], r_2$	6	1	0	02	See Table 4-38 on page 4-39
ldfd.*ldhint*					03	
ldf8.*ldhint*					01	
ldfe.*ldhint*					00	
ldfs.s.*ldhint*					06	
ldfd.s.*ldhint*					07	
ldf8.s.*ldhint*					05	
ldfe.s.*ldhint*					04	
ldfs.a.*ldhint*					0A	
ldfd.a.*ldhint*					0B	
ldf8.a.*ldhint*					09	
ldfe.a.*ldhint*					08	
ldfs.sa.*ldhint*					0E	
ldfd.sa.*ldhint*					0F	
ldf8.sa.*ldhint*					0D	
ldfe.sa.*ldhint*					0C	
ldf.fill.*ldhint*					1B	
ldfs.c.clr.*ldhint*					22	
ldfd.c.clr.*ldhint*					23	
ldf8.c.clr.*ldhint*					21	
ldfe.c.clr.*ldhint*					20	
ldfs.c.nc.*ldhint*					26	
ldfd.c.nc.*ldhint*					27	
ldf8.c.nc.*ldhint*					25	
ldfe.c.nc.*ldhint*					24	

4.4.1.8 Floating-point Load – Increment by Immediate

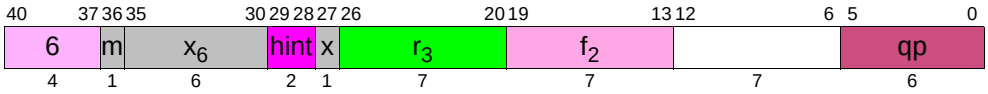
M8



Instruction	Operands	Opcode	Extension	
			x_6	hint
ldfs.*ldhint*	$f_1 = [r_3]$, imm_9	7	02	See Table 4-38 on page 4-39
ldfd.*ldhint*			03	
ldf8.*ldhint*			01	
ldfe.*ldhint*			00	
ldfs.s.*ldhint*			06	
ldfd.s.*ldhint*			07	
ldf8.s.*ldhint*			05	
ldfe.s.*ldhint*			04	
ldfs.a.*ldhint*			0A	
ldfd.a.*ldhint*			0B	
ldf8.a.*ldhint*			09	
ldfe.a.*ldhint*			08	
ldfs.sa.*ldhint*			0E	
ldfd.sa.*ldhint*			0F	
ldf8.sa.*ldhint*			0D	
ldfe.sa.*ldhint*			0C	
ldf.fill.*ldhint*			1B	
ldfs.c.clr.*ldhint*			22	
ldfd.c.clr.*ldhint*			23	
ldf8.c.clr.*ldhint*			21	
ldfe.c.clr.*ldhint*			20	
ldfs.c.nc.*ldhint*			26	
ldfd.c.nc.*ldhint*			27	
ldf8.c.nc.*ldhint*			25	
ldfe.c.nc.*ldhint*			24	

4.4.1.9 Floating-point Store

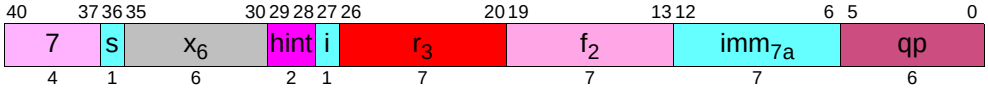
M9



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
stfs.*sthint*	$[r_3] = f_2$	6	0	0	32	See Table 4-39 on page 4-39
stfd.*sthint*					33	
stf8.*sthint*					31	
stfe.*sthint*					30	
stf.spill.*sthint*					3B	

4.4.1.10 Floating-point Store – Increment by Immediate

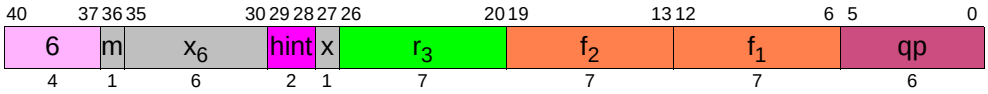
M10



Instruction	Operands	Opcode	Extension	
			x_6	hint
stfs.*sthint*	$[r_3] = f_2, imm_9$	7	32	See Table 4-39 on page 4-39
stfd.*sthint*			33	
stf8.*sthint*			31	
stfe.*sthint*			30	
stf.spill.*sthint*			3B	

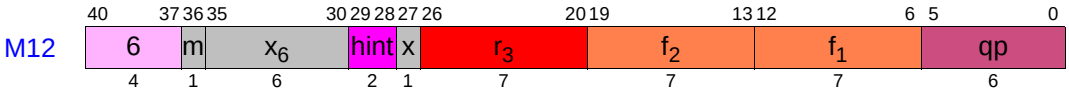
4.4.1.11 Floating-point Load Pair

M11



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfps.*ldhint*	$f_1, f_2 = [r_3]$	6	0	1	02	See Table 4-38 on page 4-39
ldfpd.*ldhint*					03	
ldfp8.*ldhint*					01	
ldfps.s.*ldhint*					06	
ldfpd.s.*ldhint*					07	
ldfp8.s.*ldhint*					05	
ldfps.a.*ldhint*					0A	
ldfpd.a.*ldhint*					0B	
ldfp8.a.*ldhint*					09	
ldfps.sa.*ldhint*					0E	
ldfpd.sa.*ldhint*					0F	
ldfp8.sa.*ldhint*					0D	
ldfps.c.clr.*ldhint*					22	
ldfpd.c.clr.*ldhint*					23	
ldfp8.c.clr.*ldhint*					21	
ldfps.c.nc.*ldhint*					26	
ldfpd.c.nc.*ldhint*					27	
ldfp8.c.nc.*ldhint*					25	

4.4.1.12 Floating-point Load Pair – Increment by Immediate



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfps.*ldhint*	$f_1, f_2 = [r_3], 8$	6	1	1	02	See Table 4-38 on page 4-39
ldfpd.*ldhint*	$f_1, f_2 = [r_3], 16$				03	
ldfp8.*ldhint*					01	
ldfps.s.*ldhint*	$f_1, f_2 = [r_3], 8$				06	
ldfpd.s.*ldhint*	$f_1, f_2 = [r_3], 16$				07	
ldfp8.s.*ldhint*					05	
ldfps.a.*ldhint*	$f_1, f_2 = [r_3], 8$				0A	
ldfpd.a.*ldhint*	$f_1, f_2 = [r_3], 16$				0B	
ldfp8.a.*ldhint*					09	
ldfps.sa.*ldhint*	$f_1, f_2 = [r_3], 8$				0E	
ldfpd.sa.*ldhint*	$f_1, f_2 = [r_3], 16$				0F	
ldfp8.sa.*ldhint*					0D	
ldfps.c.clr.*ldhint*	$f_1, f_2 = [r_3], 8$				22	
ldfpd.c.clr.*ldhint*	$f_1, f_2 = [r_3], 16$				23	
ldfp8.c.clr.*ldhint*					21	
ldfps.c.nc.*ldhint*	$f_1, f_2 = [r_3], 8$				26	
ldfpd.c.nc.*ldhint*	$f_1, f_2 = [r_3], 16$				27	
ldfp8.c.nc.*ldhint*					25	

4.4.2 Line Prefetch

The line prefetch instructions are encoded in major opcodes 6 and 7 along with the floating-point load/store instructions. See "Loads and Stores" on page 4-34 for a summary of the opcode extensions.

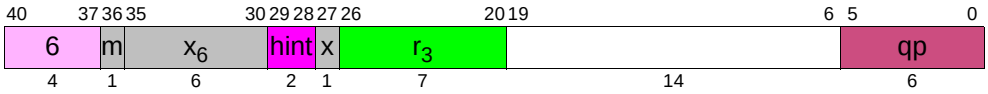
The line prefetch instructions all have a 2-bit opcode extension field in bits 29:28 (hint) which encodes locality hint information as shown in Table 4-40.

Table 4-40. Line Prefetch Hint Completer

hint Bits 29:28	*lfhint*
0	*none*
1	.nt1
2	.nt2
3	.nta

4.4.2.1 Line Prefetch

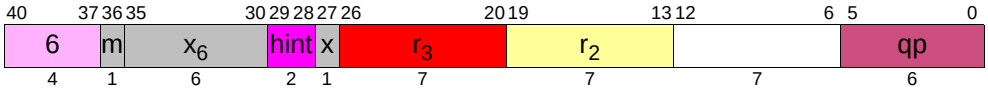
M13



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
lfetch.*lfhint* lfetch.excl.*lfhint* lfetch.fault.*lfhint* lfetch.fault.excl.*lfhint*	[r_3]	6	0	0	2C 2D 2E 2F	See Table 4-40 on page 4-49

4.4.2.2 Line Prefetch – Increment by Register

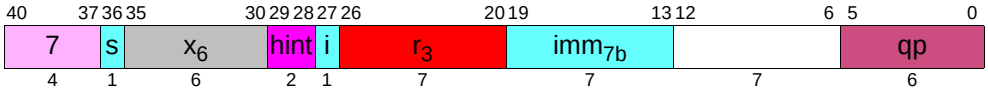
M14



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
lfetch.*lfhint* lfetch.excl.*lfhint* lfetch.fault.*lfhint* lfetch.fault.excl.*lfhint*	[r_3], r_2	6	1	0	2C 2D 2E 2F	See Table 4-40 on page 4-49

4.4.2.3 Line Prefetch – Increment by Immediate

M15



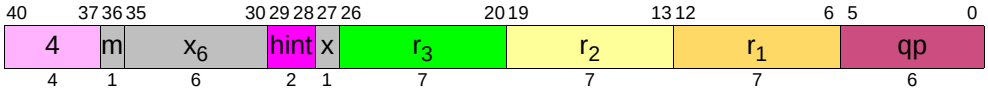
Instruction	Operands	Opcode	Extension	
			x_6	hint
lfetch.*lfhint* lfetch.excl.*lfhint* lfetch.fault.*lfhint* lfetch.fault.excl.*lfhint*	[r_3], imm_9	7	2C 2D 2E 2F	See Table 4-40 on page 4-49

4.4.3 Semaphores

The semaphore instructions are encoded in major opcode 4 along with the integer load/store instructions. See "Loads and Stores" on page 4-34 for a summary of the opcode extensions.

4.4.3.1 Exchange/Compare and Exchange

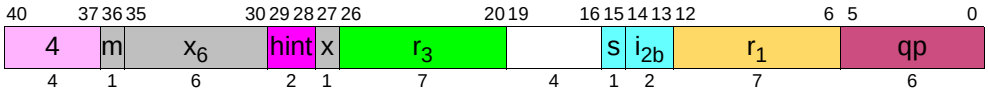
M16



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
cmpxchg1.acq.*ldhint*	r_1 = [r_3], r_2, ar.ccv	4	0	1	00	See Table 4-38 on page 4-39
cmpxchg2.acq.*ldhint*					01	
cmpxchg4.acq.*ldhint*					02	
cmpxchg8.acq.*ldhint*					03	
cmpxchg1.rel.*ldhint*					04	
cmpxchg2.rel.*ldhint*					05	
cmpxchg4.rel.*ldhint*					06	
cmpxchg8.rel.*ldhint*					07	
xchg1.*ldhint*	r_1 = [r_3], r_2				08	
xchg2.*ldhint*					09	
xchg4.*ldhint*					0A	
xchg8.*ldhint*					0B	

4.4.3.2 Fetch and Add – Immediate

M17



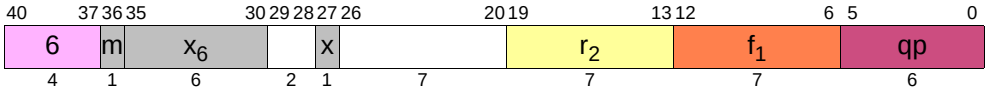
Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
fetchadd4.acq.*ldhint*	r_1 = [r_3], inc_3	4	0	1	12	See Table 4-38 on page 4-39
fetchadd8.acq.*ldhint*					13	
fetchadd4.rel.*ldhint*					16	
fetchadd8.rel.*ldhint*					17	

4.4.4 Set/Get FR

The set FR instructions are encoded in major opcode 6 along with the floating-point load/store instructions. The get FR instructions are encoded in major opcode 4 along with the integer load/store instructions. See "Loads and Stores" on page 4-34 for a summary of the opcode extensions.

4.4.4.1 Set FR

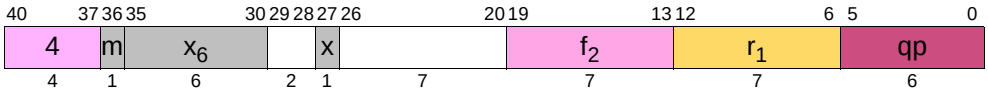
M18



Instruction	Operands	Opcode	Extension		
			m	x	x_6
setf.sig	$f_1 = r_2$	6	0	1	1C
setf.exp					1D
setf.s					1E
setf.d					1F

4.4.4.2 Get FR

M19

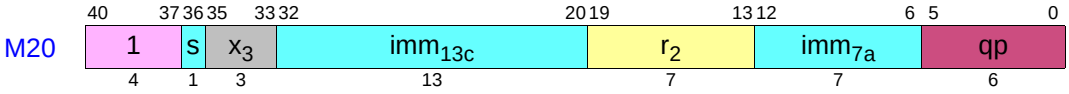


Instruction	Operands	Opcode	Extension		
			m	x	x_6
getf.sig	$r_1 = f_2$	4	0	1	1C
getf.exp					1D
getf.s					1E
getf.d					1F

4.4.5 Speculation and Advanced Load Checks

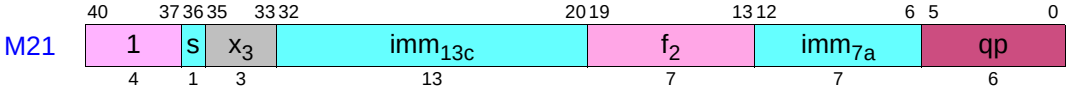
The speculation and advanced load check instructions are encoded in major opcodes 0 and 1 along with the system/memory management instructions. See "System/Memory Management" on page 4-58 for a summary of the opcode extensions.

4.4.5.1 Integer Speculation Check (M-Unit)



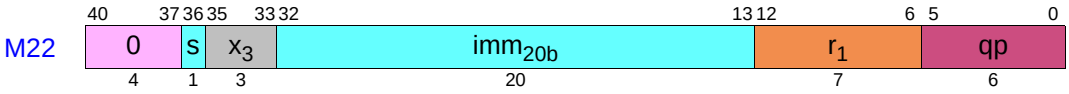
Instruction	Operands	Opcode	Extension
			x_3
chk.s.m	r_2, $target_{25}$	1	1

4.4.5.2 Floating-point Speculation Check



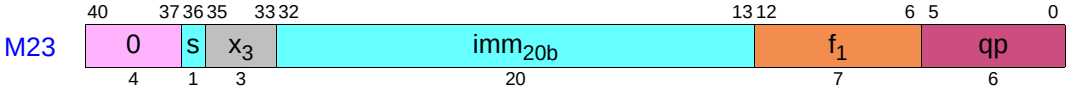
Instruction	Operands	Opcode	Extension
			x_3
chk.s	f_2, $target_{25}$	1	3

4.4.5.3 Integer Advanced Load Check



Instruction	Operands	Opcode	Extension
			x_3
chk.a.nc	r_1, $target_{25}$	0	4
chk.a.clr			5

4.4.5.4 Floating-point Advanced Load Check

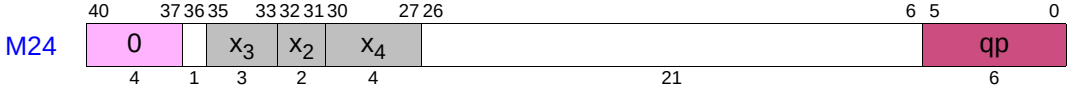


Instruction	Operands	Opcode	Extension
			x_3
chk.a.nc	f_1, $target_{25}$	0	6
chk.a.clr			7

4.4.6 Cache/Synchronization/RSE/ALAT

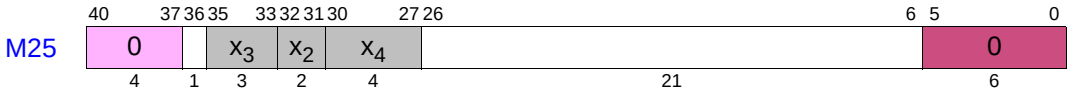
The cache/synchronization/RSE/ALAT instructions are encoded in major opcode 0 along with the memory management instructions. See "System/Memory Management" on page 4-58 for a summary of the opcode extensions.

4.4.6.1 Sync/Fence/Serialize/ALAT Control



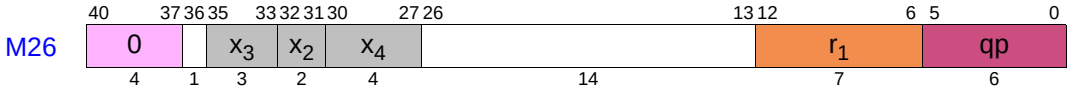
Instruction	Opcode	Extension		
		x_3	x_4	x_2
invala	0	0	0	1
fwb			0	2
mf			2	
mf.a			3	
srlz.d			0	3
srlz.i			1	
sync.i			3	

4.4.6.2 RSE Control



Instruction	Opcode	Extension		
		x_3	x_4	x_2
flushrs [f]	0	0	C	0
loadrs [f]			A	

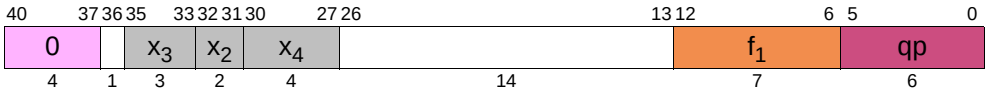
4.4.6.3 Integer ALAT Entry Invalidate



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
invala.e	r_1	0	0	2	1

4.4.6.4 Floating-point ALAT Entry Invalidate

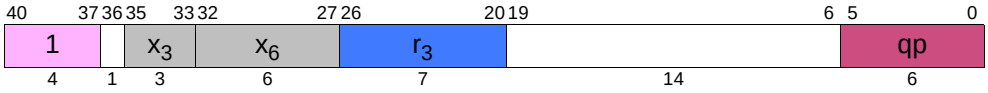
M27



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
invala.e	f_1	0	0	3	1

4.4.6.5 Flush Cache/Purge Translation Cache Entry

M28



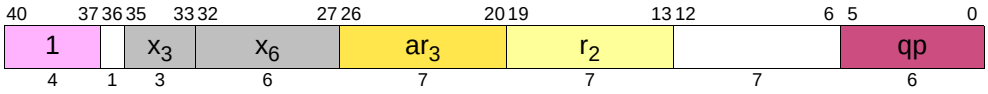
Instruction	Operands	Opcode	Extension	
			x_3	x_6
fc ptc.e [p]	r_3	1	0	30 34

4.4.7 GR/AR Moves (M-Unit)

The M-Unit GR/AR move instructions are encoded in major opcode 0 along with the system/memory management instructions. (Some ARs are accessed using system control instructions on the I-unit. See "GR/AR Moves (I-Unit)" on page 4-32.) See "System/Memory Management" on page 4-58 for a summary of the M-Unit GR/AR opcode extensions.

4.4.7.1 Move to AR – Register (M-Unit)

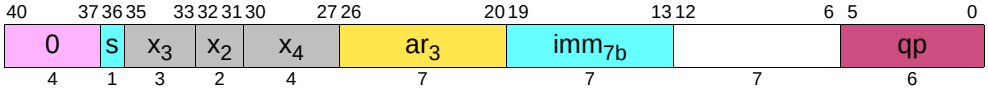
M29



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.m	$ar_3 = r_2$	1	0	2A

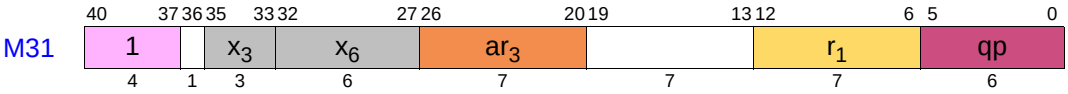
4.4.7.2 Move to AR – Immediate$_8$ (M-Unit)

M30



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
mov.m	$ar_3 = imm_8$	0	0	8	2

4.4.7.3 Move from AR (M-Unit)

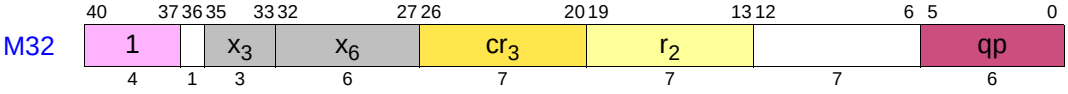


Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.m	$r_1 = ar_3$	1	0	22

4.4.8 GR/CR Moves

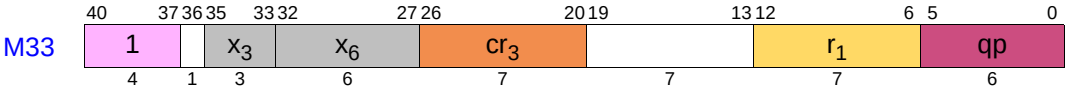
The GR/CR move instructions are encoded in major opcode 0 along with the system/memory management instructions. See "System/Memory Management" on page 4-58 for a summary of the opcode extensions.

4.4.8.1 Move to CR



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov ^{p}	$cr_3 = r_2$	1	0	2C

4.4.8.2 Move from CR

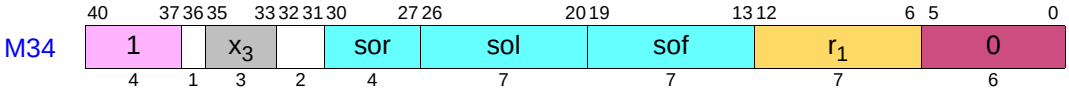


Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov ^{p}	$r_1 = cr_3$	1	0	24

4.4.9 Miscellaneous M-Unit Instructions

The miscellaneous M-unit instructions are encoded in major opcode 0 along with the system/memory management instructions. See "System/Memory Management" on page 4-58 for a summary of the opcode extensions.

4.4.9.1 Allocate Register Stack Frame

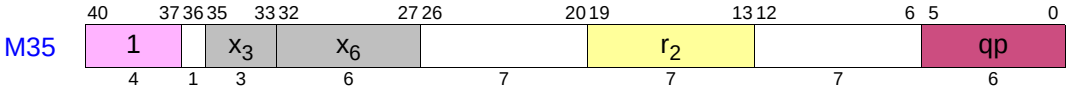


Instruction	Operands	Opcode	Extension
			x_3
alloc [f]	r_1 = ar.pfs, *i*, *l*, *o*, *r*	1	6

Note: The three immediates in the instruction encoding are formed from the operands as follows:

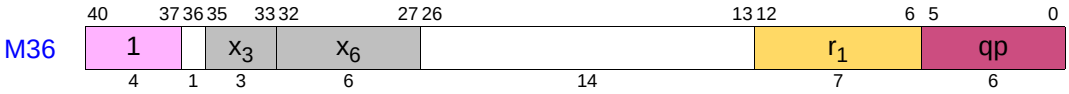
sof = *i* + *l* + *o*
sol = *i* + *l*
sor = *r* >> 3

4.4.9.2 Move to PSR



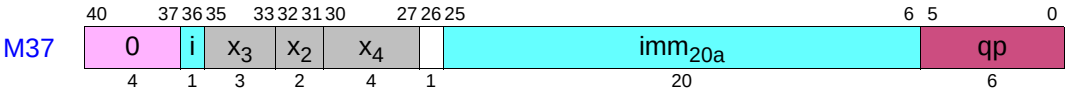
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	psr.l = r_2	1	0	2D
mov	psr.um = r_2			29

4.4.9.3 Move from PSR



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	r_1 = psr	1	0	25
mov	r_1 = psr.um			21

4.4.9.4 Break/Nop (M-Unit)



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
break.m nop.m	imm_{21}	0	0	0 1	0

4.4.10 System/Memory Management

All system/memory management instructions are encoded within major opcodes 0 and 1 using a 3-bit opcode extension field (x_3) in bits 35:33. Some instructions also have a 4-bit opcode extension field (x_4) in bits 30:27, or a 6-bit opcode extension field (x_6) in bits 32:27. Most of the instructions having a 4-bit opcode extension field also have a 2-bit extension field (x_2) in bits 32:31. Table 4-41 shows the 3-bit assignments for opcode 0, Table 4-42 summarizes the 4-bit+2-bit assignments for opcode 0, Table 4-43 shows the 3-bit assignments for opcode 1, and Table 4-44 summarizes the 6-bit assignments for opcode 1.

Table 4-41. Opcode 0 System/Memory Management 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	System/Memory Management 4-bit+2-bit Ext (Table 4-42)
	1	
	2	
	3	
	4	chk.a.nc – int M22
	5	chk.a.clr – int M22
	6	chk.a.nc – fp M23
	7	chk.a.clr – fp M23

Table 4-42. Opcode 0 System/Memory Management 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_4 Bits 30:27	x_2 Bits 32:31			
			0	1	2	3
0	0	0	break.m M37	invala M24	fwb M24	srlz.d M24
		1	nop.m M37			srlz.i M24
		2		invala.e – int M26	mf M24	
		3		invala.e – fp M27	mf.a M24	sync.i M24
		4	sum M44			
		5	rum M44			
		6	ssm M44			
		7	rsm M44			
		8			mov.m to ar – imm_8 M30	
		9				
		A	loadrs M25			
		B				
		C	flushrs M25			
		D				
		E				
		F				

Table 4-43. Opcode 1 System/Memory Management 3-bit Opcode Extensions

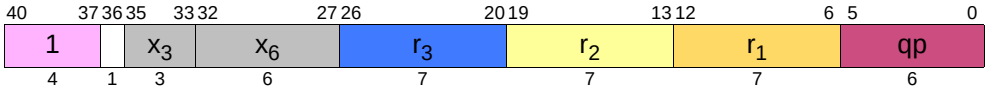
Opcode Bits 40:37	x_3 Bits 35:33	
1	0	System/Memory Management 6-bit Ext (Table 4-44)
	1	chk.s.m – int M20
	2	
	3	chk.s – fp M21
	4	
	5	
	6	alloc M34
	7	

Table 4-44. Opcode 1 System/Memory Management 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
1	0	0	mov to rr M42	mov from rr M43		fc M28
		1	mov to dbr M42	mov from dbr M43	mov from psr.um M36	probe.rw.fault – imm_2 M40
		2	mov to ibr M42	mov from ibr M43	mov.m from ar M31	probe.r.fault – imm_2 M40
		3	mov to pkr M42	mov from pkr M43		probe.w.fault – imm_2 M40
		4	mov to pmc M42	mov from pmc M43	mov from cr M33	ptc.e M28
		5	mov to pmd M42	mov from pmd M43	mov from psr M36	
		6				
		7		mov from cpuid M43		
		8		probe.r – imm_2 M39		probe.r M38
		9	ptc.l M45	probe.w – imm_2 M39	mov to psr.um M35	probe.w M38
		A	ptc.g M45	thash M46	mov.m to ar M29	
		B	ptc.ga M45	ttag M46		
		C	ptr.d M45		mov to cr M32	
		D	ptr.i M45		mov to psr.l M35	
		E	itr.d M42	tpa M46	itc.d M41	
		F	itr.i M42	tak M46	itc.i M41	

4.4.10.1 Probe – Register

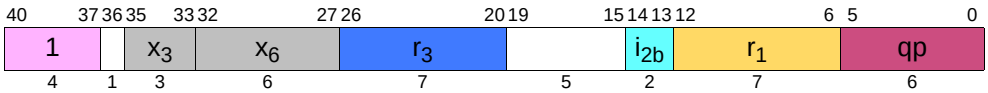
M38



Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.r probe.w	$r_1 = r_3, r_2$	1	0	38 39

4.4.10.2 Probe – Immediate$_2$

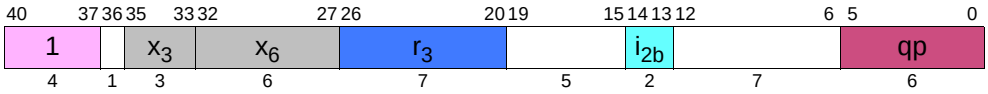
M39



Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.r probe.w	$r_1 = r_3, imm_2$	1	0	18 19

4.4.10.3 Probe Fault – Immediate$_2$

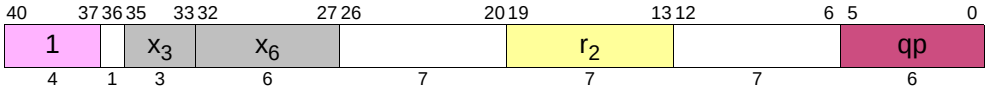
M40



Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.rw.fault	r_3, imm_2	1	0	31
probe.r.fault				32
probe.w.fault				33

4.4.10.4 Translation Cache Insert

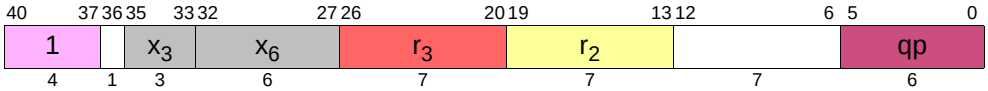
M41



Instruction	Operands	Opcode	Extension	
			x_3	x_6
itc.d [l p]	r_2	1	0	2E
itc.i [l p]				2F

4.4.10.5 Move to Indirect Register/Translation Register Insert

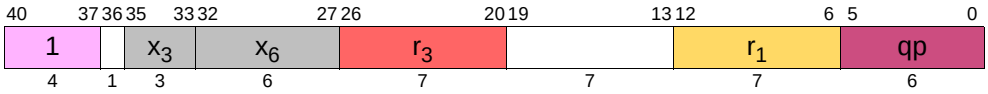
M42



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	rr[r_3] = r_2	1	0	00
	dbr[r_3] = r_2			01
	ibr[r_3] = r_2			02
	pkr[r_3] = r_2			03
	pmc[r_3] = r_2			04
	pmd[r_3] = r_2			05
itr.d [p]	dtr[r_3] = r_2			0E
itr.i [p]	itr[r_3] = r_2			0F

4.4.10.6 Move from Indirect Register

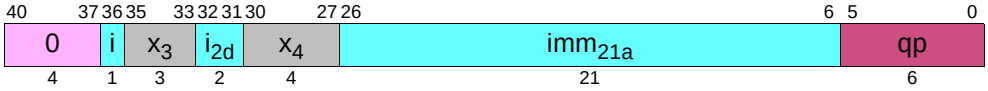
M43



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	r_1 = rr[r_3]	1	0	10
	r_1 = dbr[r_3]			11
	r_1 = ibr[r_3]			12
	r_1 = pkr[r_3]			13
	r_1 = pmc[r_3]			14
mov	r_1 = pmd[r_3]			15
	r_1 = cpuid[r_3]			17

4.4.10.7 Set/Reset User/System Mask

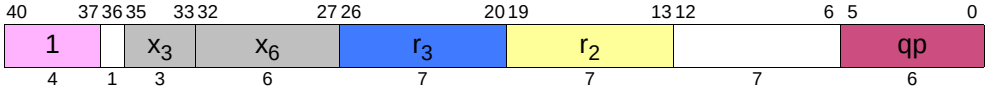
M44



Instruction	Operands	Opcode	Extension	
			x_3	x_4
sum	imm_{24}	0	0	4
rum				5
ssm [p]				6
rsm [p]				7

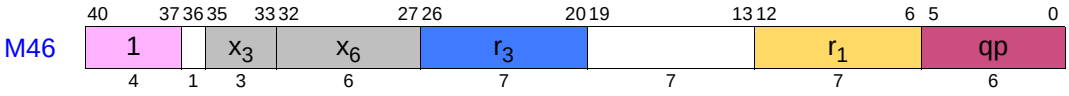
4.4.10.8 Translation Purge

M45



Instruction	Operands	Opcode	Extension	
			x_3	x_6
ptc.l [p]	r_3, r_2	1	0	09
ptc.g [l p]				0A
ptc.ga [l p]				0B
ptr.d [p]				0C
ptr.i [p]				0D

4.4.10.9 Translation Access



Instruction	Operands	Opcode	Extension	
			x_3	x_6
thash	$r_1 = r_3$	1	0	1A
ttag				1B
tpa [p]				1E
tak [p]				1F

4.5 B-Unit Instruction Encodings

The branch-unit includes branch, predict, and miscellaneous instructions.

4.5.1 Branches

Opcode 0 is used for indirect branch, opcode 1 for indirect call, opcode 4 for IP-relative branch, and opcode 5 for IP-relative call.

The IP-relative branch instructions encoded within major opcode 4 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-45.

Table 4-45. IP-relative Branch Types

Opcode Bits 40:37	btype Bits 8:6	
4	0	br.cond B1
	1	[e]
	2	br.wexit B1
	3	br.wtop B1
	4	[e]
	5	br.cloop B2
	6	br.cexit B2
	7	br.ctop B2

The indirect branch, indirect return, and miscellaneous branch-unit instructions are encoded within major opcode 0 using a 6-bit opcode extension field in bits 32:27 (x_6). Table 4-46 summarizes these assignments.

Table 4-46. Indirect/Miscellaneous Branch Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 30:27	Bits 32:31			
		0	1	2	3
0	0	break.b B9	epc B8	Indirect Branch (Table 4-47)	e
	1		e	Indirect Return (Table 4-48)	e
	2	cover B8	e	e	e
	3	e	e	e	e
	4	clrrrb B8	e	e	e
	5	clrrrb.pr B8	e	e	e
	6	e	e	e	e
	7	e	e	e	e
	8	rfi B8	e	e	e
	9		e	e	e
	A	e	e	e	e
	B	e	e	e	e
	C	bsw.0 B8	e	e	e
	D	bsw.1 B8	e	e	e
	E	e	e	e	e
	F	e	e	e	e

The indirect branch instructions encoded within major opcodes 0 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-47.

Table 4-47. Indirect Branch Types

Opcode Bits 40:37	x_6 Bits 32:27	btype Bits 8:6	
0	20	0	br.cond B4
		1	br.ia B4
		2	e
		3	e
		4	e
		5	e
		6	e
		7	e

The indirect return branch instructions encoded within major opcodes 0 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-48.

Table 4-48. Indirect Return Branch Types

Opcode Bits 40:37	x_6 Bits 32:27	btype Bits 8:6	
0	21	0	e
		1	e
		2	e
		3	e
		4	br.ret B4
		5	e
		6	e
		7	e

All of the branch instructions have a 1-bit opcode extension field, p, in bit 12 which provides a sequential prefetch hint. Table 4-49 summarizes these assignments.

Table 4-49. Sequential Prefetch Hint Completer

p Bit 12	*ph*
0	.few
1	.many

The IP-relative and indirect branch instructions all have a 2-bit opcode extension field in bits 34:33 (wh) which encodes branch prediction “whether” hint information as shown in Table 4-50. Indirect call instructions have a 3-bit opcode extension field in bits 34:32 (wh) for “whether” hint information as shown in Table 4-51.

Table 4-50. Branch Whether Hint Completer

wh Bits 34:33	*bwh*
0	.sptk
1	.spnt
2	.dptk
3	.dpnt

Table 4-51. Indirect Call Whether Hint Completer

wh Bits 34:32	*bwh*
0	
1	.sptk
2	
3	.spnt
4	
5	.dptk
6	
7	.dpnt

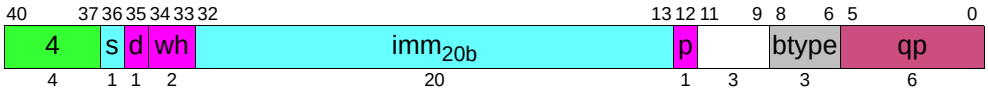
The branch instructions also have a 1-bit opcode extension field in bit 35 (d) which encodes a branch cache deallocation hint as shown in Table 4-52.

Table 4-52. Branch Cache Deallocation Hint Completer

d Bit 35	*dh*
0	*none*
1	.clr

4.5.1.1 IP-Relative Branch

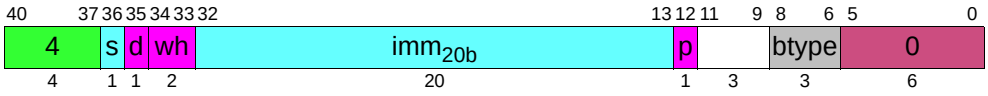
B1



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cond.*bwh.ph.dh* br.wexit.*bwh.ph.dh* br.wtop.*bwh.ph.dh*	*target*$_{25}$	4	0 2 3	See Table 4-49 on page 4-65	See Table 4-50 on page 4-65	See Table 4-52 on page 4-66

4.5.1.2 IP-Relative Counted Branch

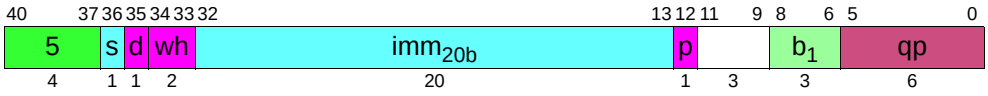
B2



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cloop.*bwh.ph.dh*e t br.cexit.*bwh.ph.dh*e t br.ctop.*bwh.ph.dh*e t	$target_{25}$	4	5 6 7	See Table 4-49 on page 4-65	See Table 4-50 on page 4-65	See Table 4-52 on page 4-66

4.5.1.3 IP-Relative Call

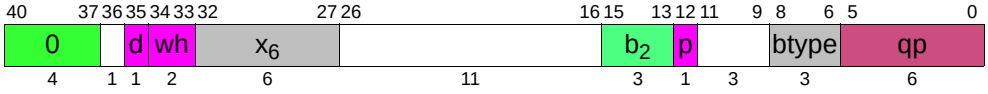
B3



Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.*bwh.ph.dh*e	b_1 = $target_{25}$	5	See Table 4-49 on page 4-65	See Table 4-50 on page 4-65	See Table 4-52 on page 4-66

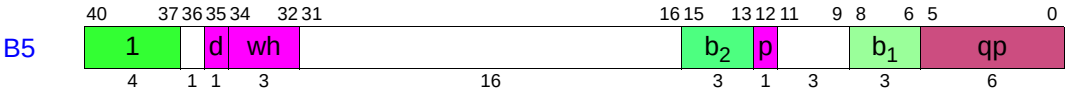
4.5.1.4 Indirect Branch

B4



Instruction	Operands	Opcode	Extension				
			x_6	btype	p	wh	d
br.cond.*bwh.ph.dh*e br.ia.*bwh.ph.dh*e	b_2	0	20	0 1	See Table 4-49 on page 4-65	See Table 4-50 on page 4-65	See Table 4-52 on page 4-66
br.ret.*bwh.ph.dh*e			21	4			

4.5.1.5 Indirect Call



Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.*bwh.ph.dh*e	$b_1 = b_2$	1	See Table 4-49 on page 4-65	See Table 4-51 on page 4-66	See Table 4-52 on page 4-66

4.5.2 Branch Predict and Nop

The branch predict and `nop` instructions are encoded in major opcodes 2 (Indirect Predict/Nop) and 7 (IP-relative Predict). The indirect predict and `nop` instructions in major opcode 2 use a 6-bit opcode extension field in bits 32:27 (x_6). Table 4-53 summarizes these assignments.

Table 4-53. Indirect Predict/Nop Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 30:27	Bits 32:31			
		0	1	2	3
2	0	nop.b B9	brp B7		
	1		brp.ret B7		
	2				
	3				
	4				
	5				
	6				
	7				
	8				
	9				
	A				
	B				
	C				
	D				
	E				
	F				

The branch predict instructions all have a 1-bit opcode extension field in bit 35 (ih) which encodes a branch importance hint. The mov to BR instruction (page 4-31) also has this hint in bit 23. Table 4-54 shows these assignments.

Table 4-54. Branch Importance Hint Completer

ih Bit 23 or Bit 35	*ih*
0	*none*
1	.imp

The IP-relative branch predict instructions have a 2-bit opcode extension field in bits 4:3 (wh) which encodes branch prediction "whether" hint information as shown in Table 4-55. Note that the combination of the .loop or .exit whether hint completer with the *none* importance hint completer is undefined.

Table 4-55. IP-relative Predict Whether Hint Completer

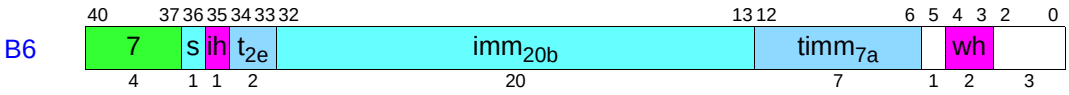
wh Bits 4:3	*ipwh*
0	.sptk
1	.loop
2	.dptk
3	.exit

The indirect branch predict instructions have a 2-bit opcode extension field in bits 4:3 (wh) which encodes branch prediction "whether" hint information as shown in Table 4-56.

Table 4-56. Indirect Predict Whether Hint Completer

wh Bits 4:3	*indwh*
0	.sptk
1	
2	.dptk
3	

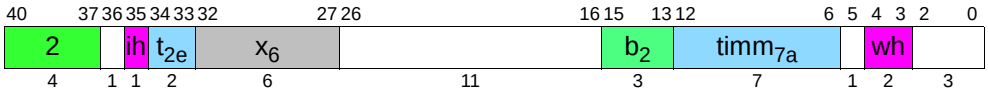
4.5.2.1 IP-Relative Predict



Instruction	Operands	Opcode	Extension	
			ih	wh
brp.*ipwh.ih*	$target_{25}$, tag_{13}	7	See Table 4-54 on page 4-69	See Table 4-55 on page 4-69

4.5.2.2 Indirect Predict

B7



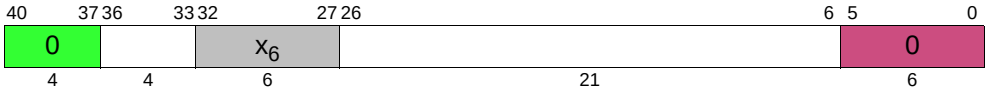
Instruction	Operands	Opcode	Extension		
			x_6	ih	wh
brp.*indwh.ih*	b_2, tag_{13}	2	10	See Table 4-54 on page 4-69	See Table 4-56 on page 4-69
brp.ret.*indwh.ih*			11		

4.5.3 Miscellaneous B-Unit Instructions

The miscellaneous branch-unit instructions include a number of instructions encoded within major opcode 0 using a 6-bit opcode extension field in bits 32:27 (x_6) as described in Table 4-46 on page 4-64.

4.5.3.1 Miscellaneous (B-Unit)

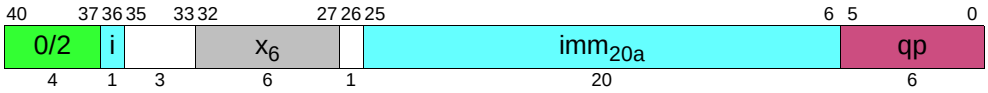
B8



Instruction	Opcode	Extension
		x_6
cover [I]	0	02
clrrrb [I]		04
clrrrb.pr [I]		05
rfi [e I p]		08
bsw.0 [I p]		0C
bsw.1 [I p]		0D
epc		10

4.5.3.2 Break/Nop (B-Unit)

B9



Instruction	Operands	Opcode	Extension
			x_6
break.b [e]	imm_{21}	0	00
nop.b		2	

4.6 F-Unit Instruction Encodings

The floating-point instructions are encoded in major opcodes 8 – E for floating-point and fixed-point arithmetic, opcode 4 for floating-point compare, opcode 5 for floating-point class, and opcodes 0 and 1 for miscellaneous floating-point instructions.

The miscellaneous and reciprocal approximation floating-point instructions are encoded within major opcodes 0 and 1 using a 1-bit opcode extension field (x) in bit 33 and either a second 1-bit extension field in bit 36 (q) or a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-57 shows the 1-bit x assignments, Table 4-60 shows the additional 1-bit q assignments for the reciprocal approximation instructions; Table 4-58 and Table 4-59 summarize the 6-bit x_6 assignments.

Most floating-point instructions have a 2-bit opcode extension field in bits 35:34 (sf) which encodes the FPSR status field to be used. Table 4-61 summarizes these assignments.

Table 4-57. Miscellaneous Floating-point 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	
0	0	6-bit Ext (Table 4-58)
	1	Reciprocal Approximation (Table 4-60)
1	0	6-bit Ext (Table 4-59)
	1	Reciprocal Approximation (Table 4-60)

Table 4-58. Opcode 0 Miscellaneous Floating-point 6-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	x_6 Bits 30:27	x_6 Bits 32:31 = 0	1	2	3
0	0	0	break.f F15	fmerge.s F9		
		1	nop.f F15	fmerge.ns F9		
		2		fmerge.se F9		
		3				
		4	fsetc F12	fmin F8		fswap F9
		5	fclrf F13	fmax F8		fswap.nl F9
		6		famin F8		fswap.nr F9
		7		famax F8		
		8	fchkf F14	fcvt.fx F10	fpack F9	
		9		fcvt.fxu F10		fmix.lr F9
		A		fcvt.fx.trunc F10		fmix.r F9
		B		fcvt.fxu.trunc F10		fmix.l F9
		C		fcvt.xf F11	fand F9	fsxt.r F9
		D			fandcm F9	fsxt.l F9
		E			for F9	
		F			fxor F9	

Table 4-59. Opcode 1 Miscellaneous Floating-point 6-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	x_6 Bits 30:27	Bits 32:31 = 0	Bits 32:31 = 1	Bits 32:31 = 2	Bits 32:31 = 3
1	0	0		fpmerge.s F9		fpcmp.eq F8
		1		fpmerge.ns F9		fpcmp.lt F8
		2		fpmerge.se F9		fpcmp.le F8
		3				fpcmp.unord F8
		4		fpmin F8		fpcmp.neq F8
		5		fpmax F8		fpcmp.nlt F8
		6		fpamin F8		fpcmp.nle F8
		7		fpamax F8		fpcmp.ord F8
		8		fpcvt.fx F10		
		9		fpcvt.fxu F10		
		A		fpcvt.fx.trunc F10		
		B		fpcvt.fxu.trunc F10		
		C				
		D				
		E				
		F				

Table 4-60. Reciprocal Approximation 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	q Bit 36	
0	1	0	frcpa F6
		1	frsqrta F7
1		0	fprcpa F6
		1	fprsqrta F7

Table 4-61. Floating-point Status Field Completer

sf Bits 35:34	*sf*
0	.s0
1	.s1
2	.s2
3	.s3

4.6.1 Arithmetic

The floating-point arithmetic instructions are encoded within major opcodes 8 – D using a 1-bit opcode extension field (x) in bit 36 and a 2-bit opcode extension field (sf) in bits 35:34. The opcode and x assignments are shown in Table 4-62.

Table 4-62. Floating-point Arithmetic 1-bit Opcode Extensions

x Bit 36	Opcode Bits 40:37					
	8	9	A	B	C	D
0	fma F1	fma.d F1	fms F1	fms.d F1	fnma F1	fnma.d F1
1	fma.s F1	fpma F1	fms.s F1	fpms F1	fnma.s F1	fpnma F1

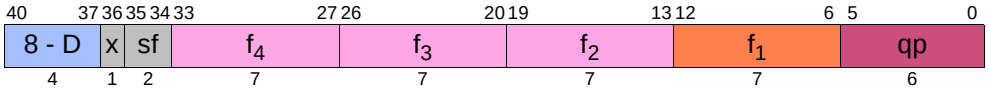
The fixed-point arithmetic and parallel floating-point select instructions are encoded within major opcode E using a 1-bit opcode extension field (x) in bit 36. The fixed-point arithmetic instructions also have a 2-bit opcode extension field (x_2) in bits 35:34. These assignments are shown in Table 4-63.

Table 4-63. Fixed-point Multiply Add and Select Opcode Extensions

Opcode Bits 40:37	x Bit 36	x_2 Bits 35:34			
		0	1	2	3
E	0	fselect F3			
	1	xma.l F2		xma.hu F2	xma.h F2

4.6.1.1 Floating-point Multiply Add

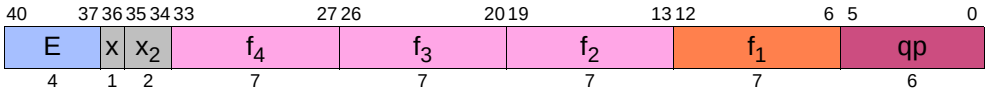
F1



Instruction	Operands	Opcode	Extension	
			x	sf
fma.*sf* fma.s.*sf*	$f_1 = f_3, f_4, f_2$	8	0 1	See Table 4-61 on page 4-72
fma.d.*sf* fpma.*sf*		9	0 1	
fms.*sf* fms.s.*sf*		A	0 1	
fms.d.*sf* fpms.*sf*		B	0 1	
fnma.*sf* fnma.s.*sf*		C	0 1	
fnma.d.*sf* fpnma.*sf*		D	0 1	

4.6.1.2 Fixed-point Multiply Add

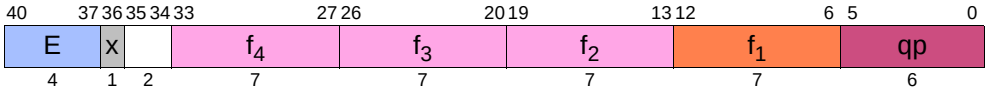
F2



Instruction	Operands	Opcode	Extension	
			x	x_2
xma.l	$f_1 = f_3, f_4, f_2$	E	1	0
xma.h				3
xma.hu				2

4.6.2 Parallel Floating-point Select

F3



Instruction	Operands	Opcode	Extension
			x
fselect	$f_1 = f_3, f_4, f_2$	E	0

4.6.3 Compare and Classify

The predicate setting floating-point compare instructions are encoded within major opcode 4 using three 1-bit opcode extension fields in bits 33 (r_a), 36 (r_b), and 12 (t_a), and a 2-bit opcode extension field (sf) in bits 35:34. The opcode, r_a, r_b, and t_a assignments are shown in Table 4-64. The sf assignments are shown in Table 4-61 on page 4-72.

The parallel floating-point compare instructions are described on page 4-77.

The floating-point class instructions are encoded within major opcode 5 using a 1-bit opcode extension field in bit 12 (t_a) as shown in Table 4-65.

Table 4-64. Floating-point Compare Opcode Extensions

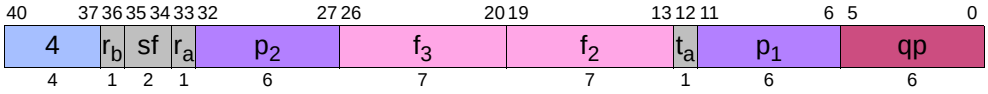
Opcode Bits 40:37	r_a Bit 33	r_b Bit 36	t_a Bit 12	
			0	1
4	0	0	fcmp.eq F4	fcmp.eq.unc F4
		1	fcmp.lt F4	fcmp.lt.unc F4
	1	0	fcmp.le F4	fcmp.le.unc F4
		1	fcmp.unord F4	fcmp.unord.unc F4

Table 4-65. Floating-point Class 1-bit Opcode Extensions

Opcode Bits 40:37	t_a Bit 12	
5	0	fclass.m F5
	1	fclass.m.unc F5

4.6.3.1 Floating-point Compare

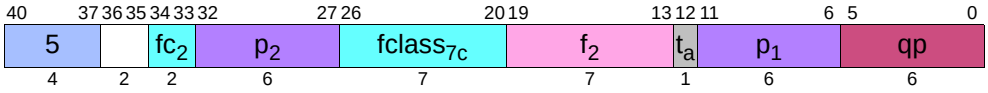
F4



Instruction	Operands	Opcode	Extension			
			r_a	r_b	t_a	sf
fcmp.eq.*sf*	$p_1, p_2 = f_2, f_3$	4	0	0	0	See Table 4-61 on page 4-72
fcmp.lt.*sf*				1		
fcmp.le.*sf*			1	0		
fcmp.unord.*sf*				1		
fcmp.eq.unc.*sf*			0	0	1	
fcmp.lt.unc.*sf*				1		
fcmp.le.unc.*sf*			1	0		
fcmp.unord.unc.*sf*				1		

4.6.3.2 Floating-point Class

F5



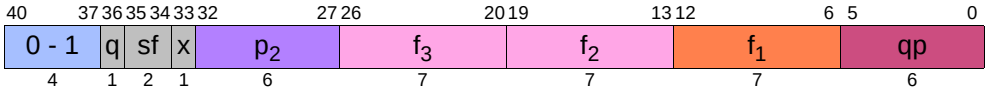
Instruction	Operands	Opcode	Extension
			t_a
fclass.m	$p_1, p_2 = f_2, fclass_9$	5	0
fclass.m.unc			1

4.6.4 Approximation

4.6.4.1 Floating-point Reciprocal Approximation

There are two Reciprocal Approximation instructions. The first, in major op 0, encodes the full register variant. The second, in major op 1, encodes the parallel variant.

F6

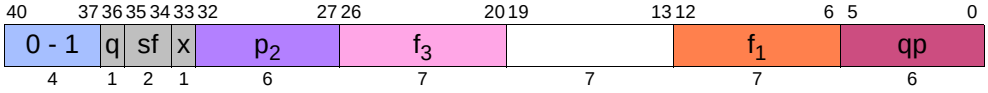


Instruction	Operands	Opcode	Extension		
			x	q	sf
frcpa.*sf* fprcpa.*sf*	$f_1, p_2 = f_2, f_3$	0 1	1	0	See Table 4-61 on page 4-72

4.6.4.2 Floating-point Reciprocal Square Root Approximation

There are two Reciprocal Square Root Approximation instructions. The first, in major op 0, encodes the full register variant. The second, in major op 1, encodes the parallel variant.

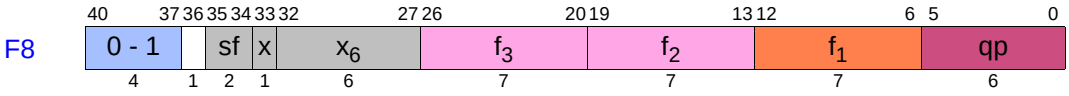
F7



Instruction	Operands	Opcode	Extension		
			x	q	sf
frsqrta.*sf* fprsqrta.*sf*	$f_1, p_2 = f_3$	0 1	1	1	See Table 4-61 on page 4-72

4.6.5 Minimum/Maximum and Parallel Compare

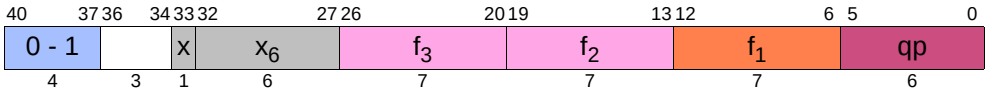
There are two groups of Minimum/Maximum instructions. The first group, in major op 0, encodes the full register variants. The second group, in major op 1, encodes the parallel variants. The parallel compare instructions are all encoded in major op 1.



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fmin.*sf*	$f_1 = f_2, f_3$	0	0	14	See Table 4-61 on page 4-72
fmax.*sf*				15	
famin.*sf*				16	
famax.*sf*				17	
fpmin.*sf*		1		14	
fpmax.*sf*				15	
fpamin.*sf*				16	
fpamax.*sf*				17	
fpcmp.eq.*sf*				30	
fpcmp.lt.*sf*				31	
fpcmp.le.*sf*				32	
fpcmp.unord.*sf*				33	
fpcmp.neq.*sf*				34	
fpcmp.nlt.*sf*				35	
fpcmp.nle.*sf*				36	
fpcmp.ord.*sf*				37	

4.6.6 Merge and Logical

F9

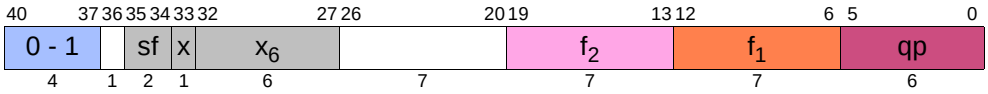


Instruction	Operands	Opcode	Extension	
			x	x6
fmerge.s	$f_1 = f_2, f_3$	0	0	10
fmerge.ns				11
fmerge.se				12
fmix.lr				39
fmix.r				3A
fmix.l				3B
fsxt.r				3C
fsxt.l				3D
fpack				28
fswap				34
fswap.nl				35
fswap.nr				36
fand				2C
fandcm				2D
for				2E
fxor				2F
fpmerge.s		1		10
fpmerge.ns				11
fpmerge.se				12

4.6.7 Conversion

4.6.7.1 Convert Floating-point to Fixed-point

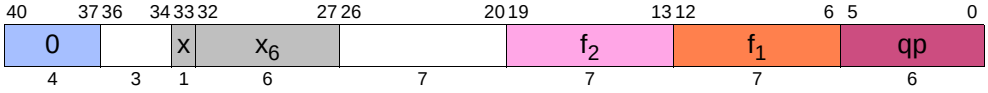
F10



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fcvt.fx.*sf*	$f_1 = f_2$	0	0	18	See Table 4-61 on page 4-72
fcvt.fxu.*sf*				19	
fcvt.fx.trunc.*sf*				1A	
fcvt.fxu.trunc.*sf*				1B	
fpcvt.fx.*sf*		1		18	
fpcvt.fxu.*sf*				19	
fpcvt.fx.trunc.*sf*				1A	
fpcvt.fxu.trunc.*sf*				1B	

4.6.7.2 Convert Fixed-point to Floating-point

F11

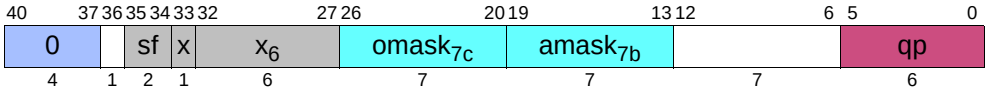


Instruction	Operands	Opcode	Extension	
			x	x_6
fcvt.xf	$f_1 = f_2$	0	0	1C

4.6.8 Status Field Manipulation

4.6.8.1 Floating-point Set Controls

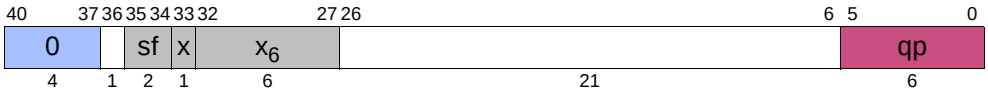
F12



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fsetc.*sf*	$amask_7$, $omask_7$	0	0	04	See Table 4-61 on page 4-72

4.6.8.2 Floating-point Clear Flags

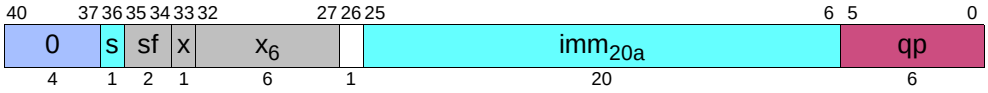
F13



Instruction	Opcode	Extension		
		x	x_6	sf
fclrf.*sf*	0	0	05	See Table 4-61 on page 4-72

4.6.8.3 Floating-point Check Flags

F14

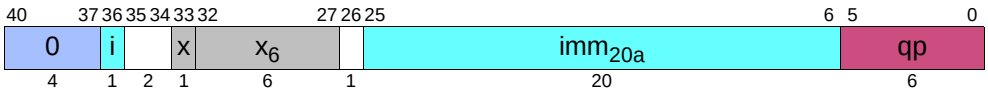


Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fchkf.*sf*	$target_{25}$	0	0	08	See Table 4-61 on page 4-72

4.6.9 Miscellaneous F-Unit Instructions

4.6.9.1 Break/Nop (F-Unit)

F15



Instruction	Operands	Opcode	Extension	
			x	x_6
break.f nop.f	imm_{21}	0	0	00 01

4.7 X-Unit Instruction Encodings

The X-unit instructions occupy two instruction slots, L+X. The major opcode, opcode extensions and hints, qp, and small immediate fields occupy the X instruction slot. For `movl`, `break.x`, and `nop.x`, the imm_{41} field occupies the L instruction slot. For `brl`, the imm_{39} field and a 2-bit Ignored field occupy the L instruction slot.

4.7.1 Miscellaneous X-Unit Instructions

The miscellaneous X-unit instructions are encoded in major opcode 0 using a 3-bit opcode extension field (x_3) in bits 35:33 and a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-66 shows the 3-bit assignments and Table 4-67 summarizes the 6-bit assignments. These instructions are executed by an I-unit.

Table 4-66. Misc X-Unit 3-bit Opcode Extensions

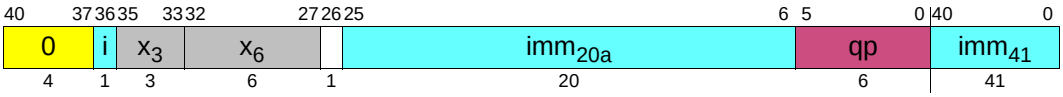
Opcode Bits 40:37	x_3 Bits 35:33	
0	0	6-bit Ext (Table 4-67)
	1	
	2	
	3	
	4	
	5	
	6	
	7	

Table 4-67. Misc X-Unit 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 30:27	x_6 Bits 32:31 = 0	1	2	3
0	0	0	break.x X1			
		1	nop.x X1			
		2				
		3				
		4				
		5				
		6				
		7				
		8				
		9				
		A				
		B				
		C				
		D				
		E				
		F				

4.7.1.1 Break/Nop (X-Unit)

X1



Instruction	Operands	Opcode	Extension	
			x_3	x_6
break.x nop.x	*imm*$_{62}$	0	0	00 01

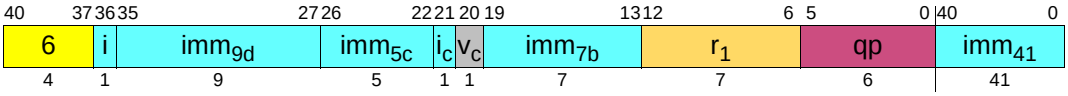
4.7.2 Move Long Immediate$_{64}$

The move long immediate instruction is encoded within major opcode 6 using a 1-bit reserved opcode extension in bit 20 (v_c) as shown in Table 4-68. This instruction is executed by an I-unit.

Table 4-68. Move Long 1-bit Opcode Extensions

Opcode Bits 40:37	v_c Bit 20	
6	0	movl X2
	1	

X2



Instruction	Operands	Opcode	Extension
			v_c
movl [i]	$r_1 = imm_{64}$	6	0

4.7.3 Long Branches

Long branches are executed by a B-unit. Opcode C is used for long branch and opcode D for long call.

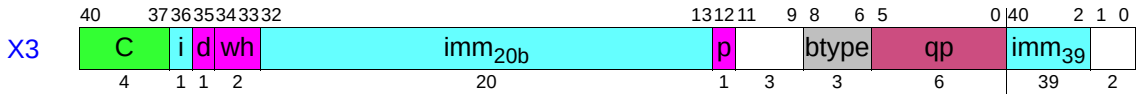
The long branch instructions encoded within major opcode C use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-69.

Table 4-69. Long Branch Types

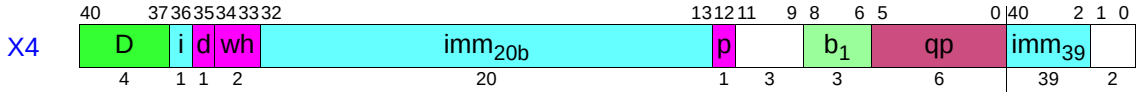
Opcode Bits 40:37	btype Bits 8:6	
C	0	
	1	
	2	
	3	
	4	
	5	
	6	
	7	

The long branch instructions have the same opcode hint fields in bit 12 (p), bits 34:33 (wh), and bit 35 (d) as normal IP-relative branches. These are shown in Table 4-49 on page 4-65, Table 4-50 on page 4-65, and Table 4-52 on page 4-66.

4.7.3.1 Long Branch



4.7.3.2 Long Call



4.8 Immediate Formation

Table 4-70 shows, for each instruction format that has one or more immediates, how those immediates are formed. In each equation, the symbol to the left of the equals is the assembly language name for the immediate. The symbols to the right are the field names in the instruction encoding.

Table 4-70. Immediate Formation

Instruction Format	Immediate Formation
A2	$count_2 = ct_{2d} + 1$
A3 A8 I27 M30	$imm_8 = sign_ext(s << 7 \mid imm_{7b}, 8)$
A4	$imm_{14} = sign_ext(s << 13 \mid imm_{6d} << 7 \mid imm_{7b}, 14)$
A5	$imm_{22} = sign_ext(s << 21 \mid imm_{5c} << 16 \mid imm_{9d} << 7 \mid imm_{7b}, 22)$
A10	$count_2 = (ct_{2d} > 2)$? reservedQP[a] : $ct_{2d} + 1$
I1	$count_2 = (ct_{2d} == 0)$? 0 : $(ct_{2d} == 1)$? 7 : $(ct_{2d} == 2)$? 15 : 16
I3	$mbtype_4 = (mbt_{4c} == 0)$? @brcst : $(mbt_{4c} == 8)$? @mix : $(mbt_{4c} == 9)$? @shuf : $(mbt_{4c} == 0xA)$? @alt : $(mbt_{4c} == 0xB)$? @rev : reservedQP[a]
I4	$mhtype_8 = mht_{8c}$
I6	$count_5 = count_{5b}$
I8	$count_5 = 31 - ccount_{5c}$
I10	$count_6 = count_{6d}$
I11	$len_6 = len_{6d} + 1$ $pos_6 = pos_{6b}$
I12	$len_6 = len_{6d} + 1$ $pos_6 = 63 - cpos_{6c}$
I13	$len_6 = len_{6d} + 1$ $pos_6 = 63 - cpos_{6c}$ $imm_8 = sign_ext(s << 7 \mid imm_{7b}, 8)$
I14	$len_6 = len_{6d} + 1$ $pos_6 = 63 - cpos_{6b}$ $imm_1 = sign_ext(s, 1)$
I15	$len_4 = len_{4d} + 1$ $pos_6 = 63 - cpos_{6d}$
I16	$pos_6 = pos_{6b}$
I19 M37	$imm_{21} = i << 20 \mid imm_{20a}$
I22	$tag_{13} = IP + (sign_ext(timm_{9c}, 9) << 4)$
I23	$mask_{17} = sign_ext(s << 16 \mid mask_{8c} << 8 \mid mask_{7a} << 1, 17)$
I24	$imm_{44} = sign_ext(s << 43 \mid imm_{27a} << 16, 44)$
M3 M8 M15	$imm_9 = sign_ext(s << 8 \mid i << 7 \mid imm_{7b}, 9)$
M5 M10	$imm_9 = sign_ext(s << 8 \mid i << 7 \mid imm_{7a}, 9)$
M17	$inc_3 = sign_ext(((s)$? –1 : 1) * $((i_{2b} == 3)$? 1 : $1 << (4 - i_{2b})), 6)$
I20 M20 M21	$target_{25} = IP + (sign_ext(s << 20 \mid imm_{13c} << 7 \mid imm_{7a}, 21) << 4)$

Table 4-70. Immediate Formation (Continued)

Instruction Format	Immediate Formation
M22 M23	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20b}, 21) << 4)
M34	il = sol o = sof – sol r = sor << 3
M39 M40	$imm_2 = i_{2b}$
M44	imm_{24} = i << 23 | i_{2d} << 21 | imm_{21a}
B1 B2 B3	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20b}, 21) << 4)
B7	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20b}, 21) << 4) tag_{13} = IP + (sign_ext(t_{2e} << 7 | $timm_{7a}$, 9) << 4)
B9	tag_{13} = IP + (sign_ext(t_{2e} << 7 | $timm_{7a}$, 9) << 4)
B9	imm_{21} = i << 20 | imm_{20a}
F5	$fclass_9 = fclass_{7c}$ << 2 | fc_2
F12	$amask_7 = amask_{7b}$ $omask_7 = omask_{7c}$
F14	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20a}, 21) << 4)
F15	imm_{21} = i << 20 | imm_{20a}
X1	$imm_{62} = imm_{41}$ << 21 | i << 20 | imm_{20a}
X2	imm_{64} = i << 63 | imm_{41} << 22 | i_c << 21 | imm_{5c} << 16 | imm_{9d} << 7 | imm_{7b}
X3 X4	$target_{64}$ = IP + ((i << 59 | imm_{39} << 20 | imm_{20b}) << 4)

a. This encoding causes an Illegal Operation fault if the value of the qualifying predicate is 1.

Part II: IA-32 Instruction Set Descriptions

Base IA-32 Instruction Reference 5

This section lists all IA-32 instructions and their behavior in the IA-64 System Environment and IA-32 System Environments on an IA-64 processor. Unless noted otherwise all IA-32 and MMX and Streaming SIMD Extension instructions operate as defined in the *Intel Architecture Software Developer's Manual*.

This volume describes the complete Intel IA-32 Architecture instruction set, including the integer, floating-point, MMX technology and Streaming SIMD Extension technology, and system instructions. The instruction descriptions are arranged in alphabetical order. For each instruction, the forms are given for each operand combination, including the opcode, operands required, and a description. Also given for each instruction are a description of the instruction and its operands, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of the exceptions that can be generated.

For all IA-32 the following relationships hold:

- **Writes** – Writes of any IA-32 general purpose, floating-point or Streaming SIMD Extension, MMX technology registers by IA-32 instructions are reflected in the IA-64 registers defined to hold that IA-32 state when IA-32 instruction set completes execution.
- **Reads** – Reads of any IA-32 general purpose, floating-point or Streaming SIMD Extension, MMX technology registers by IA-32 instructions see the state of the IA-64 registers defined to hold the IA-32 state after entering the IA-32 instruction set.
- **State mappings** – IA-32 numeric instructions are controlled by and reflect their status in FCW, FSW, FTW, FCS, FIP, FOP, FDS and FEA. On exit from the IA-32 instruction set, IA-64 numeric status and control resources defined to hold IA-32 state reflect the results of all IA-32 prior numeric instructions in FCR, FSR, FIR and FDR. IA-64 numeric status and control resources defined to hold IA-32 state are honored by IA-32 numeric instructions when entering the IA-32 instruction set.

5.1 IA-64 Additional Faults

The following fault behavior is defined for all IA-32 instructions in the IA-64 System Environment:

- **IA-32 Faults** – All IA-32 faults are performed as defined in the *Intel Architecture Software Developer's Manual*, unless otherwise noted. IA-32 faults are delivered in the IA-64 Environment on the IA-32_Exception interruption vector.
- **IA-32 GPFault** – Null segments are signified by the segment descriptor register's P-bit being set to zero. IA-32 memory references through DSD, ESD, FSD, and GSD with the P-bit set to zero result in an IA-32 GPFault.
- **IA-64 Low FP Reg Fault** – If PSR.dfl is 1, execution of any IA-32 MMX technology, Streaming SIMD Extension or floating-point instructions results in a Disabled FP Register fault (regardless of whether FR2-31 is referenced).

- **IA-64 High FP Reg Fault** – If PSR.dfh is 1, execution of the first target IA-32 instruction following an `br.ia` or `rfi` results in a Disabled FP Register fault (regardless of whether FR32-127 is referenced).
- **IA-64 Instruction Mem Faults** – The following additional IA-64 memory faults can be generated on each virtual page referenced when fetching IA-32 or MMX or Streaming SIMD Extension instructions for execution:
  - Alternative instruction TLB fault
  - VHPT instruction fault
  - Instruction TLB fault
  - Instruction Page Not Present fault
  - Instruction NaT Page Consumption Abort
  - Instruction Key Miss fault
  - Instruction Key Permission fault
  - Instruction Access Rights fault
  - Instruction Access Bit fault
- **IA-64 Data Mem Faults** – The following additional IA-64 memory faults can be generated on each virtual page touched when reading or writing memory operands from the IA-32 instruction set including MMX and Streaming SIMD Extension instructions:
  - Nested TLB fault
  - Alternative data TLB fault
  - VHPT data fault
  - Data TLB fault
  - Data Page Not Present fault
  - Data NaT Page Consumption Abort
  - Data Key Miss fault
  - Data Key Permission fault
  - Data Access Rights fault
  - Data Dirty bit fault
  - Data Access bit fault

5.2 Interpreting the IA-32 Instruction Reference Pages

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

5.2.1 IA-32 Instruction Format

The following is an example of the format used for each Intel Architecture instruction description in this chapter.

CMC—Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement carry flag

5.2.1.1 Opcode Column

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

- **/digit** – A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- **/r** – Indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.
- **cb, cw, cd, cp** – A 1-byte (cb), 2-byte (cw), 4-byte (cd), or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id** – A 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.
- **+rb, +rw, +rd** – A register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The register codes are given in Table 5-1.
- **+i** – A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

Table 5-1. Register Encodings Associated with the +rb, +rw, and +rd Nomenclature

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

5.2.1.2 Instruction Column

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

- **rel8** – A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
- **rel16 and rel32** – A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.
- **ptr16:16 and ptr16:32** – A far pointer, typically in a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.
- **r8** – One of the byte general-purpose registers AL, CL, DL, BL, AH, CH, DH, or BH.
- **r16** – One of the word general-purpose registers AX, CX, DX, BX, SP, BP, SI, or DI.
- **r32** – One of the doubleword general-purpose registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.
- **imm8** – An immediate byte value. The imm8 symbol is a signed number between –128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- **imm16** – An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.
- **imm32** – An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and –2,147,483,648 inclusive.
- **r/m8** – A byte operand that is either the contents of a byte general-purpose register (AL, BL, CL, DL, AH, BH, CH, and DH), or a byte from memory.
- **r/m16** – A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, BX, CX, DX, SP, BP, SI, and DI. The contents of memory are found at the address provided by the effective address computation.
- **r/m32** – A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI. The contents of memory are found at the address provided by the effective address computation.
- **m** – A 16- or 32-bit operand in memory.
- **m8** – A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions and the XLAT instruction.
- **m16** – A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.

- **m32** – A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m64** – A memory quadword operand in memory. This nomenclature is used only with the CMPXCHG8B instruction.
- **m16:16, m16:32** – A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.
- **m16&32, m16&16, m32&32** – A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers.
- **moffs8, moffs16, moffs32** – A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.
- **Sreg** – A segment register. The segment register bit assignments are ES=0, CS=1, SS=2, DS=3, FS=4, and GS=5.
- **m32real, m64real, m80real** – A single-, double-, and extended-real (respectively) floating-point operand in memory.
- **m16int, m32int, m64int** – A word-, short-, and long-integer (respectively) floating-point operand in memory.
- **ST or ST(0)** – The top element of the FPU register stack.
- **ST(i)** – The i^{th} element from the top of the FPU register stack. (*i* = 0 through 7).
- **mm** – An MMX technology register. The 64-bit MMX technology registers are: MM0 through MM7.
- **mm/m32** – The low order 32 bits of an MMX technology register or a 32-bit memory operand. The 64-bit MMX technology registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **mm/m64** – An MMX technology register or a 64-bit memory operand. The 64-bit MMX technology registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.

5.2.1.3 Description Column

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

5.2.1.4 Description

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

5.2.2 Operation

The “Operation” section contains an algorithmic description (written in pseudo-code) of the instruction. The pseudo-code uses a notation similar to the Algol or Pascal language. The algorithms are composed of the following elements:

- Comments are enclosed within the symbol pairs “(*” and “*)”.
- Compound statements are enclosed in keywords, such as IF, THEN, ELSE, and FI for an if statement, DO and OD for a do statement, or CASE... OF and ESAC for a case statement.
- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to SI’s default segment (DS) or overridden segment.
- Parentheses around the “E” in a general-purpose register name, such as (E)SI, indicates that an offset is read from the SI register if the current address-size attribute is 16 or is read from the ESI register if the address-size attribute is 32.
- Brackets are also used for memory operands, where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the contents of the source operand is a segment-relative offset.
- A ← B; indicates that the value of B is assigned to A.
- The symbols =, ≠, ≥, and ≤ are relational operators used to compare two values, meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.
- The expression “<< COUNT” and “>> COUNT” indicates that the destination operand should be shifted left or right, respectively, by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize and AddressSize** – The OperandSize identifier represents the operand-size attribute of the instruction, which is either 16 or 32 bits. The AddressSize identifier represents the address-size attribute, which is either 16 or 32 bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the CMPS instruction used.

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

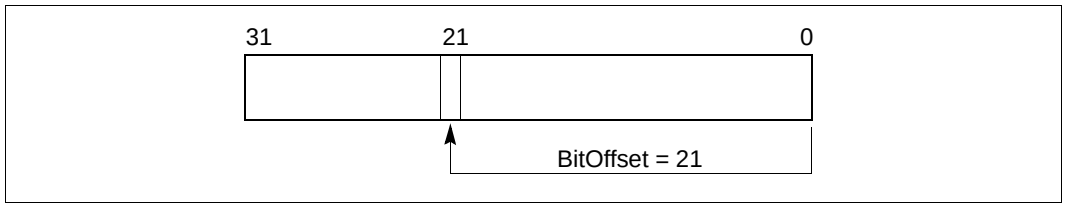
  See “Operand-Size and Address-Size Attributes” in Chapter 3 of the *Intel Architecture Software Developer’s Manual, Volume 1*, for general guidelines on how these attributes are determined.
- **StackAddrSize** – Represents the stack address-size attribute associated with the instruction, which has a value of 16 or 32 bits (see “Address-Size Attribute for Stack” in Chapter 4 of the *Intel Architecture Software Developer’s Manual, Volume 1*).
- **SRC** – Represents the source operand.
- **DEST** – Represents the destination operand.

The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)** – Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of –10 converts the byte from F6H to a doubleword value of 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.
- **SignExtend(value)** – Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value –10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **SaturateSignedWordToSignedByte** – Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateSignedDwordToSignedWord** – Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateSignedWordToUnsignedByte** – Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToSignedByte** – Represents the result of an operation as a signed 8-bit value. If the result is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateToSignedWord** – Represents the result of an operation as a signed 16-bit value. If the result is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateToUnsignedByte** – Represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToUnsignedWord** – Represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).
- **LowOrderWord(DEST * SRC)** – Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.
- **HighOrderWord(DEST * SRC)** – Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.
- **Push(value)** – Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction.
- **Pop()** – Removes the value from the top of the stack and returns it. The statement EAX ← Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word or a doubleword depending on the operand-size attribute.
- **PopRegisterStack** – Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.
- **Switch-Tasks** – Performs a task switch.
- **Bit(BitBase, BitOffset)** – Returns the value of a bit within a bit string, which is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers

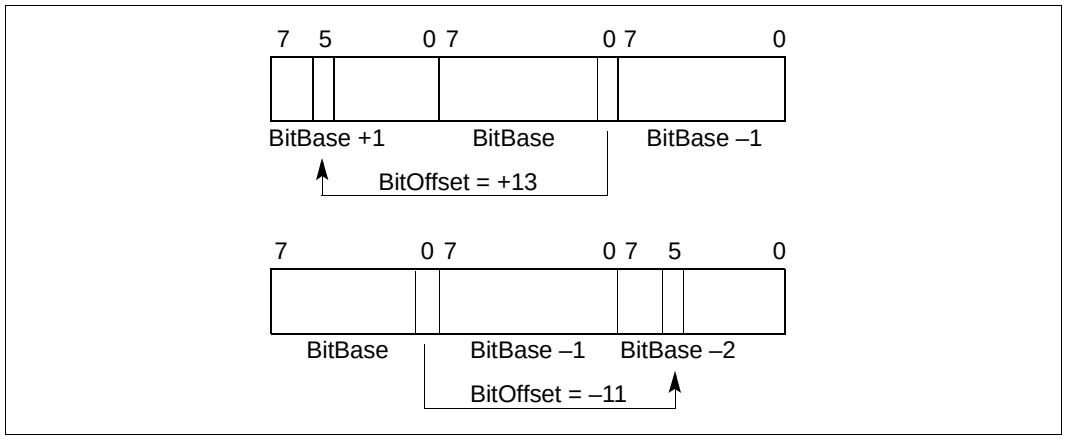
and within memory bytes. If the base operand is a register, the offset can be in the range 0..31. This offset addresses a bit within the indicated register. An example, the function Bit[EAX, 21] is illustrated in Figure 5-1.

Figure 5-1. Bit Offset for BIT[EAX,21]



If BitBase is a memory address, BitOffset can range from –2 GBits to 2 GBits. The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)), where DIV is signed division with rounding towards negative infinity, and MOD returns a positive number. This operation is illustrated in Figure 5-2.

Figure 5-2. Memory Bit Indexing



5.2.3 Flags Affected

The “Flags Affected” section lists the flags in the EFLAGS register that are affected by the instruction. When a flag is cleared, it is equal to 0; when it is set, it is equal to 1. The arithmetic and logical instructions usually assign values to the status flags in a uniform manner (see Appendix A, *EFLAGS Cross-Reference*, in the *Intel Architecture Software Developer’s Manual, Volume 1*). Non-conventional assignments are described in the “Operation” section. The values of flags listed as **undefined** may be changed by the instruction in an indeterminate manner. Flags that are not listed are unchanged by the instruction.

5.2.4 FPU Flags Affected

The floating-point instructions have an “FPU Flags Affected” section that describes how each instruction can affect the four condition code flags of the FPU status word.

5.2.5 Protected Mode Exceptions

The “Protected Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in protected mode and the reasons for the exceptions. Each exception is given a mnemonic that consists of a pound sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 5-2 associates each two-letter mnemonic with the corresponding interrupt vector number and exception name. See Chapter 5, *Interrupt and Exception Handling*, in the *Intel Architecture Software Developer’s Manual, Volume 3*, for a detailed description of the exceptions.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

5.2.6 Real-address Mode Exceptions

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

Table 5-2. Exception Mnemonics, Names, and Vector Numbers

Vector No.	Mnemonic	Name	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
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.[a]
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.
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.
16	#MF	Floating-point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[b]
18	#MC	Machine Check	Model dependent.[c]

a. The UD2 instruction was introduced in the Pentium® Pro processor.
b. This exception was introduced in the Intel486™ processor.
c. This exception was introduced in the Pentium processor and enhanced in the Pentium Pro processor.

5.2.7 Virtual-8086 Mode Exceptions

The “Virtual-8086 Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode.

5.2.8 Floating-point Exceptions

The “Floating-point Exceptions” section lists additional exceptions that can occur when a floating-point instruction is executed in any mode. All of these exception conditions result in a floating-point error exception (#MF, vector number 16) being generated. Table 5-3 associates each one- or two-letter mnemonic with the corresponding exception name. See “Floating-Point Exception Conditions” in Chapter 7 of the *Intel Architecture Software Developer’s Manual, Volume 1*, for a detailed description of these exceptions.

Table 5-3. Floating-point Exception Mnemonics and Names

Vector No.	Mnemonic	Name	Source
16		Floating-point invalid operation:	
	#IS	- Stack overflow or underflow	- FPU stack overflow or underflow
	#IA	- Invalid arithmetic operation	- Invalid FPU arithmetic operation
16	#Z	Floating-point divide-by-zero	FPU divide-by-zero
16	#D	Floating-point denormalized operation	Attempting to operate on a denormal number
16	#O	Floating-point numeric overflow	FPU numeric overflow
16	#U	Floating-point numeric underflow	FPU numeric underflow
16	#P	Floating-point inexact result (precision)	Inexact result (precision)

5.3 IA-32 Base Instruction Reference

The remainder of this chapter provides detailed descriptions of each of the Intel Architecture instructions.

AAA—ASCII Adjust After Addition

Opcode	Instruction	Description
37	AAA	ASCII adjust AL after addition

Description

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

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

Operation

```
IF ((AL AND FH) > 9) OR (AF = 1)
  THEN
       AL ← (AL + 6);
       AH ← AH + 1;
       AF ← 1;
       CF ← 1;
  ELSE
       AF ← 0;
       CF ← 0;
FI;
AL ←  AL AND FH;
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

AAD—ASCII Adjust AX Before Division

Opcode	Instruction	Description
D5 0A	AAD	ASCII adjust AX before division

Description

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

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

Operation

```
tempAL ← AL;
tempAH ← AH;
AL ← (tempAL + (tempAH * imm8)) AND FFH;
AH ← 0
```

The immediate value (*imm8*) is taken from the second byte of the instruction, which under normal assembly is 0AH (10 decimal). However, this immediate value can be changed to produce a different result.

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

AAM—ASCII Adjust AX After Multiply

Opcode	Instruction	Description
D4 0A	AAM	ASCII adjust AX after multiply

Description

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

Operation

tempAL ← AL;
AH ← tempAL / *imm8*;
AL ← tempAL MOD *imm8*;

The immediate value (*imm8*) is taken from the second byte of the instruction, which under normal assembly is 0AH (10 decimal). However, this immediate value can be changed to produce a different result.

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

AAS—ASCII Adjust AL After Subtraction

Opcode	Instruction	Description
3F	AAS	ASCII adjust AL after subtraction

Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register is decremented by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top nibble set to 0.

Operation

```
IF ((AL AND FH) > 9) OR (AF = 1)
THEN
   AL ← AL – 6;
   AH ← AH – 1;
   AF ← 1;
   CF ← 1;
ELSE
   CF ← 0;
   AF ← 0;
FI;
AL ← AL AND FH;
```

Flags Affected

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

ADC—Add with Carry

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

Description

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

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

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

Operation

```
DEST ← DEST + SRC + CF;
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

ADC—Add with Carry (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

ADD—Add

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

Description

Adds the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

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

Operation

```
DEST ← DEST + SRC;
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

ADD—Add (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

AND—Logical AND

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

Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location.

Operation

```
DEST ← DEST AND SRC;
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

AND—Logical AND (continued)

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

ARPL—Adjust RPL Field of Segment Selector

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

Description

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

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

See the *Intel Architecture Software Developer's Manual, Volume 3* for more information about the use of this instruction.

Operation

```
IF DEST(RPL) < SRC(RPL)
THEN
   ZF ← 1;
   DEST(RPL) ← SRC(RPL);
ELSE
   ZF ← 0;
FI;
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

ARPL—Adjust RPL Field of Segment Selector (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

BOUND—Check Array Index Against Bounds

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

Description

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

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

Operation

```
IF (ArrayIndex < LowerBound OR ArrayIndex > (UppderBound + OperandSize/8]))
  (* Below lower bound or above upper bound *)
  THEN
       #BR;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BOUND—Check Array Index Against Bounds (continued)

Protected Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

BSF—Bit Scan Forward

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

Description

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

Operation

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

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BSF—Bit Scan Forward (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

BSR—Bit Scan Reverse

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

Description

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

Operation

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

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BSR—Bit Scan Reverse (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

BSWAP—Byte Swap

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

Description

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

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

Operation

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

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

Intel Architecture Compatibility Information

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

BT—Bit Test

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

Description

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

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

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

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

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

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

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

Operation

CF ← Bit(BitBase, BitOffset)

Flags Affected

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

BT—Bit Test (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

BTC—Bit Test and Complement

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

Description

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

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" on page 5-30 for more information on this addressing mechanism.

Operation

```
CF ← Bit(BitBase, BitOffset)
Bit(BitBase, BitOffset) ← NOT Bit(BitBase, BitOffset);
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BTC—Bit Test and Complement (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

BTR—Bit Test and Reset

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

Description

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

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" on page 5-30 for more information on this addressing mechanism.

Operation

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

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BTR—Bit Test and Reset (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

BTS—Bit Test and Set

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

Description

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

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" on page 5-30 for more information on this addressing mechanism.

Operation

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

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BTS—Bit Test and Set (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

CALL—Call Procedure

Opcode	Instruction	Description
E8 *cw*	CALL *rel16*	Call near, displacement relative to next instruction
E8 *cd*	CALL *rel32*	Call near, displacement relative to next instruction
FF /2	CALL *r/m16*	Call near, *r/m16* indirect
FF /2	CALL *r/m32*	Call near, *r/m32* indirect
9A *cd*	CALL *ptr16:16*	Call far, to full pointer given
9A *cp*	CALL *ptr16:32*	Call far, to full pointer given
FF /3	CALL *m16:16*	Call far, address at *r/m16*
FF /3	CALL *m16:32*	Call far, address at *r/m32*

Description

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

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

- Near call – A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far call – A call to a procedure located in a different segment than the current code segment, sometimes referred to as an intersegment call.
- Inter-privilege-level far call – A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure. **Results in an IA-32_Intercept(Gate) in IA-64 System Environment.**
- Task switch – A call to a procedure located in a different task. **Results in an IA-32_Intercept(Gate) in IA-64 System Environment.**

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See Chapter 6 in the *Intel Architecture Software Developer's Manual, Volume 3* for information on task switching with the CALL instruction.

When executing a near call, the processor pushes the value of the EIP register (which contains the address of the instruction following the CALL instruction) onto the procedure stack (for use later as a return-instruction pointer. The processor then jumps to the address specified with the target operand for the called procedure. The target operand specifies either an absolute address in the code segment (that is an offset from the base of the code segment) or a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register, which points to the instruction following the call). An absolute address is specified directly in a register or indirectly in a memory location (*r/m16* or *r/m32* target-operand form). (When accessing an absolute address indirectly using the stack pointer (ESP) as a base register, the base value used is the value of the ESP before the instruction executes.) A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value, which is added to the instruction pointer.

CALL—Call Procedure (continued)

When executing a near call, the operand-size attribute determines the size of the target operand (16 or 32 bits) for absolute addresses. Absolute addresses are loaded directly into the EIP register. When a relative offset is specified, it is added to the value of the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits. The CS register is not changed on near calls.

When executing a far call, the processor pushes the current value of both the CS and EIP registers onto the procedure stack for use as a return-instruction pointer. The processor then performs a far jump to the code segment and address specified with the target operand for the called procedure. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s.

Any far call from a 32-bit code segment to a 16-bit code segment should be made from the first 64 Kbytes of the 32-bit code segment, because the operand-size attribute of the instruction is set to 16, allowing only a 16-bit return address offset to be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values will be pushed on the stack.

When the processor is operating in protected mode, a far call can also be used to access a code segment at a different privilege level or to switch tasks. Here, the processor uses the segment selector part of the far address to access the segment descriptor for the segment being jumped to. Depending on the value of the type and access rights information in the segment selector, the CALL instructon can perform:

- A far call to the same privilege level (described in the previous paragraph).
- An far call to a different privilege level. **Results in an IA-32_Intercept(Gate) in IA-64 System Environment.**
- A task switch. **Results in an IA-32_Intercept(Gate) in IA-64 System Environment.**

When executing an inter-privilege-level far call, the code segment for the procedure being called is accessed through a call gate. The segment selector specified by the target operand identifies the call gate. In executing a call through a call gate where a change of privilege level occurs, the processor switches to the stack for the privilege level of the called procedure, pushes the current values of the CS and EIP registers and the SS and ESP values for the old stack onto the new stack, then performs a far jump to the new code segment. The new code segment is specified in the call gate descriptor; the new stack segment is specified in the TSS for the currently running task. The jump to the new code segment occurs after the stack switch. On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, a set of parameters from the calling procedures stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.)

Finally, the processor jumps to the address of the procedure being called within the new code segment. The procedure address is the offset specified by the target operand. Here again, the target operand can specify the far address of the call gate and procedure either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

CALL—Call Procedure (continued)

Executing a task switch with the CALL instruction, is similar to executing a call through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to and the address of the procedure being called in the task. The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The CALL instruction can also specify the segment selector of the TSS directly. See the *Intel Architecture Software Developer's Manual, Volume 3* the for detailed information on the mechanics of a task switch.

Operation

```
IF near call
  THEN IF near relative call
       IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
       THEN IF OperandSize = 32
            THEN
                 IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
                 Push(EIP);
                 EIP ← EIP + DEST; (* DEST is rel32 *)
            ELSE (* OperandSize = 16 *)
                 IF stack not large enough for a 2-byte return address THEN #SS(0); FI;
                 Push(IP);
                 EIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
       FI;
  FI;
  ELSE (* near absolute call *)
       IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
       IF OperandSize = 32
            THEN
                 IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
                 Push(EIP);
                 EIP ← DEST; (* DEST is r/m32 *)
            ELSE (* OperandSize = 16 *)
                 IF stack not large enough for a 2-byte return address THEN #SS(0); FI;
                 Push(IP);
                 EIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
       FI;
  FI:
  IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
FI;
IF far call AND (PE = 0 OR (PE = 1 AND VM = 1)) (* real address or virtual 8086 mode *)
  THEN
       IF OperandSize = 32
            THEN
                 IF stack not large enough for a 6-byte return address THEN #SS(0); FI;
                 IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
                 Push(CS); (* padded with 16 high-order bits *)
                 Push(EIP);
                 CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
                 EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                 IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
                 IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
                 Push(CS);
                 Push(IP);
```

CALL—Call Procedure (continued)

```
                    CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
                    EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16] *)
                    EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
            FI;
            IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
FI;

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

CONFORMING-CODE-SEGMENT:
   IF DPL > CPL THEN #GP(new code segment selector); FI;
   IF not present THEN #NP(selector); FI;
   IF OperandSize = 32
        THEN
            IF stack not large enough for a 6-byte return address THEN #SS(0); FI;
            IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
            Push(CS); (* padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST(NewCodeSegmentSelector);
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL
            EIP ← DEST(offset);
        ELSE (* OperandSize = 16 *)
            IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
            IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
            Push(CS);
            Push(IP);
            CS ← DEST(NewCodeSegmentSelector);
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL
            EIP ← DEST(offset) AND 0000FFFFH; (* clear upper 16 bits *)
   FI;
   IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
END;

NONCONFORMING-CODE-SEGMENT:
   IF (RPL > CPL) OR (DPL ≠ CPL) THEN #GP(new code segment selector); FI;
   IF stack not large enough for return address THEN #SS(0); FI;
   tempEIP ← DEST(offset)
```

CALL—Call Procedure (continued)

```
    IF OperandSize=16
        THEN
            tempEIP ← tempEIP AND 0000FFFFH; (* clear upper 16 bits *)
    FI;
    IF tempEIP outside code segment limit THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST(NewCodeSegmentSelector);
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            EIP ← tempEIP;
        ELSE (* OperandSize = 16 *)
            Push(CS);
            Push(IP);
            CS ← DEST(NewCodeSegmentSelector);
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            EIP ← tempEIP;
    FI;
    IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
END;

CALL-GATE:
    IF call gate DPL < CPL or RPL THEN #GP(call gate selector); FI;
    IF not present THEN #NP(call gate selector); FI;
    IF IA-64 System Environment THEN IA-32_Intercept(Gate,CALL);
    IF call gate code-segment selector is null THEN #GP(0); FI;
    IF call gate code-segment selector index is outside descriptor table limits
        THEN #GP(code segment selector); FI;
    Read code segment descriptor;
    IF code-segment segment descriptor does not indicate a code segment
    OR code-segment segment descriptor DPL > CPL
        THEN #GP(code segment selector); FI;
    IF code segment not present THEN #NP(new code segment selector); FI;
    IF code segment is non-conforming AND DPL < CPL
        THEN go to MORE-PRIVILEGE;
        ELSE go to SAME-PRIVILEGE;
    FI;
END;

MORE-PRIVILEGE:
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ← new code segment (DPL ∗ 8) + 4
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(current TSS selector); FI;
            newSS ← TSSstackAddress + 4;
            newESP ← stack address;
        ELSE (* TSS is 16-bit *)
            TSSstackAddress ← new code segment (DPL ∗ 4) + 2
            IF (TSSstackAddress + 4) > TSS limit
                THEN #TS(current TSS selector); FI;
```

CALL—Call Procedure (continued)

```
                newESP ← TSSstackAddress;
                newSS ← TSSstackAddress + 2;
        FI;
        IF stack segment selector is null THEN #TS(stack segment selector); FI;
        IF stack segment selector index is not within its descriptor table limits
            THEN #TS(SS selector); FI
        Read code segment descriptor;
        IF stack segment selector's RPL ≠ DPL of code segment
            OR stack segment DPL ≠ DPL of code segment
            OR stack segment is not a writable data segment
                THEN #TS(SS selector); FI
        IF stack segment not present THEN #SS(SS selector); FI;
        IF CallGateSize = 32
            THEN
                IF stack does not have room for parameters plus 16 bytes
                    THEN #SS(SS selector); FI;
                IF CallGate(InstructionPointer) not within code segment limit THEN #GP(0); FI;
                SS ← newSS;
                (* segment descriptor information also loaded *)
                ESP ← newESP;
                CS:EIP ← CallGate(CS:InstructionPointer);
                (* segment descriptor information also loaded *)
                Push(oldSS:oldESP); (* from calling procedure *)
                temp ← parameter count from call gate, masked to 5 bits;
                Push(parameters from calling procedure's stack, temp)
                Push(oldCS:oldEIP); (* return address to calling procedure *)
            ELSE (* CallGateSize = 16 *)
                IF stack does not have room for parameters plus 8 bytes
                    THEN #SS(SS selector); FI;
                IF (CallGate(InstructionPointer) AND FFFFH) not within code segment limit
                    THEN #GP(0); FI;
                SS ← newSS;
                (* segment descriptor information also loaded *)
                ESP ← newESP;
                CS:IP ← CallGate(CS:InstructionPointer);
                (* segment descriptor information also loaded *)
                Push(oldSS:oldESP); (* from calling procedure *)
                temp ← parameter count from call gate, masked to 5 bits;
                Push(parameters from calling procedure's stack, temp)
                Push(oldCS:oldEIP); (* return address to calling procedure *)
        FI;
        CPL ← CodeSegment(DPL)
        CS(RPL) ← CPL
    END;

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

CALL—Call Procedure (continued)

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

TASK-GATE:
    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector);
    FI;
    IF task gate not present
        THEN #NP(task gate selector);
    FI;
    IF IA-64 System Environment THEN IA-32_Intercept(Gate,CALL);
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
        OR index not within GDT limits
            THEN #GP(TSS selector);
    FI;
    Access TSS descriptor in GDT;

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

TASK-STATE-SEGMENT:
    IF TSS DPL < CPL or RPL
    ORTSS segment selector local/global bit is set to local
    OR TSS descriptor indicates TSS not available
        THEN #GP(TSS selector);
    FI;
    IF TSS is not present
        THEN #NP(TSS selector);
    FI;
    IF IA-64 System Environment THEN IA-32_Intercept(Gate,CALL);
    SWITCH-TASKS (with nesting) to TSS
    IF EIP not within code segment limit
        THEN #GP(0);
    FI;
END;
```

CALL—Call Procedure (continued)

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Intercept	Gate Intercept for CALLs through CALL Gates, Task Gates and Task Segments
IA-32_Exception	Taken Branch Debug Exception if PSR.tb is 1

Protected Mode Exceptions

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

CALL—Call Procedure (continued)

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

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

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

Description

Double the size of the source operand by means of sign extension. The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the higher 16 bits of the EAX register.

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

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

Operation

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

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

CDQ—Convert Double to Quad

See entry for CWD/CDQ — Convert Word to Double/Convert Double to Quad.

CLC—Clear Carry Flag

Opcode	Instruction	Description
F8	CLC	Clear CF flag

Description

Clears the CF flag in the EFLAGS register.

Operation

```
CF ← 0;
```

Flags Affected

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

Exceptions (All Operating Modes)

None.

CLD—Clear Direction Flag

Opcode	Instruction	Description
FC	CLD	Clear DF flag

Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI).

Operation

DF ← 0;

Flags Affected

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

Exceptions (All Operating Modes)

None.

CLI—Clear Interrupt Flag

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

Description

Clears the IF flag in the EFLAGS register. No other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no affect on the generation of exceptions and NMI interrupts. **In the IA-64 System Environment, external interrupts are enabled for IA-32 instructions if PSR.i and (~CFLG.if or EFLAG.if) is 1 and for IA-64 instructions if PSR.i is 1.**

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

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

Notes:
X Don't care.
N Action in column 1 not taken.
Y Action in column 1 taken.

Operation

```
OLD_IF <- IF;
IF PE = 0 (* Executing in real-address mode *)
  THEN
      IF ← 0;
  ELSE
      IF VM = 0   (* Executing in protected mode *)
          THEN
              IF CR4.PVI = 1
                  THEN
                      IF CPL = 3
                      THEN
                          IF IOPL<3
                          THEN VIF <- 0;
                          ELSE IF <- 0;
                          FI;
                      ELSE (*CPL < 3*)
                          IF IOPL < CPL
                          THEN #GP(0);
                          ELSE IF <- 0;
```

CLI—Clear Interrupt Flag (continued)

```
                                FI;
                            FI;
                        ELSE (*CR4.PVI==0 *)
                            IF IOPL < CPL
                            THEN #GP(0);
                            ELSE IF <- 0;
                            FI;
                    FI;
                ELSE  (* Executing in Virtual-8086 mode  *)
                    IF IOPL = 3
                        THEN
                            IF  ← 0;
                        ELSE
                            IF CR4.VME= 0
                            THEN #GP(0);
                            ELSE VIF <- 0;
                            FI;
                    FI;
            FI;
FI;
IF IA-64 System Environment AND CFLG.ii AND IF != OLD_IF
   THEN IA-32_Intercept(System_Flag,CLI);
```

Flags Affected

The IF is cleared to 0 if the CPL is equal to or less than the IOPL; otherwise, the it is not affected. The other flags in the EFLAGS register are unaffected.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	System Flag Intercept Trap if CFLG.ii is 1 and the IF flag changes state.

Protected Mode Exceptions

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

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

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

CLTS—Clear Task-Switched Flag in CR0

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

Description

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

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the *Intel Architecture Software Developer's Manual, Volume 3* for more information about this flag.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,CLTS);
CR0(TS) ← 0;
```

Flags Affected

The TS flag in CR0 register is cleared.

Additional IA-64 System Environment Exceptions

IA-32_Intercept Mandatory Instruction Intercept fault.

Protected Mode Exceptions

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

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

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

CMC—Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement CF flag

Description

Complements the CF flag in the EFLAGS register.

Operation

CF ← NOT CF;

Flags Affected

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

Exceptions (All Operating Modes)

None.

CMOV*cc*—Conditional Move

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

CMOV*cc*—Conditional Move (continued)

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

Description

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

If the condition is false for the memory form, some processor implementations will initiate the load (and discard the loaded data), possible memory faults can be generated. Other processor models will not initiate the load and not generate any faults if the condition is false.

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

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

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

CMOV*cc*—Conditional Move (continued)

The CMOV*cc* instructions are new for the Pentium Pro processor family; however, they may not be supported by all the processors in the family. Software can determine if the CMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction (see "CPUID—CPU Identification" on page 5-68).

Operation

```
temp ← DEST
IF condition TRUE
  THEN
       DEST ← SRC
  ELSE
       DEST ← temp
FI;
```

Flags Affected

None.

If the condition is false for the memory form, some processor implementations will initiate the load (and discard the loaded data), possible memory faults can be generated. Other processor models will not initiate the load and not generate any faults if the condition is false.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

CMP—Compare Two Operands

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

Description

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

The CMP instruction is typically used in conjunction with a conditional jump (J*cc*), condition move (CMOV*cc*), or SET*cc* instruction. The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction.

Operation

```
temp ← SRC1 – SignExtend(SRC2);
ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

CMP—Compare Two Operands (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

CMPS/CMPSB/CMPSW/CMPSD—Compare String Operands

Opcode	Instruction	Description
A6	CMPS DS:(E)SI, ES:(E)DI	Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly
A7	CMPS DS:SI, ES:DI	Compares byte at address DS:SI with byte at address ES:DI and sets the status flags accordingly
A7	CMPS DS:ESI, ES:EDI	Compares byte at address DS:ESI with byte at address ES:EDI and sets the status flags accordingly
A6	CMPSB	Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly
A7	CMPSW	Compares byte at address DS:SI with byte at address ES:DI and sets the status flags accordingly
A7	CMPSD	Compares byte at address DS:ESI with byte at address ES:EDI and sets the status flags accordingly

Description

Compares the byte, word, or double word specified with the first source operand with the byte, word, or double word specified with the second source operand and sets the status flags in the EFLAGS register according to the results. The first source operand specifies the memory location at the address DS:ESI and the second source operand specifies the memory location at address ES:EDI. (When the operand-size attribute is 16, the SI and DI register are used as the source-index and destination-index registers, respectively.) The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

The CMPSB, CMPSW, and CMPSD mnemonics are synonyms of the byte, word, and doubleword versions of the CMPS instructions. They are simpler to use, but provide no type or segment checking. (For the CMPS instruction, “DS:ESI” and “ES:EDI” must be explicitly specified in the instruction.)

After the comparison, the ESI and EDI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the ESI and EDI register are incremented; if the DF flag is 1, the ESI and EDI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The CMPS, CMPSB, CMPSW, and CMPSD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made.

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

Operation

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

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

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

Virtual 8086 Mode Exceptions

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

CMPXCHG—Compare and Exchange

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

Description

Compares the value in the AL, AX, or EAX register (depending on the size of the operand) with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, or EAX register.

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

Operation

```
(* accumulator = AL, AX, or EAX, depending on whether *)
(* a byte, word, or doubleword comparison is being performed*)

IF IA-64 System Environment AND External_Atomic_Lock_Required AND DCR.lc
   THEN IA-32_Intercept(LOCK,CMPXCHG);
IF accumulator = DEST
   THEN
        ZF ← 1
        DEST ← SRC
   ELSE
        ZF ← 0
        accumulator ← DEST
FI;
```

Flags Affected

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

CMPXCHG—Compare and Exchange (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Intercept	Lock Intercept - If an external atomic bus lock is required to complete this operation and DCR.lc is 1, no atomic transaction occurs, this instruction is faulted and an IA-32_Intercept(Lock) fault is generated. The software lock handler is responsible for the emulation of this instruction.

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

Intel Architecture Compatibility

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

CMPXCHG8B—Compare and Exchange 8 Bytes

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

Description

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

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

Operation

```
IF IA-64 System Environment AND External_Atomic_Lock_Required AND DCR.lc
   THEN IA-32_Intercept(LOCK,CMPXCHG);

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

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Intercept	Lock Intercept - If an external atomic bus lock is required to complete this operation and DCR.lc is 1, no atomic transaction occurs, this instruction is faulted and an IA-32_Intercept(Lock) fault is generated. The software lock handler is responsible for the emulation of this instruction

CMPXCHG8B—Compare and Exchange 8 Bytes (continued)

Protected Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

Intel Architecture Compatibility

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

CPUID—CPU Identification

Opcode	Instruction	Description
0F A2	CPUID	EAX ← Processor identification information

Description

Provides processor identification information in registers EAX, EBX, ECX, and EDX. This information identifies Intel as the vendor, gives the family, model, and stepping of processor, feature information, and cache information as well as an Unique ID number for every manufactured processor. An input value loaded into the EAX register determines what information is returned, as shown in Table 5-4.

Table 5-4. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
0	EAX	Maximum CPUID Input Value
	EBX	756E6547H “Genu” (G in BL)
	ECX	49656E69H “ineI” (i in DL)
	EDX	6C65746EH “ntel (n in CL)”
1	EAX	Version Information (Family, Model, and Stepping ID)
	EBX	Reserved
	ECX	Reserved
	EDX	Feature Information
2	EAX	Cache Information
	EBX	Cache Information
	ECX	Cache Information
	EDX	Cache Information
3	EAX	Reserved
	EBX	Reserved
	ECX	Unique ID{31:0}
	EDX	Unique ID{63:32}

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

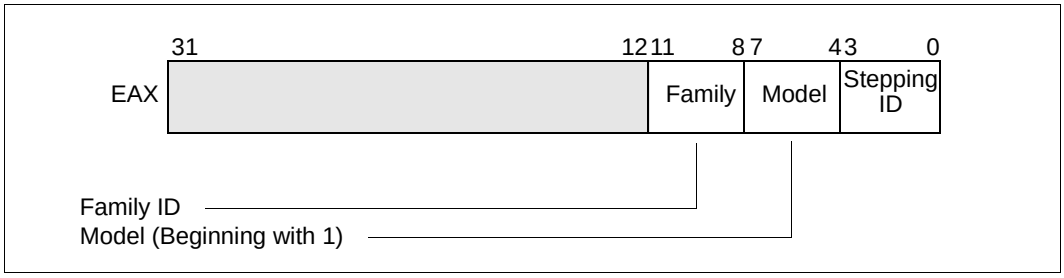
When the input value in register EAX is 0, the processor returns the highest value the CPUID instruction recognizes in the EAX register. For the Itanium processor, the highest recognized value is 3. A vendor identification string is returned in the EBX, EDX, and ECX registers. For Intel processors, the vendor identification string is “GenuineIntel” as follows:

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

When the input value is 1, the processor returns version information in the EAX register and feature information in the EDX register.

CPUID—CPU Identification (continued)

Figure 5-3. Version Information in Registers EAX



The version information consists of an Intel Architecture family identifier, a model identifier, a stepping ID, and a processor type.

See “Intel Application Note 485 — Intel Processor Identification with the CPUID Instruction” for more information on identifying earlier Intel Architecture processors. Intel releases information on stepping IDs as needed.

A feature flag set to 1 indicates the corresponding feature is supported. Software should identify Intel as the vendor to properly interpret the feature flags.

Note: IA-32 System Environment feature bits may be set in the IA-64 System Environment even if that feature cannot be used within the IA-64 System Environment.

Table 5-5. Feature Flags Returned in EDX Register

Bit	IA-32 System Environment Feature	IA-64 System Environment Feature	Description
0	FPU—Floating-point Unit on Chip	Available	Processor contains an FPU and executes the Intel387 instruction set.
1	VME—Virtual 8086 Mode Enhancements	Available	Processor supports the following virtual 8086 mode enhancements: • CR4.VME bit enables virtual 8086 mode extensions. • CR4.PVI bit enables protected-mode virtual interrupts. • Expansion of the TSS with the software indirection bitmap. • EFLAGS.VIF bit enables the virtual interrupt flag. • EFLAGS.VIP bit enables the virtual interrupt pending flag.
3	PSE—Page Size Extensions	Not Available IA-64 paging supports a wide range of page sizes.	Processor supports 4-Mbyte pages, including the CR4.PSE bit for enabling page size extensions, the modified bit in page directory entries (PDEs), page directory entries, and page table entries (PTEs).
4	TSC—Time Stamp Counter	Available	Processor supports the RDTSC (read time stamp counter) instruction, including the CR4.TSD bit that, along with the CPL, controls whether the time stamp counter can be read.

Table 5-5. Feature Flags Returned in EDX Register (Continued)

Bit	IA-32 System Environment Feature	IA-64 System Environment Feature	Description
5	MSR—Model Specific Registers	Not Available Use PAL interface to program model specific features.	Processor supports the RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions.
6	PAE—Physical Address Extension	Not Available. IA-64 always supports more than 32-bits of physical addressing	Processor supports physical addresses greater than 32 bits, the extended page-table-entry format, an extra level in the page translation tables, and 2-MByte pages. The CR4.PAE bit enables this feature. The number of address bits is implementation specific.
7	MCE—Machine Check Exception	Not Available. Processor uses PAL defined MCHK architecture.	Processor supports the CR4.MCE bit, enabling machine check exceptions. However, this feature does not define the model-specific implementations of machine-check error logging, reporting, or processor shutdowns. Machine-check exception handlers might have to check the processor version to do model-specific processing of the exception or check for the presence of the machine-check feature.
8	CX8—CMPXCHG8B Instruction	Available	Processor supports the CMPXCHG8B (compare and exchange 8 bytes) instruction.
9	APIC	Not Available. Replaced by the IA-64 interrupt mechanism.	Processor contains an on-chip Advanced Programmable Interrupt Controller (APIC) and it has been enabled and is available for use.
10	Reserved		returns zero
12	MTRR—Memory Type Range Registers	Not Available. Processor utilizes memory attributes from the IA-64 TLB	Processor supports machine-specific memory-type range registers (MTRRs). The MTRRs contains bit fields that indicate the processor's MTRR capabilities, including which memory types the processor supports, the number of variable MTRRs the processor supports, and whether the processor supports fixed MTRRs.
13	PGE—PTE Global Flag	Not available. Superceded by IA-64 virtual regions.	Processor supports the CR4.PGE flag enabling the global bit in both PTDEs and PTEs. These bits are used to indicate translation lookaside buffer (TLB) entries that are common to different tasks and need not be flushed when control register CR3 is written.
14	MCA—Machine Check Architecture	Not Available. Processor uses PAL defined MCHK architecture.	Processor supports the MCG_CAP (machine check global capability) MSR. The MCG_CAP register indicates how many banks of error reporting MSRs the processor supports.
15	CMOV—Conditional Move and Compare Instructions	Available	Processor supports the CMOV*cc* instruction and, if the FPU feature flag (bit 0) is also set, supports the FCMOV*cc* and FCOMI instructions.

Table 5-5. Feature Flags Returned in EDX Register (Continued)

Bit	IA-32 System Environment Feature	IA-64 System Environment Feature	Description
16	PSE-36 - 36-bit Page Size Extensions	Not Available. IA-64 always supports more than 32-bits of physical addressing	Indicates whether the processor supports 4-Mbyte pages that are capable of addressing physical memory beyond 4GB. This feature indicates that the upper four bits of the physical address of the 4-Mbyte page is encoded by bits 13-16 of the page directory entry.
17	PAT - Memory Attribute Palette	Not available for IA-64 paging. Superceded by IA-64 virtual regions.	Processors supports the IA-32 physical attribute table
18	PPN- Physical Processor Number	Available	Processor supports a Physical Processor Number for each manufactured processor
23	MMX - MMX™ Technology	Available	Processor supports the Intel Architecture MMX Technology
24	FXSR	Available	Processor supports the Streaming SIMD Extension FXRSTOR and FXSAVE instruction
25	XMM - Streaming SIMD Extension Technology	Available	Processor supports the Intel Architecture Streaming SIMD Extension
30	IA-64 Processor	Available	The processor is an IA-64 processor capable of executing the IA-64 instruction set. IA-32 application level software MUST also check with the running operating system to see if the system can also support IA-64 code before switching to the IA-64 instruction set.

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

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (cleared to 0) or is reserved (set to 1).

The 96-bit Processor Identifier may not be unique and there may be duplicate processors with the same 96-bit Processor Identifier, although processor vendors will make every attempt to ensure uniqueness.

The intended usage model is for processor and system inventory control.

Software that relies on the 96-bit Processor Identifier must include provisions to support processor replacement, processor upgrades, or system replacement.Otherwise, that software will fail.

The Physical Processor Number can be disabled until the next processor reset event via a model specific register. If Physical Processor Numbers are not supported or disabled, ECX and EDX will return zero.

CPUID—CPU Identification (continued)

CPUID performs instruction serialization and a memory fence operation.

The intended usage model is for processor and system inventory control.

Software that relies on the 96-bit Processor Identifier must include provisions to support processor replacement, processor upgrades, or system replacement.Otherwise, that software will fail.

The Physical Processor Number can be disabled until the next processor reset event via a model specific register. If Physical Processor Numbers are not supported or disabled, ECX and EDX will return zero.

CPUID performs instruction serialization and a memory fence operation.

Operation

```
CASE (EAX) OF
  EAX = 0:
       EAX ← highest input value understood by CPUID; (* 2 for Pentium Pro processor *)
       EBX ← Vendor identification string;
       EDX ← Vendor identification string;
       ECX ← Vendor identification string;
  BREAK;
  EAX = 1:
       EAX[3:0] ← Stepping ID;
       EAX[7:4] ← Model;
       EAX[11:8] ← Family;
       EAX[13:12] ← Processor type;
       EAX[31:12] ← Reserved;
       EBX ← Reserved;
       ECX ← Reserved;
       EDX ← Feature flags;
  BREAK;
  EAX = 2:
       EAX[7:0] ← N_Param_Descrip_Blocks = 1;
       EAX[31:8], EBX, ECX, EDX = cache and TLB parameters
  BREAK;
  EAX = 3:
       EAX,EBX ← Reserved;
       ECX ← PPN[31:0];
       EDX ← PPN[63:32];
  BREAK;
ESAC;

memory_fence();
instruction_serialize();
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

CPUID—CPU Identification (continued)

Exceptions (All Operating Modes)

None.

Intel Architecture Compatibility

The CPUID instruction is not supported in early models of the Intel486 processor or in any Intel Architecture processor earlier than the Intel486 processor. The ID flag in the EFLAGS register can be used to determine if this instruction is supported. If a procedure is able to set or clear this flag, the CPUID is supported by the processor running the procedure.

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

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

Description

Doubles the size of the operand in register AX or EAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX or EDX:EAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register.

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

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

Operation

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Flags Affected

None.

Exceptions (All Operating Modes)

None.

CWDE—Convert Word to Doubleword

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

DAA—Decimal Adjust AL after Addition

Opcode	Instruction	Description
27	DAA	Decimal adjust AL after addition

Description

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

Operation

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

Example

```
ADD AL, BL        Before: AL=79H  BL=35H EFLAGS(OSZAPC)=XXXXXX
                  After: AL=AEH  BL=35H EFLAGS(0SZAPC)=110000
DAA               Before: AL=79H  BL=35H EFLAGS(OSZAPC)=110000
                  After: AL=AEH  BL=35H EFLAGS(0SZAPC)=X00111
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

DAS—Decimal Adjust AL after Subtraction

Opcode	Instruction	Description
2F	DAS	Decimal adjust AL after subtraction

Description

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

Operation

```
IF (AL AND 0FH) > 9 OR AF = 1
  THEN
       AL ← AL – 6;
       CF ← CF OR BorrowFromLastSubtraction; (* CF OR borrow from AL ← AL – 6 *)
       AF ← 1;
  ELSE AF ← 0;
FI;
IF ((AL > 9FH) or CF = 1)
  THEN
       AL ← AL – 60H;
       CF ← 1;
  ELSE CF ← 0;
FI;
```

Example

```
SUB AL, BL        Before: AL=35H  BL=47H EFLAGS(OSZAPC)=XXXXXX
                  After:  AL=EEH  BL=47H EFLAGS(0SZAPC)=010111
DAA               Before: AL=EEH  BL=47H EFLAGS(OSZAPC)=010111
                  After:  AL=88H  BL=47H EFLAGS(0SZAPC)=X10111
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

DEC—Decrement by 1

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

Description

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

Operation

DEST ← DEST – 1;

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

DEC—Decrement by 1 (continued)

Virtual 8086 Mode Exceptions

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

DIV—Unsigned Divide

Opcode	Instruction	Description
F6 /6	DIV *r/m8*	Unsigned divide AX by *r/m8*; AL ← Quotient, AH ← Remainder
F7 /6	DIV *r/m16*	Unsigned divide DX:AX by *r/m16*; AX ← Quotient, DX ← Remainder
F7 /6	DIV *r/m32*	Unsigned divide EDX:EAX by *r/m32* doubleword; EAX ← Quotient, EDX ← Remainder

Description

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

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

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

Operation

```
IF SRC = 0
  THEN #DE; (* divide error *)
FI;
IF OpernadSize = 8 (* word/byte operation *)
  THEN
      temp ← AX / SRC;
      IF temp > FFH
          THEN #DE; (* divide error *) ;
          ELSE
              AL ← temp;
              AH ← AX MOD SRC;
      FI;
  ELSE
      IF OpernadSize = 16 (* doubleword/word operation *)
          THEN
              temp ← DX:AX / SRC;
              IF temp > FFFFH
                  THEN #DE; (* divide error *) ;
                  ELSE
                      AX ← temp;
                      DX ← DX:AX MOD SRC;
              FI;
```

DIV—Unsigned Divide (continued)

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

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

#DE	If the source operand (divisor) is 0.
	If the quotient is too large for the designated register.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.

DIV—Unsigned Divide (continued)

Virtual 8086 Mode Exceptions

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

ENTER—Make Stack Frame for Procedure Parameters

Opcode	Instruction	Description
C8 *iw* 00	ENTER *imm16*,0	Create a stack frame for a procedure
C8 *iw* 01	ENTER *imm16*,1	Create a nested stack frame for a procedure
C8 *iw* ib	ENTER *imm16,imm8*	Create a nested stack frame for a procedure

Description

Creates a stack frame for a procedure. The first operand (size operand) specifies the size of the stack frame (that is, the number of bytes of dynamic storage allocated on the stack for the procedure). The second operand (nesting level operand) gives the lexical nesting level (0 to 31) of the procedure. The nesting level determines the number of stack frame pointers that are copied into the "display area" of the new stack frame from the preceding frame. Both of these operands are immediate values.

The stack-size attribute determines whether the BP (16 bits) or EBP (32 bits) register specifies the current frame pointer and whether SP (16 bits) or ESP (32 bits) specifies the stack pointer.

The ENTER and companion LEAVE instructions are provided to support block structured languages. They do not provide a jump or call to another procedure; they merely set up a new stack frame for an already called procedure. An ENTER instruction is commonly followed by a CALL, JMP, or J*cc* instruction to transfer program control to the procedure being called.

If the nesting level is 0, the processor pushes the frame pointer from the EBP register onto the stack, copies the current stack pointer from the ESP register into the EBP register, and loads the ESP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack.

Operation

```
NestingLevel ← NestingLevel MOD 32
IF StackSize = 32
  THEN
      Push(EBP) ;
      FrameTemp ← ESP;
  ELSE (* StackSize = 16*)
      Push(BP);
      FrameTemp ← SP;
FI;
IF NestingLevel = 0
  THEN GOTO CONTINUE;
FI;
IF (NestingLevel > 0)
  FOR i ← 1 TO (NestingLevel – 1)
      DO
          IF OperandSize = 32
              THEN
                  IF StackSize = 32
                      EBP ← EBP – 4;
```

ENTER—Make Stack Frame for Procedure Parameters (continued)

```
                        Push([EBP]); (* doubleword push *)

                    ELSE (* StackSize = 16*)
                        BP ← BP – 4;
                        Push([BP]); (* doubleword push *)
                    FI;
                ELSE (* OperandSize = 16 *)
                    IF StackSize = 32
                        THEN
                            EBP ← EBP – 2;
                            Push([EBP]); (* word push *)
                        ELSE (* StackSize = 16*)
                            BP ← BP – 2;
                            Push([BP]); (* word push *)
                    FI;
            FI;
    OD;
    IF OperandSize = 32
        THEN
            Push(FrameTemp); (* doubleword push *)
        ELSE (* OperandSize = 16 *)
            Push(FrameTemp); (* word push *)
    FI;
    GOTO CONTINUE;
FI;
CONTINUE:
IF StackSize = 32
    THEN
        EBP ← FrameTemp
        ESP ← EBP – Size;
    ELSE (* StackSize = 16*)
        BP ← FrameTemp
        SP ← BP – Size;
FI;
END;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

IA-64 Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

ENTER—Make Stack Frame for Procedure Parameters (continued)

Protected Mode Exceptions

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

None.

F2XM1—Compute 2^x-1

Opcode	Instruction	Description
D9 F0	F2XM1	Replace ST(0) with $(2^{ST(0)} - 1)$

Description

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

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

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

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

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

Operation

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

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

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

F2XM1—Compute 2^X-1 (continued)

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FABS—Absolute Value

Opcode	Instruction	Description
D9 E1	FABS	Replace ST with its absolute value.

Description

Clears the sign bit of ST(0) to create the absolute value of the operand. The following table shows the results obtained when creating the absolute value of various classes of numbers.

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

Note:
F means finite-real number.

Operation

ST(0) ← |ST(0)|

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FADD/FADDP/FIADD—Add

Opcode	Instruction	Description
D8 /0	FADD *m32 real*	Add *m32real* to ST(0) and store result in ST(0)
DC /0	FADD *m64real*	Add *m64real* to ST(0) and store result in ST(0)
D8 C0+i	FADD ST(0), ST(i)	Add ST(0) to ST(i) and store result in ST(0)
DC C0+i	FADD ST(i), ST(0)	Add ST(i) to ST(0) and store result in ST(*i*)
DE C0+i	FADDP ST(i), ST(0)	Add ST(0) to ST(i), store result in ST(*i*), and pop the register stack
DE C1	FADDP	Add ST(0) to ST(1), store result in ST(1), and pop the register stack
DA /0	FIADD *m32int*	Add *m32int* to ST(0) and store result in ST(0)
DE /0	FIADD *m16int*	Add *m16int* to ST(0) and store result in ST(0)

Description

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

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

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

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

The FIADD instructions convert an integer source operand to extended-real format before performing the addition.

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

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

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

FADD/FADDP/FIADD—Add (continued)

.

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

Notes:
F means finite-real number.
L means integer.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FIADD
  THEN
      DEST ← DEST + ConvertExtendedReal(SRC);
  ELSE (* source operand is real number *)
      DEST ← DEST + SRC;
FI;
IF instruction = FADDP
  THEN
      PopRegisterStack;
FI;
```

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

FADD/FADDP/FIADD—Add (continued)

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of unlike sign.
#D	Result is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FBLD—Load Binary Coded Decimal

Opcode	Instruction	Description
DF /4	FBLD *m80 dec*	Convert BCD value to real and push onto the FPU stack.

Description

Converts the BCD source operand into extended-real format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of –0.

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

Operation

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

FPU Flags Affected

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

Floating-point Exceptions

#IS	Stack overflow occurred.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FBLD—Load Binary Coded Decimal (continued)

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FBSTP—Store BCD Integer and Pop

Opcode	Instruction	Description
DF /6	FBSTP m80bcd	Store ST(0) in m80bcd and pop ST(0).

Description

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

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

ST(0)	DEST
–•	*
–F < –1	–D
–1 < –F < –0	**
–0	–0
+0	+0
+0 < +F < +1	**
+F > +1	+D
+∞	*
NaN	*

Notes:
F means finite-real number.
D means packed-BCD number.
* indicates floating-point invalid-operation (#IA) exception.
** ±0 or ±1, depending on the rounding mode.

If the source value is too large for the destination format and the invalid-operation exception is not masked, an invalid-operation exception is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

If the source value is a quiet NaN, an invalid-operation exception is generated. Quiet NaNs do not normally cause this exception to be generated.

Operation

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

FBSTP—Store BCD Integer and Pop (continued)

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is empty; contains a NaN, ±∞, or unsupported format; or contains value that exceeds 18 BCD digits in length.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a segment register is being loaded with a segment selector that points to a nonwritable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

FBSTP—Store BCD Integer and Pop (continued)

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FCHS—Change Sign

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

Description

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

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

Note:
F means finite-real number.

Operation

SignBit(ST(0)) ← NOT (SignBit(ST(0)))

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FCLEX/FNCLEX—Clear Exceptions

Opcode	Instruction	Description
9B DB E2	FCLEX	Clear floating-point exception flags after checking for pending unmasked floating-point exceptions.
DB E2	FNCLEX	Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.

Description

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

Operation

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

FPU Flags Affected

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

Floating-point Exceptions

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set. /

FCMOV*cc*—Floating-point Conditional Move

Opcode	Instruction	Description
DA C0+i	FCMOVB ST(0), ST(*i*)	Move if below (CF=1)
DA C8+i	FCMOVE ST(0), ST(*i*)	Move if equal (ZF=1)
DA D0+i	FCMOVBE ST(0), ST(*i*)	Move if below or equal (CF=1 or ZF=1)
DA D8+i	FCMOVU ST(0), ST(*i*)	Move if unordered (PF=1)
DB C0+i	FCMOVNB ST(0), ST(*i*)	Move if not below (CF=0)
DB C8+i	FCMOVNE ST(0), ST(*i*)	Move if not equal (ZF=0)
DB D0+i	FCMOVNBE ST(0), ST(*i*)	Move if not below or equal (CF=0 and ZF=0)
DB D8+i	FCMOVNU ST(0), ST(*i*)	Move if not unordered (PF=0)

Description

Tests the status flags in the EFLAGS register and moves the source operand (second operand) to the destination operand (first operand) if the given test condition is true. The source operand is always in the ST(*i*) register and the destination operand is always ST(0).

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

A processor in the Pentium Pro processor family may not support the FCMOV*cc* instructions. Software can check if the FCMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction (see "CPUID—CPU Identification" on page 5-68). If both the CMOV and FPU feature bits are set, the FCMOV*cc* instructions are supported.

Operation

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

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.

Integer Flags Affected

None.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FCOM/FCOMP/FCOMPP—Compare Real

Opcode	Instruction	Description
D8 /2	FCOM *m32real*	Compare ST(0) with *m32real*.
DC /2	FCOM *m64real*	Compare ST(0) with *m64real*.
D8 D0+i	FCOM ST(i)	Compare ST(0) with ST(i).
D8 D1	FCOM	Compare ST(0) with ST(1).
D8 /3	FCOMP *m32real*	Compare ST(0) with *m32real* and pop register stack.
DC /3	FCOMP *m64real*	Compare ST(0) with m64real and pop register stack.
D8 D8+i	FCOMP ST(i)	Compare ST(0) with ST(i) and pop register stack.
D8 D9	FCOMP	Compare ST(0) with ST(1) and pop register stack.
DE D9	FCOMPP	Compare ST(0) with ST(1) and pop register stack twice.

Description

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that –0.0 = +0.0.

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

a. Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

This instruction checks the class of the numbers being compared. If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to “unordered.” If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

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

The FCOM instructions perform the same operation as the FUCOM instructions. The only difference is how they handle QNaN operands. The FCOM instructions raise an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value or is in an unsupported format. The FUCOM instructions perform the same operation as the FCOM instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

FCOM/FCOMP/FCOMPP—Compare Real (continued)

Operation

```
CASE (relation of operands) OF
   ST > SRC:        C3, C2, C0 ← 000;
   ST < SRC:        C3, C2, C0 ← 001;
   ST = SRC:        C3, C2, C0 ← 100;
ESAC;
IF ST(0) or SRC = NaN or unsupported format
   THEN
        #IA
        IF FPUControlWord.IM = 1
             THEN
                  C3, C2, C0 ← 111;
        FI;
FI;
IF instruction = FCOMP
   THEN
        PopRegisterStack;
FI;
IF instruction = FCOMPP
   THEN
        PopRegisterStack;
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, cleared to 0.
C0, C2, C3	See table on previous page.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

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

FCOM/FCOMP/FCOMPP—Compare Real (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Real and Set EFLAGS

Opcode	Instruction	Description
DB F0+i	FCOMI ST, ST(*i*)	Compare ST(0) with ST(*i*) and set status flags accordingly
DF F0+i	FCOMIP ST, ST(*i*)	Compare ST(0) with ST(*i*), set status flags accordingly, and pop register stack
DB E8+i	FUCOMI ST, ST(*i*)	Compare ST(0) with ST(*i*), check for ordered values, and set status flags accordingly
DF E8+i	FUCOMIP ST, ST(*i*)	Compare ST(0) with ST(*i*), check for ordered values, set status flags accordingly, and pop register stack

Description

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

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

a. Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

The FCOMI/FCOMIP instructions perform the same operation as the FUCOMI/FUCOMIP instructions. The only difference is how they handle QNaN operands. The FCOMI/FCOMIP instructions set the status flags to “unordered” and generate an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value (SNaN or QNaN) or is in an unsupported format.

The FUCOMI/FUCOMIP instructions perform the same operation as the FCOMI/FCOMIP instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

If invalid-operation exception is unmasked, the status flags are not set if the invalid-arithmetic-operand exception is generated.

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

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Real and Set EFLAGS (continued)

Operation

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

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Real and Set EFLAGS (continued)

Floating-point Exceptions

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

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set./

FCOS—Cosine

Opcode	Instruction	Description
D9 FF	FCOS	Replace ST(0) with its cosine

Description

Calculates the cosine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the cosine of various classes of numbers, assuming that neither overflow nor underflow occurs.

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

Notes:
F means finite-real number.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

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

FCOS—Cosine (continued)

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
	Undefined if C2 is 1.
C2	Set to 1 if source operand is outside the range -2^{63} to $+2^{63}$; otherwise, cleared to 0.
C0, C3	Undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

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

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FDECSTP—Decrement Stack-Top Pointer

Opcode	Instruction	Description
D9 F6	FDECSTP	Decrement TOP field in FPU status word.

Description

Subtracts one from the TOP field of the FPU status word (decrements the top-of-stack pointer). The contents of the FPU data registers and tag register are not affected.

Operation

```
IF TOP = 0
  THEN TOP ← 7;
  ELSE TOP ← TOP – 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0; otherwise, cleared to 0. The C0, C2, and C3 flags are undefined.

Floating-point Exceptions

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FDIV/FDIVP/FIDIV—Divide

Opcode	Instruction	Description
D8 /6	FDIV *m32real*	Divide ST(0) by *m32real* and store result in ST(0)
DC /6	FDIV *m64real*	Divide ST(0) by *m64real* and store result in ST(0)
D8 F0+i	FDIV ST(0), ST(*i*)	Divide ST(0) by ST(*i*) and store result in ST(0)
DC F8+i	FDIV ST(i), ST(0)	Divide ST(*i*) by ST(0) and store result in ST(*i*)
DE F8+i	FDIVP ST(*i*), ST(0)	Divide ST(*i*) by ST(0), store result in ST(*i*), and pop the register stack
DE F9	FDIVP	Divide ST(1) by ST(0), store result in ST(1), and pop the register stack
DA /6	FIDIV *m32int*	Divide ST(0) by *m32int* and store result in ST(0)
DE /6	FIDIV *m16int*	Divide ST(0) by *m64int* and store result in ST(0)

Description

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

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

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

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

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

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

FDIV/FDIVP/FIDIV—Divide (continued)

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

Notes:
F means finite-real number.
I means integer.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.
** indicates floating-point zero-divide (#Z) exception.

Operation

```
IF SRC = 0
  THEN
      #Z
  ELSE
      IF instruction is FIDIV
          THEN
              DEST ← DEST / ConvertExtendedReal(SRC);
          ELSE (* source operand is real number *)
              DEST ← DEST / SRC;
      FI;
FI;
IF instruction = FDIVP
  THEN
      PopRegisterStack
FI;
```

FPU Flags Affected

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

FDIV/FDIVP/FIDIV—Divide (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

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

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FDIVR/FDIVRP/FIDIVR—Reverse Divide

Opcode	Instruction	Description
D8 /7	FDIVR *m32real*	Divide *m32real* by ST(0) and store result in ST(0)
DC /7	FDIVR *m64real*	Divide *m64real* by ST(0) and store result in ST(0)
D8 F8+i	FDIVR ST(0), ST(i)	Divide ST(*i*) by ST(0) and store result in ST(0)
DC F0+i	FDIVR ST(i), ST(0)	Divide ST(0) by ST(*i*) and store result in ST(*i*)
DE F0+i	FDIVRP ST(i), ST(0)	Divide ST(0) by ST(*i*), store result in ST(*i*), and pop the register stack
DE F1	FDIVRP	Divide ST(0) by ST(1), store result in ST(1), and pop the register stack
DA /7	FIDIVR *m32int*	Divide *m32int* by ST(0) and store result in ST(0)
DE /7	FIDIVR *m16int*	Divide *m64int* by ST(0) and store result in ST(0)

Description

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

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

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

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

The FIDIVR instructions convert an integer source operand to extended-real format before performing the division.

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

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

FDIVR/FDIVRP/FIDIVR—Reverse Divide (continued)

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

Notes:
F means finite-real number.
I means integer.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.
** indicates floating-point zero-divide (#Z) exception.

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

Operation

```
IF DEST = 0
  THEN
      #Z
  ELSE
      IF instruction is FIDIVR
          THEN
              DEST ← ConvertExtendedReal(SRC) / DEST;
          ELSE (* source operand is real number *)
              DEST ← SRC / DEST;
      FI;
FI;
IF instruction = FDIVRP
  THEN
      PopRegisterStack
FI;
```

FPU Flags Affected

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

FDIVR/FDIVRP/FIDIVR—Reverse Divide (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

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

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FFREE—Free Floating-point Register

Opcode	Instruction	Description
DD C0+i	FFREE ST(*i*)	Sets tag for ST(*i*) to empty

Description

Sets the tag in the FPU tag register associated with register ST(*i*) to empty (11B). The contents of ST(*i*) and the FPU stack-top pointer (TOP) are not affected.

Operation

```
TAG(i) ← 11B;
```

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-point Exceptions

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FICOM/FICOMP—Compare Integer

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

Description

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

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

These instructions perform an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared. If either operand is a NaN or is in an undefined format, the condition flags are set to “unordered.”

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

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

Operation

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

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, set to 0.
C0, C2, C3	See table on previous page.

FICOM/FICOMP—Compare Integer (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

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

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FILD—Load Integer

Opcode	Instruction	Description
DF /0	FILD *m16int*	Push *m16int* onto the FPU register stack.
DB /0	FILD *m32int*	Push *m32int* onto the FPU register stack.
DF /5	FILD *m64int*	Push *m64int* onto the FPU register stack.

Description

Converts the signed-integer source operand into extended-real format and pushes the value onto the FPU register stack. The source operand can be a word, short, or long integer value. It is loaded without rounding errors. The sign of the source operand is preserved.

Operation

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

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FILD—Load Integer (continued)

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FINCSTP—Increment Stack-Top Pointer

Opcode	Instruction	Description
D9 F7	FINCSTP	Increment the TOP field in the FPU status register

Description

Adds one to the TOP field of the FPU status word (increments the top-of-stack pointer). The contents of the FPU data registers and tag register are not affected. This operation is not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.

Operation

```
IF TOP = 7
  THEN TOP ← 0;
  ELSE TOP ← TOP + 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0; otherwise, generates an #IS fault. The C0, C2, and C3 flags are undefined.

Floating-point Exceptions

#IS

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FINIT/FNINIT—Initialize Floating-point Unit

Opcode	Instruction	Description
9B DB E3	FINIT	Initialize FPU after checking for pending unmasked floating-point exceptions.
DB E3	FNINIT	Initialize FPU without checking for pending unmasked floating-point exceptions.

Description

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

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

Operation

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

FPU Flags Affected

C0, C1, C2, C3 cleared to 0.

Floating-point Exceptions

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FIST/FISTP—Store Integer

Opcode	Instruction	Description
DF /2	FIST *m16int*	Store ST(0) in *m16int*
DB /2	FIST *m32int*	Store ST(0) in *m32int*
DF /3	FISTP *m16int*	Store ST(0) in *m16int* and pop register stack
DB /3	FISTP *m32int*	Store ST(0) in *m32int* and pop register stack
DF /7	FISTP *m64int*	Store ST(0) in *m64int* and pop register stack

Description

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

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

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

ST(0)	DEST
–∞	*
–F < –1	–I
–1 < –F < –0	**
–0	0
+0	0
+0 < +F < +1	**
+F > +1	+I
+∞	*
NaN	*

Notes:
F means finite-real number.
I means integer.
* indicates floating-point invalid-operation (#IA) exception.
** ±0 or ±1, depending on the rounding mode.

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

If the value being stored is too large for the destination format, is an ∞, is a NaN, or is in an unsupported format and if the invalid-arithmetic-operand exception (#IA) is unmasked, an invalid-operation exception is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in the destination operand.

FIST/FISTP—Store Integer (continued)

Operation

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

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is too large for the destination format
	Source operand is a NaN value or unsupported format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FIST/FISTP—Store Integer (continued)

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FLD—Load Real

Opcode	Instruction	Description
D9 /0	FLD *m32real*	Push *m32real* onto the FPU register stack.
DD /0	FLD *m64real*	Push *m64real* onto the FPU register stack.
DB /5	FLD *m80real*	Push *m80real* onto the FPU register stack.
D9 C0+i	FLD ST(i)	Push ST(i) onto the FPU register stack.

Description

Pushes the source operand onto the FPU register stack. If the source operand is in single- or double-real format, it is automatically converted to the extended-real format before being pushed on the stack.

The FLD instruction can also push the value in a selected FPU register [ST(*i*)] onto the stack. Here, pushing register ST(0) duplicates the stack top.

Operation

```
IF SRC is ST(i)
  THEN
       temp ← ST(i)
TOP ← TOP – 1;
IF SRC is memory-operand
  THEN
       ST(0) ← ExtendedReal(SRC);
  ELSE (* SRC is ST(i) *)
       ST(0) ← temp;
```

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS	Stack overflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value. Does not occur if the source operand is in extended-real format.

FLD—Load Real (continued)

Protected Mode Exceptions

#GP(0)	If destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant

Opcode	Instruction	Description
D9 E8	FLD1	Push +1.0 onto the FPU register stack.
D9 E9	FLDL2T	Push $\log_2 10$ onto the FPU register stack.
D9 EA	FLDL2E	Push $\log_2 e$ onto the FPU register stack.
D9 EB	FLDPI	Push π onto the FPU register stack.
D9 EC	FLDLG2	Push $\log_{10} 2$ onto the FPU register stack.
D9 ED	FLDLN2	Push $\log_e 2$ onto the FPU register stack.
D9 EE	FLDZ	Push +0.0 onto the FPU register stack.

Description

Push one of seven commonly-used constants (in extended-real format) onto the FPU register stack. The constants that can be loaded with these instructions include +1.0, +0.0, $\log_2 10$, $\log_2 e$, π, $\log_{10} 2$, and $\log_e 2$. For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to external-real format. The inexact-result exception (#P) is not generated as a result of the rounding.

Operation

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

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.

Floating-point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant (continued)

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

Intel Architecture Compatibility Information

When the RC field is set to round-to-nearest, the FPU produces the same constants that is produced by the Intel 8087 and Intel287 math coprocessors.

FLDCW—Load Control Word

Opcode	Instruction	Description
D9 /5	FLDCW m2byte	Load FPU control word from *m2byte.*

Description

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

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmasks one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions. To avoid raising exceptions when changing FPU operating modes, clear any pending exceptions (using the FCLEX or FNCLEX instruction) before loading the new control word.

Operation

```
FPUControlWord ← SRC;
```

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-point Exceptions

None; however, this operation might unmask a pending exception in the FPU status word. That exception is then generated upon execution of the next waiting floating-point instruction.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FLDCW—Load Control Word (continued)

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FLDENV—Load FPU Environment

Opcode	Instruction	Description
D9 /4	FLDENV *m14/28byte*	Load FPU environment from *m14byte* or *m28byte.*

Description

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

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. See the *Intel Architecture Software Developer's Manual* for the layout in memory of the loaded environment, depending on the operating mode of the processor (protected or real) and the size of the current address attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FLDENV instruction should be executed in the same operating mode as the corresponding FSTENV/FNSTENV instruction.

If one or more unmasked exception flags are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions. To avoid generating exceptions when loading a new environment, clear all the exception flags in the FPU status word that is being loaded.

Operation

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

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

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

FLDENV—Load FPU Environment (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FMUL/FMULP/FIMUL—Multiply

Opcode	Instruction	Description
D8 /1	FMUL *m32real*	Multiply ST(0) by *m32real* and store result in ST(0)
DC /1	FMUL *m64real*	Multiply ST(0) by *m64real* and store result in ST(0)
D8 C8+i	FMUL ST(0), ST(i)	Multiply ST(0) by ST(i) and store result in ST(0)
DC C8+i	FMUL ST(*i*), ST(0)	Multiply ST(*i*) by ST(0) and store result in ST(*i*)
DE C8+i	FMULP ST(*i*), ST(0)	Multiply ST(*i*) by ST(0), store result in ST(*i*), and pop the register stack
DE C9	FMULP	Multiply ST(0) by ST(1), store result in ST(0), and pop the register stack
DA /1	FIMUL *m32int*	Multiply *m32int* by ST(0) and store result in ST(0)
DE /1	FIMUL *m16int*	Multiply *m16int* by ST(0) and store result in ST(0)

Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

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

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

The FIMUL instructions convert an integer source operand to extended-real format before performing the multiplication.

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

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

FMUL/FMULP/FIMUL—Multiply (continued)

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

Notes:
F means finite-real number.
I means Integer.
* indicates invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FIMUL
  THEN
      DEST ← DEST * ConvertExtendedReal(SRC);
  ELSE (* source operand is real number *)
      DEST ← DEST * SRC;
FI;
IF instruction = FMULP
  THEN
      PopRegisterStack
FI;
```

FPU Flags Affected

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

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	One operand is ±0 and the other is ±∞.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

FMUL/FMULP/FIMUL—Multiply (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FNOP—No Operation

Opcode	Instruction	Description
D9 D0	FNOP	No operation is performed.

Description

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register.

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-point Exceptions

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FPATAN—Partial Arctangent

Opcode	Instruction	Description
D9 F3	FPATAN	Replace ST(1) with arctan(ST(1)/ST(0)) and pop the register stack

Description

Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than +π.

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

Table 5-6. FPATAN Zeros and NaNs

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

Note:
F means finite-real number.

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

Operation

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

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FPATAN—Partial Arctangent (continued)

Floating-point Exceptions

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

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

Intel Architecture Compatibility Information

The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

$0 \leq |ST(1)| < |ST(0)| < +\infty$

FPREM—Partial Remainder

Opcode	Instruction	Description
D9 F8	FPREM	Replace ST(0) with the remainder obtained on dividing ST(0) by ST(1)

Description

Computes the remainder obtained on dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or *modulus*), and stores the result in ST(0). The remainder represents the following value:

Remainder = ST(0) – (N ∗ ST(1))

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

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

Table 5-7. FPREM Zeros and NaNs

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

Notes:
F means finite-real number.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.
** indicates floating-point zero-divide (#Z) exception.

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

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

FPREM—Partial Remainder (continued)

The FPREM instruction gets its name “partial remainder” because of the way it computes the remainder. This instructions arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the *partial remainder*. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared.

Note: While executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.

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

Operation

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

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FPREM—Partial Remainder (continued)

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus is 0, dividend is ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FPREM1—Partial Remainder

Opcode	Instruction	Description
D9 F5	FPREM1	Replace ST(0) with the IEEE remainder obtained on dividing ST(0) by ST(1)

Description

Computes the IEEE remainder obtained on dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or *modulus*), and stores the result in ST(0). The remainder represents the following value:

Remainder = ST(0) – (N ∗ ST(1))

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

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

Table 5-8. FPREM1 Zeros and NaNs

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

Notes:
F means finite-real number.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.
** indicates floating-point zero-divide (#Z) exception.

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

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

FPREM1—Partial Remainder (continued)

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

Note: While executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.

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

Operation

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

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FPREM1—Partial Remainder (continued)

Floating-point Exceptions

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

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FPTAN—Partial Tangent

Opcode	Instruction	Clocks	Description
D9 F2	FPTAN	17-173	Replace ST(0) with its tangent and push 1 onto the FPU stack.

Description

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

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

Notes:
F means finite-real number.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

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

Operation

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

FPTAN—Partial Tangent (continued)

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C2	Set to 1 if source operand is outside the range -2^{63} to $+2^{63}$; otherwise, cleared to 0.
C0, C3	Undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

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

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FRNDINT—Round to Integer

Opcode	Instruction	Description
D9 FC	FRNDINT	Round ST(0) to an integer.

Description

Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is ∞, the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

Operation

```
ST(0) ← RoundToIntegralValue(ST(0));
```

FPU Flags Affected

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

Floating-point Exceptions

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FRSTOR—Restore FPU State

Opcode	Instruction	Description
DD /4	FRSTOR *m94/108byte*	Load FPU state from *m94byte* or *m108byte*.

Description

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

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. See the *Intel Architecture Software Developer's Manual* for the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the size of the current address attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

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

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

Operation

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

FPU Flags Affected

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

Floating-point Exceptions

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

FRSTOR—Restore FPU State (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FSAVE/FNSAVE—Store FPU State

Opcode	Instruction	Description
9B DD /6	FSAVE *m94/108byte*	Store FPU state to *m94byte* or *m108byte* after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE *m94/108byte*	Store FPU environment to *m94byte* or *m108byte* without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

Description

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

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. See the *Intel Architecture Software Developer's Manual* for the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the size of the current address attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed.

After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see "FINIT/FNINIT—Initialize Floating-point Unit" on page 5-122).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a "clean" FPU to a procedure.

Operation

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

FSAVE/FNSAVE—Store FPU State (continued)

```
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

FPU Flags Affected

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

Floating-point Exceptions

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

FSAVE/FNSAVE—Store FPU State (continued)

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Intel Architecture Compatibility Information

For Intel math coprocessors and FPUs prior to the Intel Pentium™ processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps insure that the storage operation has been completed.

FSCALE—Scale

Opcode	Instruction	Description
D9 FD	FSCALE	Scale ST(0) by ST(1).

Description

Multiplies the destination operand by 2 to the power of the source operand and stores the result in the destination operand. This instruction provides rapid multiplication or division by integral powers of 2. The destination operand is a real value that is located in register ST(0). The source operand is the nearest integer value that is smaller than the value in the ST(1) register (that is, the value in register ST(1) is truncate toward 0 to its nearest integer value to form the source operand). The actual scaling operation is performed by adding the source operand (integer value) to the exponent of the value in register ST(0). The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

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

Notes:
F means finite-real number.
N means integer.

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

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

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

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction returns the recreated value to the FPU register where it originally resided.

FSCALE—Scale (continued)

Operation

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

FPU Flags Affected

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

Floating-point Exceptions

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FSIN—Sine

Opcode	Instruction	Description
D9 FE	FSIN	Replace ST(0) with its sine.

Description

Calculates the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

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

Notes:
F means finite-real number.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

```
IF I

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

FSIN—Sine (continued)

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C2	Set to 1 if source operand is outside the range -2^{63} to $+2^{63}$; otherwise, cleared to 0.
C0, C3	Undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

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

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FSINCOS—Sine and Cosine

Opcode	Instruction	Description
D9 FB	FSINCOS	Compute the sine and cosine of ST(0); replace ST(0) with the sine, and push the cosine onto the register stack.

Description

Computes both the sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

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

Notes:
F means finite-real number.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

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

FSINCOS—Sine and Cosine (continued)

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 of stack overflow occurs.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C2	Set to 1 if source operand is outside the range -2^{63} to $+2^{63}$; otherwise, cleared to 0.
C0, C3	Undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

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

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FSQRT—Square Root

Opcode	Instruction	Description
D9 FA	FSQRT	Calculates square root of ST(0) and stores the result in ST(0)

Description

Calculates the square root of the source value in the ST(0) register and stores the result in ST(0).

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

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

Notes:
F means finite-real number.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
ST(0) ← SquareRoot(ST(0));
```

FPU Flags Affected

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

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
	Source operand is a negative value (except for –0).
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

FSQRT—Square Root (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FST/FSTP—Store Real

Opcode	Instruction	Description
D9 /2	FST *m32real*	Copy ST(0) to *m32real*
DD /2	FST *m64real*	Copy ST(0) to *m64real*
DD D0+i	FST ST(*i*)	Copy ST(0) to ST(i)
D9 /3	FSTP *m32real*	Copy ST(0) to *m32real* and pop register stack
DD /3	FSTP *m64real*	Copy ST(0) to *m64real* and pop register stack
DB /7	FSTP *m80real*	Copy ST(0) to *m80real* and pop register stack
DD D8+i	FSTP ST(*i*)	Copy ST(0) to ST(*i*) and pop register stack

Description

The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU registers stack. When storing the value in memory, the value is converted to single- or double-real format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also stores values in memory in extended-real format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single- or double-real, the significand of the value being stored is rounded to the width of the destination (according to rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

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

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

Operation

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

FST/FSTP—Store Real (continued)

FPU Flags Affected

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

Floating-point Exceptions

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

FST/FSTP—Store Real (continued)

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FSTCW/FNSTCW—Store Control Word

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

Description

Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

Operation

```
DEST ← FPUControlWord;
```

FPU Flags Affected

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

Floating-point Exceptions

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FSTCW/FNSTCW—Store Control Word (continued)

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FSTENV/FNSTENV—Store FPU Environment

Opcode	Instruction	Description
9B D9 /6	FSTENV *m14/28byte*	Store FPU environment to *m14byte* or *m28byte* after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV *m14/28byte*	Store FPU environment to *m14byte* or *m28byte* without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.

Description

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. See the *Intel Architecture Software Developer's Manual* for the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the size of the current address attribute (16-bit or 32-bit). (In virtual-8086 mode, the real mode layouts are used.)

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not.The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the procedure stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

Operation

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

FPU Flags Affected

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

Floating-point Exceptions

None.

FSTENV/FNSTENV—Store FPU Environment (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FSTSW/FNSTSW—Store Status Word

Opcode	Instruction	Description
9B DD /7	FSTSW *m2byte*	Store FPU status word at *m2byte* after checking for pending unmasked floating-point exceptions.
9B DF E0	FSTSW AX	Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.
DD /7	FNSTSW *m2byte*	Store FPU status word at *m2byte* without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW AX	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

Description

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

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

Operation

```
DEST ← FPUStatusWord;
```

FPU Flags Affected

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

Floating-point Exceptions

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

FSTSW/FNSTSW—Store Status Word (continued)

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FSUB/FSUBP/FISUB—Subtract

Opcode	Instruction	Description
D8 /4	FSUB *m32real*	Subtract *m32real* from ST(0) and store result in ST(0)
DC /4	FSUB *m64real*	Subtract *m64real* from ST(0) and store result in ST(0)
D8 E0+i	FSUB ST(0), ST(*i*)	Subtract ST(*i*) from ST(0) and store result in ST(0)
DC E8+i	FSUB ST(*i*), ST(0)	Subtract ST(0) from ST(*i*) and store result in ST(*i*)
DE E8+i	FSUBP ST(*i*), ST(0)	Subtract ST(0) from ST(*i*), store result in ST(*i*), and pop register stack
DE E9	FSUBP	Subtract ST(0) from ST(1), store result in ST(1), and pop register stack
DA /4	FISUB *m32int*	Subtract *m32int* from ST(0) and store result in ST(0)
DE /4	FISUB *m16int*	Subtract *m16int* from ST(0) and store result in ST(0)

Description

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

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

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

The FISUB instructions convert an integer source operand to extended-real format before performing the subtraction.

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

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

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

FSUB/FSUBP/FISUB—Subtract (continued)

Table 5-9. FSUB Zeros and NaNs

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

Notes:
F means finite-real number.
I means integer.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FISUB
  THEN
      DEST ← DEST – ConvertExtendedReal(SRC);
  ELSE (* source operand is real number *)
      DEST ← DEST – SRC;
FI;
IF instruction = FSUBP
  THEN
      PopRegisterStack
FI;
```

FPU Flags Affected

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

Floating-point Exceptions

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

FSUB/FSUBP/FISUB—Subtract (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FSUBR/FSUBRP/FISUBR—Reverse Subtract

Opcode	Instruction	Description
D8 /5	FSUBR *m32real*	Subtract ST(0) from *m32real* and store result in ST(0)
DC /5	FSUBR *m64real*	Subtract ST(0) from *m64real* and store result in ST(0)
D8 E8+i	FSUBR ST(0), ST(i)	Subtract ST(0) from ST(*i*) and store result in ST(0)
DC E0+i	FSUBR ST(i), ST(0)	Subtract ST(*i*) from ST(0) and store result in ST(*i*)
DE E0+i	FSUBRP ST(i), ST(0)	Subtract ST(0) from ST(*i*), store result in ST(*i*), and pop register stack
DE E1	FSUBRP	Subtract ST(1) from ST(0), store result in ST(1), and pop register stack
DA /5	FISUBR *m32int*	Subtract ST(0) from *m32int* and store result in ST(0)
DE /5	FISUBR *m16int*	Subtract ST(0) from *m16int* and store result in ST(0)

Description

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

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

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

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

The FISUBR instructions convert an integer source operand to extended-real format before performing the subtraction.

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

FSUBR/FSUBRP/FISUBR—Reverse Subtract (continued)

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

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

Table 5-10. FSUBR Zeros and NaNs

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

Notes:
F means finite-real number.
I means integer.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FISUBR
  THEN
      DEST ← ConvertExtendedReal(SRC) – DEST;
  ELSE (* source operand is real number *)
      DEST ← SRC – DEST;
FI;
IF instruction = FSUBRP
  THEN
      PopRegisterStack
FI;
```

FPU Flags Affected

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

FSUBR/FSUBRP/FISUBR—Reverse Subtract (continued)

Floating-point Exceptions

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FTST—TEST

Opcode	Instruction	Description
D9 E4	FTST	Compare ST(0) with 0.0.

Description

Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

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

This instruction performs an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see “FXAM—Examine” on page 5-182). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to “unordered.”)

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

Operation

```
CASE (relation of operands) OF
   Not comparable:  C3, C2, C0 ← 111;
   ST(0) > 0.0:     C3, C2, C0 ← 000;
   ST(0) < 0.0:     C3, C2, C0 ← 001;
   ST(0) = 0.0:     C3, C2, C0 ← 100;
ESAC;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, cleared to 0.
C0, C2, C3	See above table.

Floating-point Exceptions

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FUCOM/FUCOMP/FUCOMPP—Unordered Compare Real

Opcode	Instruction	Description
DD E0+i	FUCOM ST(i)	Compare ST(0) with ST(i)
DD E1	FUCOM	Compare ST(0) with ST(1)
DD E8+i	FUCOMP ST(i)	Compare ST(0) with ST(i) and pop register stack
DD E9	FUCOMP	Compare ST(0) with ST(1) and pop register stack
DA E9	FUCOMPP	Compare ST(0) with ST(1) and pop register stack twice

Description

Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that –0.0 = +0.0.

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

a. Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see "FXAM—Examine" on page 5-182). The FUCOM instructions perform the same operation as the FCOM instructions. The only difference is that the FUCOM instruction raises the invalid-arithmetic-operand exception (#IA) only when either or both operands is an SNaN or is in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM instruction raises an invalid-operation exception when either or both of the operands is a NaN value of any kind or is in an unsupported format.

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

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

Operation

```
CASE (relation of operands) OF
   ST > SRC:        C3, C2, C0 ← 000;
   ST < SRC:        C3, C2, C0 ← 001;
   ST = SRC:        C3, C2, C0 ← 100;
ESAC;
IF ST(0) or SRC = QNaN, but not SNaN or unsupported format
```

FUCOM/FUCOMP/FUCOMPP—Unordered Compare Real (continued)

```
  THEN
       C3, C2, C0 ← 111;
  ELSE (* ST(0) or SRC is SNaN or unsupported format *)
       #IA;
       IF FPUControlWord.IM = 1
            THEN
                 C3, C2, C0 ← 111;
       FI;
FI;
IF instruction = FUCOMP
  THEN
       PopRegisterStack;
FI;
IF instruction = FUCOMPP
  THEN
       PopRegisterStack;
       PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
C0, C2, C3	See table on previous page.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception.
#D	One or both operands are denormal values.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FWAIT—Wait

See entry for WAIT.

FXAM—Examine

Opcode	Instruction	Description
D9 E5	FXAM	Classify value or number in ST(0)

Description

Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

.

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

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

Operation

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

FPU Flags Affected

C1	Sign of value in ST(0).
C0, C2, C3	See table above.

Floating-point Exceptions

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FXAM—Examine (continued)

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FXCH—Exchange Register Contents

Opcode	Instruction	Description
D9 C8+i	FXCH ST(i)	Exchange the contents of ST(0) and ST(*i*)
D9 C9	FXCH	Exchange the contents of ST(0) and ST(1)

Description

Exchanges the contents of registers ST(0) and ST(*i*). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

Operation

```
IF number-of-operands is 1
  THEN
       temp ← ST(0);
       ST(0) ← SRC;
       SRC ← temp;
  ELSE
       temp ← ST(0);
       ST(0) ← ST(1);
       ST(1) ← temp;
```

FPU Flags Affected

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

Floating-point Exceptions

#IS	Stack underflow occurred.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

FXCH—Exchange Register Contents (continued)

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FXTRACT—Extract Exponent and Significand

Opcode	Instruction	Description
D9 F4	FXTRACT	Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack.

Description

Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a real number. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand's true (unbiased) exponent expressed as a real number. (The operation performed by this instruction is a superset of the IEEE-recommended logb(*x*) function.)

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

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

Operation

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

FPU Flags Affected

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

Floating-point Exceptions

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FXTRACT—Extract Exponent and Significand (continued)

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FYL2X—Compute y × $\log_2 x$

Opcode	Instruction	Description
D9 F1	FYL2X	Replace ST(1) with (ST(1) ∗ $\log_2$ST(0)) and pop the register stack

Description

Calculates (ST(1) ∗ $\log_2$ (ST(0))), stores the result in resister ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

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

Table 5-11. FYL2X Zeros and NaNs

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

Notes:
F means finite-real number.
* indicates floating-point invalid-operation (#IA) exception.
** indicates floating-point zero-divide (#Z) exception.

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

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

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

Operation

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

FPU Flags Affected

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

FYL2X—Compute $y \times \log_2 x$ (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

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

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FYL2XP1—Compute y ∗ $\log_2$(x +1)

Opcode	Instruction	Description
D9 F9	FYL2XP1	Replace ST(1) with ST(1) ∗ $\log_2$(ST(0) + 1.0) and pop the register stack

Description

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

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

The source operand in ST(1) can range from –∞ to +∞. If either of the source operands is outside its acceptable range, the result is undefined and no exception is generated.

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

Table 5-12. FYL2XP1 Zeros and NaNs

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

Notes:
F means finite-real number.
* indicates floating-point invalid-operation (#IA) exception.

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

$$\text{scale factor} = \log_n 2$$

Operation

```
I

ST(1) ← ST(1) ∗ log2(ST(0) + 1.0);
PopRegisterStack;
```

FYL2XP1—Compute y ∗ $\log_2(x+1)$ (continued)

FPU Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

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

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

HLT—Halt

Opcode	Instruction	Description
F4	HLT	Halt

Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt, NMI, or a reset will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

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

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,HALT);
Enter Halt state;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0)	If the current privilege level is not 0.

IDIV—Signed Divide

Opcode	Instruction	Description
F6 /7	IDIV *r/m8*	Signed divide AX (where AH must contain sign-extension of AL) by *r/m* byte. (Results: AL=Quotient, AH=Remainder)
F7 /7	IDIV *r/m16*	Signed divide DX:AX (where DX must contain sign-extension of AX) by *r/m* word. (Results: AX=Quotient, DX=Remainder)
F7 /7	IDIV *r/m32*	Signed divide EDX:EAX (where EDX must contain sign-extension of EAX) by *r/m* doubleword. (Results: EAX=Quotient, EDX=Remainder)

Description

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

Table 5-13. IDIV Operands

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	–128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	–32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{32} - 1$

Non-integral results are truncated (chopped) towards 0. The sign of the remainder is always the same as the sign of the dividend. The absolute value of the remainder is always less than the absolute value of the divisor. Overflow is indicated with the #DE (divide error) exception rather than with the OF flag.

Operation

```
IF SRC = 0
  THEN #DE; (* divide error *)
FI;
IF OpernadSize = 8 (* word/byte operation *)
  THEN
      temp ← AX / SRC; (* signed division *)
      IF (temp > 7FH) OR (temp < 80H)
      (* if a positive result is greater than 7FH or a negative result is less than 80H *)
          THEN #DE; (* divide error *) ;
          ELSE
              AL ← temp;
              AH ← AX SignedModulus SRC;
      FI;
  ELSE
      IF OpernadSize = 16 (* doubleword/word operation *)
          THEN
```

IDIV—Signed Divide (continued)

```
                temp ← DX:AX / SRC; (* signed division *)
                IF (temp > 7FFFH) OR (temp < 8000H)
                (* if a positive result is greater than 7FFFH *)
                (* or a negative result is less than 8000H *)
                    THEN #DE; (* divide error *) ;
                    ELSE
                        AX ← temp;
                        DX ← DX:AX SignedModulus SRC;
                FI;
            ELSE (* quadword/doubleword operation *)
                temp ← EDX:EAX / SRC; (* signed division *)
                IF (temp > 7FFFFFFFH) OR (temp < 80000000H)
                (* if a positive result is greater than 7FFFFFFFH *)
                (* or a negative result is less than 80000000H *)
                    THEN #DE; (* divide error *) ;
                    ELSE
                        EAX ← temp;
                        EDX ← EDXE:AX SignedModulus SRC;
                FI;
        FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

IDIV—Signed Divide (continued)

Real Address Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

IMUL—Signed Multiply

Opcode	Instruction	Description
F6 /5	IMUL *r/m8*	AX← AL ∗ *r/m* byte
F7 /5	IMUL *r/m16*	DX:AX ← AX ∗ *r/m* word
F7 /5	IMUL *r/m32*	EDX:EAX ← EAX ∗ *r/m* doubleword
0F AF /*r*	IMUL *r16,r/m16*	word register ← word register ∗ *r/m* word
0F AF /*r*	IMUL *r32,r/m32*	doubleword register ← doubleword register ∗ *r/m* doubleword
6B /*r ib*	IMUL *r16,r/m16,imm8*	word register ← *r/m16* ∗ sign-extended immediate byte
6B /*r ib*	IMUL *r32,r/m32,imm8*	doubleword register ← *r/m32* ∗ sign-extended immediate byte
6B /*r ib*	IMUL *r16,imm8*	word register ← word register ∗ sign-extended immediate byte
6B /*r ib*	IMUL *r32,imm8*	doubleword register ← doubleword register ∗ sign-extended immediate byte
69 /*r iw*	IMUL *r16,r/m16,imm16*	word register ← *r/m16* ∗ immediate word
69 /*r id*	IMUL *r32,r/m32,imm32*	doubleword register ← *r/m32* ∗ immediate doubleword
69 /*r iw*	IMUL *r16,imm16*	word register ← *r/m16* ∗ immediate word
69 /*r id*	IMUL *r32,imm32*	doubleword register ← *r/m32* ∗ immediate doubleword

Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form.** This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, or EAX register (depending on the operand size) and the product is stored in the AX, DX:AX, or EDX:EAX registers, respectively.
- **Two-operand form.** With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The product is then stored in the destination operand location.
- **Three-operand form.** This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The product is then stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when significant bits are carried into the upper half of the result. The CF and OF flags are cleared when the result fits exactly in the lower half of the result.

IMUL—Signed Multiply (continued)

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

Operation

```
IF (NumberOfOperands = 1)
  THEN IF (OperandSize = 8)
       THEN
            AX ← AL ∗ SRC  (* signed multiplication *)
            IF ((AH = 00H) OR (AH = FFH))
                 THEN CF = 0; OF = 0;
                 ELSE CF = 1; OF = 1;
            FI;
       ELSE IF OperandSize = 16
            THEN
                 DX:AX ← AX ∗ SRC  (* signed multiplication *)
                 IF ((DX = 0000H) OR (DX = FFFFH))
                      THEN CF = 0; OF = 0;
                      ELSE CF = 1; OF = 1;
                 FI;
            ELSE (* OperandSize = 32 *)
                 EDX:EAX ← EAX ∗ SRC  (* signed multiplication *)
                 IF ((EDX = 00000000H) OR (EDX = FFFFFFFFH))
                      THEN CF = 0; OF = 0;
                      ELSE CF = 1; OF = 1;
                 FI;
       FI;
  ELSE IF (NumberOfOperands = 2)
       THEN
            temp ← DEST ∗ SRC    (* signed multiplication; temp is double DEST size*)
            DEST ← DEST ∗ SRC  (* signed multiplication *)
            IF temp ≠ DEST
                 THEN CF = 1; OF = 1;
                 ELSE CF = 0; OF = 0;
            FI;

       ELSE (* NumberOfOperands = 3 *)
            DEST ← SRC1 ∗ SRC2   (* signed multiplication *)
            temp ← SRC1 ∗ SRC2    (* signed multiplication; temp is double SRC1 size *)
            IF temp ≠ DEST
                 THEN CF = 1; OF = 1;
                 ELSE CF = 0; OF = 0;
            FI;
  FI;
FI;
```

IMUL—Signed Multiply (continued)

Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

IN—Input from Port

Opcode	Instruction	Description
E4 *ib*	IN AL,*imm8*	Input byte from *imm8* I/O port address into AL
E5 *ib*	IN AX,*imm8*	Input byte from *imm8* I/O port address into AX
E5 *ib*	IN EAX,*imm8*	Input byte from *imm8* I/O port address into EAX
EC	IN AL,DX	Input byte from I/O port in DX into AL
ED	IN AX,DX	Input word from I/O port in DX into AX
ED	IN EAX,DX	Input doubleword from I/O port in DX into EAX

Description

Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space.

I/O transactions are performed after all prior data memory operations. No subsequent data memory operations can pass an I/O transaction.

In the IA-64 System Environment, I/O port references are mapped into the 64-bit virtual address pointed to by the IOBase register, with four ports per 4K-byte virtual page. Operating systems can utilize the IA-64 TLB to grant or deny permission to any four I/O ports. The I/O port space can be mapped into any arbitrary 64-bit physical memory location by operating system code. If CFLG.io is 1 and CPL>IOPL, the TSS is consulted for I/O permission. If CFLG.io is 0 or CPL<=IOPL, permission is granted regardless of the state of the TSS I/O permission bitmap (the bitmap is not referenced).

If the referenced I/O port is mapped to an unimplemented virtual address (via the I/O Base register) or if data translations are disabled (PSR.dt is 0) a GPFault is generated on the referencing IN instruction.

Operation

```
IF ((PE = 1) AND ((VM = 1) OR (CPL > IOPL)))
  THEN (* Protected mode or virtual-8086 mode with CPL > IOPL *)
       IF (CFLG.io AND Any I/O Permission Bit for I/O port being accessed = 1)
            THEN #GP(0);
       FI;
  ELSE ( * Real-address mode or protected mode with CPL ≤ IOPL *)
  (* or virtual-8086 mode with all I/O permission bits for I/O port cleared *)
```

IN—Input from Port (continued)

```
FI;
IF (IA-64_System_Environment THEN
   SRC_VA = IOBase | (Port{15:2}<<12) | Port{11:0};
   SRC_PA = translate(SRC_VA);
   DEST ← [SRC_PA]; (* Reads from I/O port *)
FI;

memory_fence();
DEST <-SRC;
memory-fence();
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Exception	Debug traps for data breakpoints and single step
IA-32_Exception	Alignment faults
#GP(0)	Referenced Port is to an unimplemented virtual address or PSR.dt is zero.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1 **when CFLG.io is 1.**

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

INC—Increment by 1

Opcode	Instruction	Description
FE /0	INC *r/m8*	Increment *r/m* byte by 1
FF /0	INC *r/m16*	Increment *r/m* word by 1
FF /0	INC *r/m32*	Increment *r/m* doubleword by 1
40+ *rw*	INC *r16*	Increment word register by 1
40+ *rd*	INC *r32*	Increment doubleword register by 1

Description

Adds 1 to the operand, while preserving the state of the CF flag. The source operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform a increment operation that does updates the CF flag.)

Operation

```
DEST ← DEST – 1;
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

INC—Increment by 1 (continued)

Virtual 8086 Mode Exceptions

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

INS/INSB/INSW/INSD—Input from Port to String

Opcode	Instruction	Description
6C	INS ES:(E)DI, DX	Input byte from port DX into ES:(E)DI
6D	INS ES:DI, DX	Input word from port DX into ES:DI
6D	INS ES:EDI, DX	Input doubleword from port DX into ES:EDI
6C	INSB	Input byte from port DX into ES:(E)DI
6D	INSW	Input word from port DX into ES:DI
6D	INSD	Input doubleword from port DX into ES:EDI

Description

Copies the data from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand must be the DX register, allowing I/O port addresses from 0 to 65,535 to be accessed. When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size.

The destination operand is a memory location at the address ES:EDI. (When the operand-size attribute is 16, the DI register is used as the destination-index register.) The ES segment cannot be overridden with a segment override prefix.

The INSB, INSW, and INSD mnemonics are synonyms of the byte, word, and doubleword versions of the INS instructions. (For the INS instruction, “ES:EDI” must be explicitly specified in the instruction.)

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the EDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the EDI register is incremented; if the DF flag is 1, the EDI register is decremented.) The EDI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords.

This instruction is only useful for accessing I/O ports located in the processor’s I/O address space.

I/O transactions are performed after all prior data memory operations. No subsequent data memory operations can pass an I/O transaction.

In the IA-64 System Environment, I/O port references are mapped into the 64-bit virtual address pointed to by the IOBase register, with four ports per 4K-byte virtual page. Operating systems can utilize the IA-64 TLBs to grant or deny permission to any four I/O ports. The I/O port space can be mapped into any arbitrary 64-bit physical memory location by operating system code. If CFLG.io is 1 and CPL>IOPL, the TSS is consulted for I/O permission. If CFLG.io is 0 or CPL<=IOPL, permission is granted regardless of the state of the TSS I/O permission bitmap (the bitmap is not referenced).

If the referenced I/O port is mapped to an unimplemented virtual address (via the IOBase register) or if data translations are disabled (PSR.dt is 0) a GPFault is generated on the referencing INS instruction.

INS/INSB/INSW/INSD—Input from Port to String (continued)

Operation

```
IF ((PE = 1) AND ((VM = 1) OR (CPL > IOPL)))
   THEN (* Protected mode or virtual-8086 mode with CPL > IOPL *)
        IF (CFLG.io AND Any I/O Permission Bit for I/O port being accessed = 1)
            THEN #GP(0);
        FI;
   ELSE ( * I/O operation is allowed *)
FI;
IF (IA-64_System_Environment) THEN
   SRC_VA = IOBase | (Port{15:2}<<12) | Port{11:0};
   SRC_PA = translate(SRC_VA);
   DEST ← [SRC_PA]; (* Reads from I/O port *)
FI;

memory_fence();
DEST <- SRC;
memory_fence();
        IF (byte transfer)
            THEN IF DF = 0
                 THEN (E)DI ← 1;
                 ELSE (E)DI ← –1;
            FI;
            ELSE IF (word transfer)
                 THEN IF DF = 0
                      THEN DI ← 2;
                      ELSE DI ← –2;
                 FI;
                 ELSE (* doubleword transfer *)
                      THEN IF DF = 0
                           THEN EDI ← 4;
                           ELSE EDI ← –4;
                      FI;
            FI;
        FI;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Exception	Debug traps for data breakpoints and single step
IA-32_Exception	Alignment faults
#GP(0)	Referenced Port is to an unimplemented virtual address or PSR.dt is zero.

INS/INSB/INSW/INSD—Input from Port to String (continued)

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1 **and when CFLG.io is 1**.
	If the destination is located in a nonwritable segment.
	If an illegal memory operand effective address in the ES segments is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

INT*n*/INTO/INT3—Call to Interrupt Procedure

Opcode	Instruction	Description
CC	INT3	Interrupt 3—trap to debugger
CD *ib*	INT *imm8*	Interrupt vector numbered by immediate byte
CE	INTO	Interrupt 4—if overflow flag is 1

Description

The INT*n* instruction generates a call to the interrupt or exception handler specified with the destination operand. The destination operand specifies an interrupt vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. The first 32 interrupt vectors are reserved by Intel for system use. Some of these interrupts are used for internally generated exceptions.

The INT*n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), interrupt vector 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1.

The INT3 instruction is a special mnemonic for calling the debug exception handler. The action of the INT3 instruction (opcode CC) is slightly different from the operation of the INT 3 instruction (opcode CC03), as follows:

- Interrupt redirection does not happen when in VME mode; the interrupt is handled by a protected-mode handler.
- The virtual-8086 mode IOPL checks do not occur. The interrupt is taken without faulting at any IOPL level.

The action of the INT*n* instruction (including the INTO and INT3 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT*n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

The interrupt vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which points to an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to procedure in the selected segment.

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the “Operation” section for this instruction (except #GP).

INT*n*/INTO/INT3—Call to Interrupt Procedure (continued)

Table 5-14. INT Cases

PE	0	1	1	1	1	1	1	1
VM	–	–	–	–	–	0	1	1
IOPL	–	–	–	–	–	–	<3	=3
DPL/CPL RELATIONSHIP	–	DPL< CPL	–	DPL> CPL	DPL= CPL or C	DPL< CPL & NC	–	–
INTERRUPT TYPE	–	S/W	–	–	–	–	–	–
GATE TYPE	–	–	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

Notes:
– Don't Care
Y Yes, Action Taken
Blank Action Not Taken

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT*n* instruction. If the IOPL is less than 3, the processor generates a general protection exception (#GP); if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to three and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Operation

The following operational description applies not only to the INT*n* and INTO instructions, but also to external interrupts and exceptions.

```
IF IA-64 System EnvironmentTHEN
   IF INT3 Form THEN IA-32_Exception(3);
   IF INTO Form THEN IA-32_Exception(4);
   IF INT Form THEN IA-32_Interrupt(N);
FI;
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (continued)

```
/*IN the IA-64 System Environment all of the following operations are intercepted*/
IF PE=0
  THEN
      GOTO REAL-ADDRESS-MODE;
  ELSE (* PE=1 *)
      GOTO PROTECTED-MODE;
FI;

REAL-ADDRESS-MODE:
  IF ((DEST ∗ 4) + 3) is not within IDT limit THEN #GP; FI;
  IF stack not large enough for a 6-byte return information THEN #SS; FI;
  Push (EFLAGS[15:0]);
  IF ← 0; (* Clear interrupt flag *)
  TF ← 0; (* Clear trap flag *)
  AC ← 0; (*Clear AC flag*)
  Push(CS);
  Push(IP);
  (* No error codes are pushed *)
  CS ← IDT(Descriptor (vector ∗ 4), selector));
  EIP ← IDT(Descriptor (vector ∗ 4), offset)); (* 16 bit offset AND 0000FFFFH *)
END;

PROTECTED-MODE:
  IF ((DEST ∗ 8) + 7) is not within IDT limits
      OR selected IDT descriptor is not an interrupt-, trap-, or task-gate type
          THEN #GP((DEST ∗ 8) + 2 + EXT);
          (* EXT is bit 0 in error code *)
  FI;
  IF software interrupt (* generated by INTn, INT3, or INTO *)
      THEN
          IF gate descriptor DPL < CPL
              THEN #GP((vector number ∗ 8) + 2 );
              (* PE=1, DPL<CPL, software interrupt *)
          FI;
  FI;
  IF gate not present THEN #NP((vector number ∗ 8) + 2 + EXT); FI;
  IF task gate (* specified in the selected interrupt table descriptor *)
      THEN GOTO TASK-GATE;
      ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE=1, trap/interrupt gate *)
  FI;
END;

TASK-GATE: (* PE=1, task gate *)
  Read segment selector in task gate (IDT descriptor);
      IF local/global bit is set to local
          OR index not within GDT limits
              THEN #GP(TSS selector);
      FI;
      Access TSS descriptor in GDT;
      IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
          THEN #GP(TSS selector);
      FI;
      IF TSS not present
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (continued)

```
                THEN #NP(TSS selector);
            FI;
        SWITCH-TASKS (with nesting) to TSS;
        IF interrupt caused by fault with error code
            THEN
                IF stack limit does not allow push of two bytes
                    THEN #SS(0);
                FI;
                Push(error code);
        FI;
        IF EIP not within code segment limit
            THEN #GP(0);
        FI;
    END;
    TRAP-OR-INTERRUPT-GATE
        Read segment selector for trap or interrupt gate (IDT descriptor);
        IF segment selector for code segment is null
            THEN #GP(0H + EXT); (* null selector with EXT flag set *)
        FI;
        IF segment selector is not within its descriptor table limits
            THEN #GP(selector + EXT);
        FI;
        Read trap or interrupt handler descriptor;
        IF descriptor does not indicate a code segment
            OR code segment descriptor DPL > CPL
                THEN #GP(selector + EXT);
        FI;
        IF trap or interrupt gate segment is not present,
            THEN #NP(selector + EXT);
        FI;
        IF code segment is non-conforming AND DPL < CPL
            THEN IF VM=0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE=1, interrupt or trap gate, nonconforming *)
                    (* code segment, DPL<CPL, VM=0 *)
                ELSE (* VM=1 *)
                    IF code segment DPL ≠ 0 THEN #GP(new code segment selector); FI;
                    GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE;
                    (* PE=1, interrupt or trap gate, DPL<CPL, VM=1 *)
            FI;
            ELSE (* PE=1, interrupt or trap gate, DPL ≥ CPL *)
                IF VM=1 THEN #GP(new code segment selector); FI;
                IF code segment is conforming OR code segment DPL = CPL
                    THEN
                        GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                    ELSE
                        #GP(CodeSegmentSelector + EXT);
                        (* PE=1, interrupt or trap gate, nonconforming *)
                        (* code segment, DPL>CPL *)
                FI;
        FI;
    END;
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (continued)

```
INTER-PRIVILEGE-LEVEL-INTERRUPT
  (* PE=1, interrupt or trap gate, non-conforming code segment, DPL<CPL *)
  (* Check segment selector and descriptor for stack of new privilege level in current TSS *)
  IF current TSS is 32-bit TSS
       THEN
            TSSstackAddress ← new code segment (DPL ∗ 8) + 4
            IF (TSSstackAddress + 7) > TSS limit
                 THEN #TS(current TSS selector); FI;
            NewSS ← TSSstackAddress + 4;
            NewESP ← stack address;
       ELSE (* TSS is 16-bit *)
            TSSstackAddress ← new code segment (DPL ∗ 4) + 2
            IF (TSSstackAddress + 4) > TSS limit
                 THEN #TS(current TSS selector); FI;
            NewESP ← TSSstackAddress;
            NewSS ← TSSstackAddress + 2;
  FI;
  IF segment selector is null THEN #TS(EXT); FI;
  IF segment selector index is not within its descriptor table limits
       OR segment selector's RPL ≠ DPL of code segment,
            THEN #TS(SS selector + EXT);
  FI;
Read segment descriptor for stack segment in GDT or LDT;
  IF stack segment DPL ≠ DPL of code segment,
       OR stack segment does not indicate writable data segment,
            THEN #TS(SS selector + EXT);
  FI;
  IF stack segment not present THEN #SS(SS selector+EXT); FI;
  IF 32-bit gate
       THEN
            IF new stack does not have room for 24 bytes (error code pushed)
                 OR 20 bytes (no error code pushed)
                      THEN #SS(segment selector + EXT);
            FI;
       ELSE (* 16-bit gate *)
            IF new stack does not have room for 12 bytes (error code pushed)
                 OR 10 bytes (no error code pushed);
                      THEN #SS(segment selector + EXT);
            FI;
  FI;
  IF instruction pointer is not within code segment limits THEN #GP(0); FI;
  SS:ESP ← TSS(SS:ESP) (* segment descriptor information also loaded *)
  IF 32-bit gate
       THEN
            CS:EIP ← Gate(CS:EIP); (* segment descriptor information also loaded *)
       ELSE (* 16-bit gate *)
            CS:IP ← Gate(CS:IP); (* segment descriptor information also loaded *)
  FI;
  IF 32-bit gate
       THEN
            Push(far pointer to old stack); (* old SS and ESP, 3 words padded to 4 *);
            Push(EFLAGS);
            Push(far pointer to return instruction); (* old CS and EIP, 3 words padded to 4*);
            Push(ErrorCode); (* if needed, 4 bytes *)
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (continued)

```
        ELSE(* 16-bit gate *)
            Push(far pointer to old stack); (* old SS and SP, 2 words *);
            Push(EFLAGS);
            Push(far pointer to return instruction); (* old CS and IP, 2 words *);
            Push(ErrorCode); (* if needed, 2 bytes *)
    FI;
    CPL ← CodeSegmentDescriptor(DPL);
    CS(RPL) ← CPL;
    IF interrupt gate
        THEN IF ← 0 (* interrupt flag to 0 (disabled) *); FI;
    TF ← 0;
    VM ← 0;
    RF ← 0;
    NT ← 0;
I   END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
    (* Check segment selector and descriptor for privilege level 0 stack in current TSS *)
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ← new code segment (DPL * 8) + 4
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ← TSSstackAddress + 4;
            NewESP ← stack address;
        ELSE (* TSS is 16-bit *)
            TSSstackAddress ← new code segment (DPL * 4) + 2
            IF (TSSstackAddress + 4) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewESP ← TSSstackAddress;
            NewSS ← TSSstackAddress + 2;
    FI;
        IF segment selector is null THEN #TS(EXT); FI;
        IF segment selector index is not within its descriptor table limits
            OR segment selector's RPL ≠ DPL of code segment,
                THEN #TS(SS selector + EXT);
        FI;
    Access segment descriptor for stack segment in GDT or LDT;
    IF stack segment DPL ≠ DPL of code segment,
        OR stack segment does not indicate writable data segment,
            THEN #TS(SS selector + EXT);
    FI;
    IF stack segment not present THEN #SS(SS selector+EXT); FI;
    IF 32-bit gate
        THEN
            IF new stack does not have room for 40 bytes (error code pushed)
                OR 36 bytes (no error code pushed);
                    THEN #SS(segment selector + EXT);
            FI;
        ELSE (* 16-bit gate *)
            IF new stack does not have room for 20 bytes (error code pushed)
                OR 18 bytes (no error code pushed);
                    THEN #SS(segment selector + EXT);
            FI;
    FI;
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (continued)

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

    IF CR4.VME = 0
        THEN
            IF IOPL=3
                THEN
                    IF Gate DPL = 3
                        THEN    (*CPL=3, VM=1, IOPL=3, VME=0, gate DPL=3)
                            IF Target CPL != 0
                                THEN #GP(0);
                                ELSE Goto VM86_INTERURPT_TO_PRIV0;
                            FI;
                        ELSE (*Gate DPL < 3*)
                            #GP(0);
                    FI;
                ELSE (*IOPL < 3*)
                    #GP(0);
            FI;
        ELSE (*VME = 1*)
            (*Check whether interrupt is directed for INT n instruction only,
            (*executes virtual 8086 interupt, protected mode interrupt or faults*)
            Ptr <- [TSS + 66];          (*Fetch IO permission bitmpa pointer*)
            IF BIT[Ptr-32,N] = 0        (*software redirection bitmap is 32 bytes below IO
Permission*)
            THEN (*Interrupt redirected*)
                Goto    VM86_INTERRUPT_TO_VM86;
            ELSE
                IF IOPL = 3
                    THEN
                        IF Gate DPL = 3
                            THEN
                                IF Target CPL != 0
                                THEN #GP(0);
                                ELSE Goto VM86_INTERRUPT_TO_PRIV0;
                                FI;
                            ELSE #GP(0);
                        FI;
                    ELSE (*IOPL < 3*)
                        #GP(0);
                FI;
            FI;
    FI;

END;

VM86_INTERRUPT_TO_PRIV0:

    tempEFLAGS ← EFLAGS;
    VM ← 0;
    TF ← 0;
    RF ← 0;
    IF service through interrupt gate THEN IF ← 0; FI;
    TempSS ← SS;
    TempESP ← ESP;
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (continued)

```
    SS:ESP ← TSS(SS0:ESP0); (* Change to level 0 stack segment *)
    (* Following pushes are 16 bits for 16-bit gate and 32 bits for 32-bit gates *)
    (* Segment selector pushes in 32-bit mode are padded to two words *)
    Push(GS);
    Push(FS);
    Push(DS);
    Push(ES);
    Push(TempSS);
    Push(TempESP);
    Push(TempEFlags);
    Push(CS);
    Push(EIP);
    GS ← 0; (*segment registers nullified, invalid in protected mode *)
    FS ← 0;
    DS ← 0;
    ES ← 0;
    CS ← Gate(CS);
    IF OperandSize=32
        THEN
            EIP ← Gate(instruction pointer);
        ELSE (* OperandSize is 16 *)
            EIP ← Gate(instruction pointer) AND 0000FFFFH;
    FI;
    (* Starts execution of new routine in Protected Mode *)
END;

VM86_INTERRUPT_TO_VM86:
    IF IOPL = 3
        THEN
            push(FLAGS OR 3000H);           (*Push FLAGS w/ IOPL bits as 11B or IOPL 3*)
            push(CS);
            push(IP);
            CS <- [N*4 + 2];                (*N is vector num, read from interrupt table*)
            IP <- [N*4];
            FLAGS <- FLAGS AND 7CD5H;       (*Clear TF and IF in EFLAGS like 8086*)
        ELSE
            TempFlags <- FLAGS OR 3000H;    (*Set IOPL to 11B or IOPL 3*)
            TempFlags.IF <- EFLAGS.VIF;
            push(TempFlags);
            push(CS);
            push(IP);
            CS <- [N*4 + 2];                (*N is vector num, read from interrupt table*)
            IP <- [N*4];
            FLAGS <- FLAGS AND 77ED5H;      (*Clear VIF and TF and IF in EFLAGS like 8086*)
    FI;
END;

INTRA-PRIVILEGE-LEVEL-INTERRUPT:
    (* PE=1, DPL = CPL or conforming segment *)
    IF 32-bit gate
        THEN
            IF current stack does not have room for 16 bytes (error code pushed)
                OR 12 bytes (no error code pushed); THEN #SS(0);
            FI;
```

```
        ELSE (* 16-bit gate *)
            IF current stack does not have room for 8 bytes (error code pushed)
                OR 6 bytes (no error code pushed); THEN #SS(0);
            FI;
    IF instruction pointer not within code segment limit THEN #GP(0); FI;
    IF 32-bit gate
        THEN
            Push (EFLAGS);
            Push (far pointer to return instruction); (* 3 words padded to 4 *)
            CS:EIP ← Gate(CS:EIP); (* segment descriptor information also loaded *)
            Push (ErrorCode); (* if any *)
        ELSE (* 16-bit gate *)
            Push (FLAGS);
            Push (far pointer to return location); (* 2 words *)
            CS:IP ← Gate(CS:IP); (* segment descriptor information also loaded *)
            Push (ErrorCode); (* if any *)
    FI;
    CS(RPL) ← CPL;
    IF interrupt gate
        THEN
            IF ← 0; FI;
            TF ← 0;
            NT ← 0;
            VM ← 0;
            RF ← 0;
    FI;
END;
```

Flags Affected

The EFLAGS register is pushed onto stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see "Operation" section.)

Additional IA-64 System Environment Exceptions

IA-32_Exception	If INT3 or INTO form, vector numbers are 3 and 4 respectively.
IA-32_Interrupt	If INT n form, vector number is N.

Protected Mode Exceptions

#GP(0)	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
#GP(selector)	If the segment selector in the interrupt-, trap-, or task gate is null.
	If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the interrupt vector is outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT*n* instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.

INT*n*/INTO/INT3—Call to Interrupt Procedure (continued)

	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the stack segment.
#NP(selector)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(selector)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is null.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the interrupt vector is outside the IDT limits.
#SS	If stack limit violation on push.
	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment when a stack switch occurs.

Virtual 8086 Mode Exceptions

#GP(0)	(For INT*n* instruction) If the IOPL is less than 3 and the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3.
	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
#GP(selector)	If the segment selector in the interrupt-, trap-, or task gate is null.
	If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the interrupt vector is outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.

INT*n*/INTO/INT3—Call to Interrupt Procedure (continued)

	If an interrupt is generated by the INT*n* instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.
#NP(selector)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(selector)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is null.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#BP	If the INT3 instruction is executed.
#OF	If the INTO instruction is executed and the OF flag is set.

INVD—Invalidate Internal Caches

Opcode	Instruction	Description
0F 08	INVD	Flush internal caches; initiate flushing of external caches.

Description

Invalidates (flushes) the processor's internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also implementation-dependent; its function may be implemented differently on future Intel Architecture processors.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,INVD);
Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept Mandatory Instruction Intercept

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0) The INVD instruction cannot be executed at the virtual 8086 mode.

INVD—Invalidate Internal Caches (continued)

Intel Architecture Compatibility

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

INVLPG—Invalidate TLB Entry

Opcode	Instruction	Description
0F 01/7	INVLPG m	Invalidate TLB Entry for page that contains *m*

Description

Invalidates (flushes) the translation lookaside buffer (TLB) entry specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes the TLB entry for that page.

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also implementation-dependent; its function may be implemented differently on future Intel Architecture processors.

The INVLPG instruction normally flushes the TLB entry only for the specified page; however, in some cases, it flushes the entire TLB.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,INVLPG);
Flush(RelevantTLBEntries);
Continue (* Continue execution);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	Operand is a register.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0)	The INVLPG instruction cannot be executed at the virtual 8086 mode.

Intel Architecture Compatibility

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

IRET/IRETD—Interrupt Return

Opcode	Instruction	Description
CF	IRET	Interrupt return (16-bit operand size)
CF	IRETD	Interrupt return (32-bit operand size)

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt or, a software-generated interrupt, or returns from a nested task. IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real Address Mode, the IRET instruction preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Real Mode.
- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.

Return from nested task (task switch)

All forms of IRET result in an IA-32_Intercept(Inst,IRET) in the IA-64 System Environment.

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack). As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a return from a nested task (switches from the called task back to the calling task) or reverses the operation of an interrupt or exception that caused a task switch. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is reentered later, the code that follows the IRET instruction is executed.

IRET performs an instruction serialization and a memory fence operation.

IRET/IRETD—Interrupt Return (continued)

Operation

```
IF(IA-64 System Environment)
        THEN IA-32_Intercept(Inst,IRET);
IF PE = 0
   THEN
        GOTO REAL-ADDRESS-MODE:;
   ELSE
        GOTO PROTECTED-MODE;
FI;

REAL-ADDRESS-MODE;
   IF OperandSize = 32
        THEN
             IF top 12 bytes of stack not within stack limits THEN #SS; FI;
             IF instruction pointer not within code segment limits THEN #GP(0); FI;
             EIP ← Pop();
             CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
             tempEFLAGS ← Pop();
             EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
        ELSE (* OperandSize = 16 *)
             IF top 6 bytes of stack are not within stack limits THEN #SS; FI;
             IF instruction pointer not within code segment limits THEN #GP(0); FI;
             EIP ← Pop();
             EIP ← EIP AND 0000FFFFH;
             CS ← Pop(); (* 16-bit pop *)
             EFLAGS[15:0] ← Pop();
   FI;

END;

PROTECTED-MODE:
   IF VM = 1 (* Virtual-8086 mode: PE=1, VM=1 *)
        THEN
             GOTO RETURN-FROM-VIRTUAL-8086-MODE; (* PE=1, VM=1 *)
   FI;
   IF NT = 1
        THEN
             GOTO TASK-RETURN;( *PE=1, VM=0, NT=1 *)
   FI;

   IF OperandSize=32
        THEN
             IF top 12 bytes of stack not within stack limits
                  THEN #SS(0)
             FI;
             tempEIP ← Pop();
             tempCS ← Pop();
             tempEFLAGS ← Pop();
        ELSE (* OperandSize = 16 *)
             IF top 6 bytes of stack are not within stack limits
                  THEN #SS(0);
```

IRET/IRETD—Interrupt Return (continued)

```
                FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
                tempEIP ← tempEIP AND FFFFH;
                tempEFLAGS ← tempEFLAGS AND FFFFH;
    FI;
    IF tempEFLAGS(VM) = 1 AND CPL=0
        THEN
            GOTO RETURN-TO-VIRTUAL-8086-MODE;
            (* PE=1, VM=1 in EFLAGS image *)
        ELSE
            GOTO PROTECTED-MODE-RETURN;
            (* PE=1, VM=0 in EFLAGS image *)
    FI;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)

    IF CR4.VME = 0
    THEN
        IF IOPL=3 (* Virtual mode: PE=1, VM=1, IOPL=3 *)
            THEN
                IF OperandSize = 32
                THEN
                    IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
                    IF instruction pointer not within code segment limits THEN #GP(0); FI;
                    EIP ← Pop();
                    CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
                    EFLAGS ← Pop();
                    (*VM,IOPL,VIP,and VIF EFLAGS bits are not modified by pop *)
                ELSE (* OperandSize = 16 *)
                    IF top 6 bytes of stack are not within stack limits THEN #SS(0); FI;
                    IF instruction pointer not within code segment limits THEN #GP(0); FI;
                    EIP ← Pop();
                    EIP ← EIP AND 0000FFFFH;
                    CS ← Pop(); (* 16-bit pop *)
                    EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS is not modified by pop *)
                FI;
            ELSE #GP(0); (* trap to virtual-8086 monitor: PE=1, VM=1, IOPL<3 *)
        FI;
    ELSE (*VME is 1*)
        IF IOPL = 3
            THEN
                IF OperandSize = 32
                THEN
                    EIP ← Pop();
                    CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
                    TempEFlags ← Pop();
                    FLAGS = (EFLAGS AND 1B3000H) OR (TempEFlags AND 244FD7H)
                    (*VM,IOPL,RF,VIP,and VIF EFLAGS bits are not modified by pop *)
                ELSE (* OperandSize = 16 *)
                    EIP ← Pop();
                    EIP ← EIP AND 0000FFFFH;
```

IRET/IRETD—Interrupt Return (continued)

```
                        CS ← Pop(); (* 16-bit pop *)
                        TempFlags <- Pop();
                        FLAGS = (FLAGS AND 3000H) OR (TempFLags AND 4FD5H)
                        (*IOPL unmodified*)
                    FI;
            ELSE (*IOPL < 3*)
                IF OperandSize = 16
                    THEN
                        IF ((STACK.TF !-0) OR (EFLAGS.VIP=1 AND STACK.IF=1))
                            THEN #GP(0);
                            ELSE
                                IP <- Pop();        (*Word Pops*)
                                CS <- Pop(0);
                                TempFlags <- Pop();
                                (*FLAGS IOPL, IF and TF are not modified*)
                                FLAGS = (FLAGS AND 3302H) OR (TempFlags AND 4CD5H)
                                EFLAGS.VIF <- TempFlags.IF;
                        FI;
                    ELSE (*OperandSize = 32 *)
                        #GP(0);
                FI;
    FI;

END;

RETURN-TO-VIRTUAL-8086-MODE:

(* Interrupted procedure was in virtual-8086 mode: PE=1, VM=1 in flags image *)
    IF top 24 bytes of stack are not within stack segment limits
        THEN #SS(0);
    FI;
    IF instruction pointer not within code segment limits
        THEN #GP(0);
    FI;
    CS ← tempCS;
    EIP ← tempEIP;
    EFLAGS ← tempEFLAGS
    TempESP ← Pop();
    TempSS ← Pop();
    ES ← Pop(); (* pop 2 words; throw away high-order word *)
    DS ← Pop(); (* pop 2 words; throw away high-order word *)
    FS ← Pop(); (* pop 2 words; throw away high-order word *)
    GS ← Pop(); (* pop 2 words; throw away high-order word *)
    SS:ESP ← TempSS:TempESP;
    (* Resume execution in Virtual 8086 mode *)
END;

TASK-RETURN: (* PE=1, VM=1, NT=1 *)
    Read segment selector in link field of current TSS;
    IF local/global bit is set to local
        OR index not within GDT limits
            THEN #GP(TSS selector);
    FI;
    Access TSS for task specified in link field of current TSS;
```

IRET/IRETD—Interrupt Return (continued)

```
    IF TSS descriptor type is not TSS or if the TSS is marked not busy
        THEN #GP(TSS selector);
    FI;
    IF TSS not present
        THEN #NP(TSS selector);
    FI;
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within code segment limit
        THEN #GP(0);
    FI;
END;

PROTECTED-MODE-RETURN: (* PE=1, VM=0 in flags image *)
    IF return code segment selector is null THEN GP(0); FI;
    IF return code segment selector addrsses descriptor beyond descriptor table limit
        THEN GP(selector; FI;
    Read segment descriptor pointed to by the return code segment selector
    IF return code segment descriptor is not a code segment THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
        AND return code segment DPL > return code segment selector RPL
            THEN #GP(selector); FI;
    IF return code segment descriptor is not present THEN #NP(selector); FI:
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL
    FI;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE=1, VM=0 in flags image, RPL=CPL *)
    IF EIP is not within code segment limits THEN #GP(0); FI;
    EIP ← tempEIP;
    CS ← tempCS; (* segment descriptor information also loaded *)
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize=32
        THEN
            EFLAGS(RF, AC, ID) ← tempEFLAGS;
    FI;
    IF CPL ≤ IOPL
        THEN
            EFLAGS(IF) ← tempEFLAGS;
    FI;
    IF CPL = 0
        THEN
            EFLAGS(IOPL) ← tempEFLAGS;
            IF OperandSize=32
                THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS;
            FI;
    FI;

END;

RETURN-TO-OUTER-PRIVILGE-LEVEL:
```

IRET/IRETD—Interrupt Return (continued)

```
IF OperandSize=32
    THEN
        IF top 8 bytes on stack are not within limits THEN #SS(0); FI;
    ELSE (* OperandSize=16 *)
        IF top 4 bytes on stack are not within limits THEN #SS(0); FI;
FI;
Read return segment selector;
IF stack segment selector is null THEN #GP(0); FI;
IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(SSselector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
    IF stack segment selector RPL ≠ RPL of the return code segment selector
    OR the stack segment descriptor does not indicate a a writable data segment;
    OR stack segment DPL ≠ RPL of the return code segment selector
            THEN #GP(SS selector);
    FI;
    IF stack segment is not present THEN #NP(SS selector); FI;
IF tempEIP is not within code segment limit THEN #GP(0); FI;
EIP ← tempEIP;
CS ← tempCS;
EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
IF OperandSize=32
    THEN
        EFLAGS(RF, AC, ID) ← tempEFLAGS;
FI;
IF CPO ≤ IOPL
    THEN
        EFLAGS(IF) ← tempEFLAGS;
FI;
IF CPL = 0
    THEN
        EFLAGS(IOPL) ← tempEFLAGS;
        IF OperandSize=32
            THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS;
        FI;
FI;
CPL ← RPL of the return code segment selector;
FOR each of segment register (ES, FS, GS, and DS)
    DO;
        IF segment register points to data or non-conforming code segment
        AND CPL > segment descriptor DPL (* stored in hidden part of segment register *)
            THEN (* segment register invalid *)
                SegmentSelector ← 0; (* null segment selector *)
        FI;
    OD;
END:
```

Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor.

IRET/IRETD—Interrupt Return (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Intercept	Instruction Intercept Trap for ALL forms of IRET.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is null.
	If the return instruction pointer is not within the return code segment limit.
#GP(selector)	If a segment selector index is outside its descriptor table limits.
	If the return code segment selector RPL is greater than the CPL.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If the top bytes of stack are not within stack limits.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.

Real Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

IRET/IRETD—Interrupt Return (continued)

Virtual 8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit.
	IF IOPL not equal to 3
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.

Jcc—Jump if Condition Is Met

Opcode	Instruction	Description
77 *cb*	JA *rel8*	Jump short if above (CF=0 and ZF=0)
73 *cb*	JAE *rel8*	Jump short if above or equal (CF=0)
72 *cb*	JB *rel8*	Jump short if below (CF=1)
76 *cb*	JBE *rel8*	Jump short if below or equal (CF=1 or ZF=1)
72 *cb*	JC *rel8*	Jump short if carry (CF=1)
E3 *cb*	JCXZ *rel8*	Jump short if CX register is 0
E3 *cb*	JECXZ *rel8*	Jump short if ECX register is 0
74 *cb*	JE *rel8*	Jump short if equal (ZF=1)
7F *cb*	JG *rel8*	Jump short if greater (ZF=0 and SF=OF)
7D *cb*	JGE *rel8*	Jump short if greater or equal (SF=OF)
7C *cb*	JL *rel8*	Jump short if less (SF<>OF)
7E *cb*	JLE *rel8*	Jump short if less or equal (ZF=1 or SF<>OF)
76 *cb*	JNA *rel8*	Jump short if not above (CF=1 or ZF=1)
72 *cb*	JNAE *rel8*	Jump short if not above or equal (CF=1)
73 *cb*	JNB *rel8*	Jump short if not below (CF=0)
77 *cb*	JNBE *rel8*	Jump short if not below or equal (CF=0 and ZF=0)
73 *cb*	JNC *rel8*	Jump short if not carry (CF=0)
75 *cb*	JNE *rel8*	Jump short if not equal (ZF=0)
7E *cb*	JNG *rel8*	Jump short if not greater (ZF=1 or SF<>OF)
7C *cb*	JNGE *rel8*	Jump short if not greater or equal (SF<>OF)
7D *cb*	JNL *rel8*	Jump short if not less (SF=OF)
7F *cb*	JNLE *rel8*	Jump short if not less or equal (ZF=0 and SF=OF)
71 *cb*	JNO *rel8*	Jump short if not overflow (OF=0)
7B *cb*	JNP *rel8*	Jump short if not parity (PF=0)
79 *cb*	JNS *rel8*	Jump short if not sign (SF=0)
75 *cb*	JNZ *rel8*	Jump short if not zero (ZF=0)
70 *cb*	JO *rel8*	Jump short if overflow (OF=1)
7A *cb*	JP *rel8*	Jump short if parity (PF=1)
7A *cb*	JPE *rel8*	Jump short if parity even (PF=1)
7B *cb*	JPO *rel8*	Jump short if parity odd (PF=0)
78 *cb*	JS *rel8*	Jump short if sign (SF=1)
74 *cb*	JZ *rel8*	Jump short if zero (ZF = 1)
0F 87 *cw/cd*	JA *rel16/32*	Jump near if above (CF=0 and ZF=0)
0F 83 *cw/cd*	JAE *rel16/32*	Jump near if above or equal (CF=0)
0F 82 *cw/cd*	JB *rel16/32*	Jump near if below (CF=1)
0F 86 *cw/cd*	JBE *rel16/32*	Jump near if below or equal (CF=1 or ZF=1)
0F 82 *cw/cd*	JC *rel16/32*	Jump near if carry (CF=1)
0F 84 *cw/cd*	JE *rel16/32*	Jump near if equal (ZF=1)
0F 84 *cw/cd*	JZ *rel16/32*	Jump near if 0 (ZF=1)
0F 8F *cw/cd*	JG *rel16/32*	Jump near if greater (ZF=0 and SF=OF)

Jcc—Jump if Condition Is Met (continued)

Opcode	Instruction	Description
0F 8D *cw/cd*	JGE *rel16/32*	Jump near if greater or equal (SF=OF)
0F 8C *cw/cd*	JL *rel16/32*	Jump near if less (SF<>OF)
0F 8E *cw/cd*	JLE *rel16/32*	Jump near if less or equal (ZF=1 or SF<>OF)
0F 86 *cw/cd*	JNA *rel16/32*	Jump near if not above (CF=1 or ZF=1)
0F 82 *cw/cd*	JNAE *rel16/32*	Jump near if not above or equal (CF=1)
0F 83 *cw/cd*	JNB *rel16/32*	Jump near if not below (CF=0)
0F 87 *cw/cd*	JNBE *rel16/32*	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 *cw/cd*	JNC *rel16/32*	Jump near if not carry (CF=0)
0F 85 *cw/cd*	JNE *rel16/32*	Jump near if not equal (ZF=0)
0F 8E *cw/cd*	JNG *rel16/32*	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C *cw/cd*	JNGE *rel16/32*	Jump near if not greater or equal (SF<>OF)
0F 8D *cw/cd*	JNL *rel16/32*	Jump near if not less (SF=OF)
0F 8F *cw/cd*	JNLE *rel16/32*	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 *cw/cd*	JNO *rel16/32*	Jump near if not overflow (OF=0)
0F 8B *cw/cd*	JNP *rel16/32*	Jump near if not parity (PF=0)
0F 89 *cw/cd*	JNS *rel16/32*	Jump near if not sign (SF=0)
0F 85 *cw/cd*	JNZ *rel16/32*	Jump near if not zero (ZF=0)
0F 80 *cw/cd*	JO *rel16/32*	Jump near if overflow (OF=1)
0F 8A *cw/cd*	JP *rel16/32*	Jump near if parity (PF=1)
0F 8A *cw/cd*	JPE *rel16/32*	Jump near if parity even (PF=1)
0F 8B *cw/cd*	JPO *rel16/32*	Jump near if parity odd (PF=0)
0F 88 *cw/cd*	JS *rel16/32*	Jump near if sign (SF=1)
0F 84 *cw/cd*	JZ *rel16/32*	Jump near if 0 (ZF=1)

Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the J*cc* instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (*rel8*, *rel16,* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of –128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each J*cc* mnemonic are given in the “Description” column of the above table. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Jcc—Jump if Condition Is Met (continued)

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The J*cc* instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the J*cc* instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The JECXZ and JCXZ instructions differs from the other J*cc* instructions because they do not check the status flags. Instead they check the contents of the ECX and CX registers, respectively, for 0. These instructions are useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as LOOPNE). They prevent entering the loop when the ECX or CX register is equal to 0, which would cause the loop to execute 2^{32} or 64K times, respectively, instead of zero times.

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

Operation

```
IF condition
  THEN
       EIP ← EIP + SignExtend(DEST);
       IF OperandSize = 16
            THEN
                 EIP ← EIP AND 0000FFFFH;
       FI;
  IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Exception	Taken Branch Debug Exception if PSR.tb is 1

Protected Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment.

Jcc—Jump if Condition Is Met (continued)

Real Address Mode Exceptions

#GP	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if 32-address size override prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if 32-address size override prefix is used.

JMP—Jump

Opcode	Instruction	Description
EB *cb*	JMP *rel8*	Jump near, relative address
E9 *cw*	JMP *rel16*	Jump near, relative address
E9 *cd*	JMP *rel32*	Jump near, relative address
FF /4	JMP *r/m16*	Jump near, indirect address
FF /4	JMP *r/m32*	Jump near, indirect address
EA *cd*	JMP *ptr16:16*	Jump far, absolute address
EA *cp*	JMP *ptr16:32*	Jump far, absolute address
FF /5	JMP *m16:16*	Jump far, indirect address
FF /5	JMP *m16:32*	Jump far, indirect address

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

- Near jump – A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far jump – A jump to an instruction located in a different segment than the current code segment, sometimes referred to as an intersegment call.
- Task switch – A jump to an instruction located in a different task. (This is a form of a far jump.) **Results in an IA-32_Intercept(Gate) in IA-64 System Environment.**

A task switch can only be executed in protected mode (see Chapter 6 in the *Intel Architecture Software Developer's Manual, Volume 3* for information on task switching with the JMP instruction).

When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute address (that is an offset from the base of the code segment) or a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). An absolute address is specified directly in a register or indirectly in a memory location (*r/m16* or *r/m32* operand form). A relative offset (*rel8*, *rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the value in the EIP register (that is, to the instruction following the JMP instruction). The operand-size attribute determines the size of the target operand (16 or 32 bits) for absolute addresses. Absolute addresses are loaded directly into the EIP register. When a relative offset is specified, it is added to the value of the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits. The CS register is not changed on near jumps.

JMP—Jump (continued)

When executing a far jump, the processor jumps to the code segment and address specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s.

When the processor is operating in protected mode, a far jump can also be used to access a code segment through a call gate or to switch tasks. Here, the processor uses the segment selector part of the far address to access the segment descriptor for the segment being jumped to. Depending on the value of the type and access rights information in the segment selector, the JMP instruction can perform:

- A far jump to a conforming or non-conforming code segment (same mechanism as the far jump described in the previous paragraph, except that the processor checks the access rights of the code segment being jumped to).
- An far jump through a call gate.
- A task switch. **Results in an IA-32_Intercept(Gate) in IA-64 System Environment.**

The JMP instruction cannot be used to perform inter-privilege level jumps.

When executing an far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate and instruction either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction, is similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to. (The offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code, data, and stack segments and the instruction pointer to the target instruction. One form of the JMP instruction allows the jump to be made directly to a TSS, without going through a task gate. See Chapter 13 in *Intel Architecture Software Developer's Manual, Volume 3* the for detailed information on the mechanics of a task switch.

All branches are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

JMP—Jump (continued)

Operation

```
IF near jump
  THEN IF near relative jump
       THEN
            tempEIP ← EIP + DEST; (* EIP is instruction following JMP instruction*)
       ELSE (* near absolute jump *)
            tempEIP ← DEST;
  FI;
  IF tempEIP is beyond code segment limit THEN #GP(0); FI;
  IF OperandSize = 32
       THEN
            EIP ← tempEIP;
       ELSE (* OperandSize=16 *)
            EIP ← tempEIP AND 0000FFFFH;
  FI;
  IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
FI:

IF far jump AND (PE = 0 OR (PE = 1 AND VM = 1)) (* real address or virtual 8086 mode *)
  THEN
       tempEIP ← DEST(offset); (* DEST is ptr16:32 or [m16:32] *)
       IF tempEIP is beyond code segment limit THEN #GP(0); FI;
       CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
       IF OperandSize = 32
            THEN
                 EIP ← tempEIP; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                 EIP ← tempEIP AND 0000FFFFH; (* clear upper 16 bits *)
       FI;
       IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
FI;
IF far call AND (PE = 1 AND VM = 0) (* Protected mode, not virtual 8086 mode *)
  THEN
       IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
            OR segment selector in target operand null
            THEN #GP(0);
       FI;
       IF segment selector index not within descriptor table limits
            THEN #GP(new selector);
       FI;
       Read type and access rights of segment descriptor;
       IF segment type is not a conforming or nonconforming code segment, call gate,
            task gate, or TSS THEN #GP(segment selector); FI;
       Depending on type and access rights
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
  ELSE
       #GP(segment selector);
FI;
```

JMP—Jump (continued)

```
CONFORMING-CODE-SEGMENT:
   IF DPL > CPL THEN #GP(segment selector); FI;
   IF segment not present THEN #NP(segment selector); FI;
   tempEIP ← DEST(offset);
   IF OperandSize=16
        THEN tempEIP ← tempEIP AND 0000FFFFH;
   FI;
   IF tempEIP not in code segment limit THEN #GP(0); FI;
   CS ← DEST(SegmentSelector); (* segment descriptor information also loaded *)
   CS(RPL) ← CPL
   EIP ← tempEIP;
   IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
END;

NONCONFORMING-CODE-SEGMENT:
   IF (RPL > CPL) OR (DPL ≠ CPL) THEN #GP(code segment selector); FI;
   IF segment not present THEN #NP(segment selector); FI;
   IF instruction pointer outside code segment limit THEN #GP(0); FI;
   tempEIP ← DEST(offset);
   IF OperandSize=16
        THEN tempEIP ← tempEIP AND 0000FFFFH;
   FI;
   IF tempEIP not in code segment limit THEN #GP(0); FI;
   CS ← DEST(SegmentSelector); (* segment descriptor information also loaded *)
   CS(RPL) ← CPL
   EIP ← tempEIP;
   IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
END;

CALL-GATE:

   IF call gate DPL < CPL
        OR call gate DPL < call gate segment-selector RPL
             THEN #GP(call gate selector); FI;
   IF call gate not present THEN #NP(call gate selector); FI;
   IF IA-64 System Environment THEN IA-32_Intercept(Gate,JMP);
   IF call gate code-segment selector is null THEN #GP(0); FI;
   IF call gate code-segment selector index is outside descriptor table limits
        THEN #GP(code segment selector); FI;
   Read code segment descriptor;
   IF code-segment segment descriptor does not indicate a code segment
        OR code-segment segment descriptor is conforming and DPL > CPL
        OR code-segment segment descriptor is non-conforming and DPL ≠ CPL
             THEN #GP(code segment selector); FI;
   IF code segment is not present THEN #NP(code-segment selector); FI;
   IF instruction pointer is not within code-segment limit THEN #GP(0); FI;
   tempEIP ← DEST(offset);
   IF GateSize=16
        THEN tempEIP ← tempEIP AND 0000FFFFH;
   FI;
   IF tempEIP not in code segment limit THEN #GP(0); FI;
   CS ← DEST(SegmentSelector); (* segment descriptor information also loaded *)
   CS(RPL) ← CPL
   EIP ← tempEIP;
```

```
END;

TASK-GATE:
  IF task gate DPL < CPL
       OR task gate DPL < task gate segment-selector RPL
            THEN #GP(task gate selector); FI;
  IF task gate not present THEN #NP(gate selector); FI;
  IF IA-64 System Environment THEN IA-32_Intercept(Gate,JMP);
  Read the TSS segment selector in the task-gate descriptor;
  IF TSS segment selector local/global bit is set to local
       OR index not within GDT limits
       OR TSS descriptor specifies that the TSS is busy
            THEN #GP(TSS selector); FI;
  IF TSS not present THEN #NP(TSS selector); FI;
  SWITCH-TASKS to TSS;
  IF EIP not within code segment limit THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
  IF TSS DPL < CPL
       OR TSS DPL < TSS segment-selector RPL
       OR TSS descriptor indicates TSS not available
            THEN #GP(TSS selector); FI;
  IF TSS is not present THEN #NP(TSS selector); FI;
  IF IA-64 System Environment THENIA-32_Intercept(Gate,JMP);
  SWITCH-TASKS to TSS
  IF EIP not within code segment limit THEN #GP(0); FI;
END;
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Intercept	Gate Intercept for JMP through CALL Gates, Task Gates and Task Segments
IA-32_Exception	Taken Branch Debug Exception if PSR.tb is 1

Protected Mode Exceptions

#GP(0)	If offset in target operand, call gate, or TSS is beyond the code segment limits. If the segment selector in the destination operand, call gate, task gate, or TSS is null. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

JMP—Jump (continued)

	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
	If the DPL for a nonconforming-code segment is not equal to the CPL
	(When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.
	If the segment descriptor for selector in a call gate does not indicate it is a code segment.
	If the segment descriptor for the segment selector in a task gate does not indicate available TSS.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP (selector)	If the code segment being accessed is not present.
	If call gate, task gate, or TSS not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

JMPE—Jump to IA-64 Instruction Set

Opcode	Instruction	Description
0F 00 /6	JMPE *r/m16*	Jump to IA-64, indirect address specified by r/m16
0F 00 /6	JMPE *r/m32*	Jump to IA-64, indirect address specified by r/m32
0F B8	JMPE *disp16*	Jump to IA-64, absolute address specified by addr16
0F B8	JMPE *disp32*	Jump to IA-64, absolute address specified by addr32

Description

This instruction is available only on IA-64 processors in the IA-64 System Environment. Otherwise, execution of this instruction at privilege levels 1, 2, and 3 results in an Illegal Opcode fault, and at privilege level 0, termination of the IA-32 System Environment on an IA-64 processor.

JMPE switches the processor to the IA-64 instruction set and starts execution at the specified target address There are two forms; an indirect form, r/m*r16/32*, and an unsigned absolute form, *disp16/32*. Both 16 and 32-bit formats are supported.

The absolute form computes the 16-byte aligned 64-bit virtual target address in the IA-64 instruction set by adding the unsigned 16 or 32-bit displacement to the current CS base (*IP{31:0} = disp16/32 + CSD.base)*. The indirect form specifies the virtual IA-64 target address by the contents of a register or memory location (*IP{31:0} = [r/m16/32] + CSD.base)*. IA-64 targets are constrained to the lower 4G-bytes of the 64-bit virtual address space within virtual region 0.

GR[1] is loaded with the next sequential instruction address following JMPE.

If PSR.di is 1, the instruction is nullified and a Disabled Instruction Set Transition fault is generated. If IA-64 branch debugging is enabled, an IA-64 branch debug trap is taken after JMPE completes execution.

JMPE can be performed at any privilege level and does not change the privilege level of the processor.

JMPE performs a FWAIT operation, any pending IA-32 unmasked floating-point exceptions are reported as faults on the JMPE instruction.

JMPE does not perform a memory fence or serialization operation.

Successful execution of JMPE clears EFLAG.rf and PSR.id to zero.

If the IA-64 register stack engine is enabled for eager execution, the register stack engine may immediately start loading registers when the processor enters the IA-64 instruction set.

JMPE—Jump to IA-64 Instruction Set (continued)

Operation

```
if (NOT IA-64 System Environment) {
        if (PSR.cpl==0) Terminate_IA-32_System_Env();
        else IA_32_Exception(IllegalOpcode);
} else if (PSR.di==1) {
        Disabled_Instruction_Set_Transition_Fault();
} else if (pending_numeric_exceptions()) {
        IA_32_exception(FPError);
} else {
        if (absolute_form) {                        //compute virtual target
            IP{31:0} = disp16/32 + AR[CSD].base;//disp is 16/32-bit unsigned value
        } else if (indirect_form) {
            IP{31:0} = [r/m16/32] + AR[CSD].base;
        }
        PSR.is = 0;                                 //set IA-64 Instruction Set Mode
        IP{3:0}= 0;                                 //Force 16-byte alignment
        IP{63:32} = 0;                              //zero extend from 32-bits to 64-bits
        GR[1]{31:0} = EIP + AR[CSD].base;           //next sequential instruction address
        GR[1]{63:32} = 0;
        PSR.id = EFLAG.rf = 0;

        if (PSR.tb)                                 //taken branch trap
            IA_32_Exception(Debug);
}
```

Flags Affected

None (other than EFLAG.rf)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Fault.
Disabled ISA	Disabled Instruction Set Transition Fault, if PSR.di is 1
IA-32_Exception	Floating-point Error, if any floating-point exceptions are pending
IA-32_Exception	Taken Branch trap, if PSR.tb is 1.

IA-32 System Environment Exceptions (All Operating Modes)

#UD	JMPE raises an invalid opcode exception at privilege levels 1, 2 and 3. Privilege level 0 results in termination of the IA-32 System Environment on an IA-64 processor.

LAHF—Load Status Flags into AH Register

Opcode	Instruction	Description
9F	LAHF	Load: AH = EFLAGS(SF:ZF:0:AF:0:PF:1:CF)

Description

Moves the low byte of the EFLAGS register (which includes status flags SF, ZF, AF, PF, and CF) to the AH register. Reserved bits 1, 3, and 5 of the EFLAGS register are set in the AH register as shown in the “Operation” below.

Operation

```
AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
```

Flags Affected

None (that is, the state of the flags in the EFLAGS register are not affected).

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

LAR—Load Access Rights Byte

Opcode	Instruction	Description
0F 02 /*r*	LAR *r16,r/m16*	*r16* ← *r/m16* masked by FF00H
0F 02 /*r*	LAR *r32,r/m32*	*r32* ← *r/m32* masked by 00FxFF00H

Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can preform additional checks on the access rights information.

When the operand size is 32 bits, the access rights for a segment descriptor comprise the type and DPL fields and the S, P, AVL, D/B, and G flags, all of which are located in the second doubleword (bytes 4 through 7) of the segment descriptor. The doubleword is masked by 00FXFF00H before it is loaded into the destination operand. When the operand size is 16 bits, the access rights comprise the type and DPL fields. Here, the two lower-order bytes of the doubleword are masked by FF00H before being loaded into the destination operand.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not null.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed.
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode.

LAR—Load Access Rights Byte (continued)

Table 5-15. LAR Descriptor Validity

Type	Name	Valid
0	Reserved	No
1	Available 16-bit TSS	Yes
2	LDT	Yes
3	Busy 16-bit TSS	Yes
4	16-bit call gate	Yes
5	16-bit/32-bit task gate	Yes
6	16-bit trap gate	No
7	16-bit interrupt gate	No
8	Reserved	No
9	Available 32-bit TSS	Yes
A	Reserved	No
B	Busy 32-bit TSS	Yes
C	32-bit call gate	Yes
D	Reserved	No
E	32-bit trap gate	No
F	32-bit interrupt gate	No

Operation

```
IF SRC(Offset) > descriptor table limit THEN ZF ← 0; FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ conforming code segment
  AND (CPL > DPL) OR (RPL > DPL)
  OR Segment type is not valid for instruction
      THEN
          ZF ← 0
      ELSE
          IF OperandSize = 32
              THEN
                  DEST ← [SRC] AND 00FxFF00H;
              ELSE (*OperandSize = 16*)
                  DEST ← [SRC] AND FF00H;
          FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

LAR—Load Access Rights Byte (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

#UD	The LAR instruction cannot be executed in virtual 8086 mode.

LDS/LES/LFS/LGS/LSS—Load Far Pointer

Opcode	Instruction	Description
C5 /*r*	LDS *r16,m16:16*	Load DS:*r16* with far pointer from memory
C5 /*r*	LDS *r32,m16:32*	Load DS:*r32* with far pointer from memory
0F B2 /*r*	LSS *r16,m16:16*	Load SS:*r16* with far pointer from memory
0F B2 /*r*	LSS *r32,m16:32*	Load SS:*r32* with far pointer from memory
C4 /*r*	LES *r16,m16:16*	Load ES:*r16* with far pointer from memory
C4 /*r*	LES *r32,m16:32*	Load ES:*r32* with far pointer from memory
0F B4 /*r*	LFS *r16,m16:16*	Load FS:*r16* with far pointer from memory
0F B4 /*r*	LFS *r32,m16:32*	Load FS:*r32* with far pointer from memory
0F B5 /*r*	LGS *r16,m16:16*	Load GS:*r16* with far pointer from memory
0F B5 /*r*	LGS *r32,m16:32*	Load GS:*r32* with far pointer from memory

Description

Load a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register implied with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a null selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a null selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

Operation

```
IF ProtectedMode
  THEN IF SS is loaded
      THEN IF SegementSelector = null
          THEN #GP(0);
      FI;
      ELSE IF Segment selector index is not within descriptor table limits
      OR Segment selector RPL ≠ CPL
      OR Access rights indicate nonwritable data segment
      OR DPL ≠ CPL
          THEN #GP(selector);
      FI;
      ELSE IF Segment marked not present
          THEN #SS(selector);
      FI;
      SS ← SegmentSelector(SRC);
      SS ← SegmentDescriptor([SRC]);
```

LDS/LES/LFS/LGS/LSS—Load Far Pointer (continued)

```
    ELSE IF DS, ES, FS, or GS is loaded with non-null segment selector
        THEN IF Segment selector index is not within descriptor table limits
        OR Access rights indicate segment neither data nor readable code segment
        OR (Segment is data or nonconforming-code segment
            AND both RPL and CPL > DPL)
            THEN #GP(selector);
        FI;
        ELSE IF Segment marked not present
            THEN #NP(selector);
        FI;
        SegmentRegister ← SegmentSelector(SRC) AND RPL;
        SegmentRegister ← SegmentDescriptor([SRC]);
    ELSE IF DS, ES, FS or GS is loaded with a null selector:
        SegmentRegister ← NullSelector;
        SegmentRegister(DescriptorValidBit) ← 0; (*hidden flag; not accessible by software*)
    FI;
FI;
IF (Real-Address or Virtual 8086 Mode)
    THEN
        SS ← SegmentSelector(SRC);
FI;
DEST ← Offset(SRC);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#UD	If source operand is not a memory location.
#GP(0)	If a null selector is loaded into the SS register.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#GP(selector)	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL.

LDS/LES/LFS/LGS/LSS—Load Far Pointer (continued)

	If the DS, ES, FS, or GS register is being loaded with a non-null segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If DS, ES, FS, or GS register is being loaded with a non-null segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

LEA—Load Effective Address

Opcode	Instruction	Description
8D /*r*	LEA *r16,m*	Store effective address for *m* in register *r16*
8D /*r*	LEA *r32,m*	Store effective address for *m* in register *r32*

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

Table 5-16. LEA Address and Operand Sizes

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

Operation

```
IF OperandSize = 16 AND AddressSize = 16
  THEN
      DEST ← EffectiveAddress(SRC); (* 16-bit address *)
  ELSE IF OperandSize = 16 AND AddressSize = 32
      THEN
          temp ← EffectiveAddress(SRC); (* 32-bit address *)
          DEST ← temp[0..15]; (* 16-bit address *)
  ELSE IF OperandSize = 32 AND AddressSize = 16
      THEN
          temp ← EffectiveAddress(SRC); (* 16-bit address *)
          DEST ← ZeroExtend(temp); (* 32-bit address *)
  ELSE IF OperandSize = 32 AND AddressSize = 32
      THEN
          DEST ← EffectiveAddress(SRC); (* 32-bit address *)
  FI;
FI;
```

LEA—Load Effective Address (continued)

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Protected Mode Exceptions

#UD If source operand is not a memory location.

Real Address Mode Exceptions

#UD If source operand is not a memory location.

Virtual 8086 Mode Exceptions

#UD If source operand is not a memory location.

LEAVE—High Level Procedure Exit

Opcode	Instruction	Description
C9	LEAVE	Set SP to BP, then pop BP
C9	LEAVE	Set ESP to EBP, then pop EBP

Description

Executes a return from a procedure or group of nested procedures established by an earlier ENTER instruction. The instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), releasing the stack space used by a procedure for its local variables. The old frame pointer (the frame pointer for the calling procedure that issued the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure and remove any arguments pushed onto the stack by the procedure being returned from.

Operation

```
IF StackAddressSize = 32
  THEN
       ESP ← EBP;
  ELSE (* StackAddressSize = 16*)
       SP ← BP;
FI;
IF OperandSize = 32
  THEN
       EBP ← Pop();
  ELSE (* OperandSize = 16*)
       BP ← Pop();
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#SS(0)	If the EBP register points to a location that is not within the limits of the current stack segment.

LEAVE—High Level Procedure Exit (continued)

Real Address Mode Exceptions

#GP	If the EBP register points to a location outside of the effective address space from 0 to 0FFFFH.

Virtual 8086 Mode Exceptions

#GP(0)	If the EBP register points to a location outside of the effective address space from 0 to 0FFFFH.

LES—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

LFS—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Description
0F 01 /2	LGDT *m16&32*	Load *m* into GDTR
0F 01 /3	LIDT *m16&32*	Load *m* into IDTR

Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand is a pointer to 6 bytes of data in memory that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,LGDT/LIDT);

IF instruction is LIDT
  THEN
        IF OperandSize = 16
        THEN
            IDTR(Limit) ← SRC[0:15];
            IDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
        ELSE (* 32-bit Operand Size *)
            IDTR(Limit) ← SRC[0:15];
            IDTR(Base) ← SRC[16:47];
    FI;
  ELSE (* instruction is LGDT *)
    IF OperandSize = 16
        THEN
            GDTR(Limit) ← SRC[0:15];
            GDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
        ELSE (* 32-bit Operand Size *)
            GDTR(Limit) ← SRC[0:15];
            GDTR(Base) ← SRC[16:47];
    FI;
FI;
```

Flags Affected

None.

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register (continued)

Additional IA-64 System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept for LIDT and LGDT

Protected Mode Exceptions

#UD	If source operand is not a memory location.
#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

#UD	If source operand is not a memory location.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

LGS—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Description
0F 00 /2	LLDT *r/m16*	Load segment selector *r/m16* into LDTR

Description

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses to segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If the source operand is 0, the LDTR is marked invalid and all references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. Also, this instruction can only be executed in protected mode.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,LLDT);

IF SRC(Offset) > descriptor table limit THEN #GP(segment selector); FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ LDT THEN #GP(segment selector); FI;
IF segment descriptor is not present THEN #NP(segment selector);
LDTR(SegmentSelector) ← SRC;
LDTR(SegmentDescriptor) ← GDTSegmentDescriptor;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	Instruction Intercept

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.

LLDT—Load Local Descriptor Table Register (continued)

#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.
	Segment selector is beyond GDT limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

#UD	The LLDT instruction is recognized in virtual 8086 mode.

LIDT—Load Interrupt Descriptor Table Register

See entry for LGDT/LIDT—Load Global Descriptor Table Register/Load Interrupt Descriptor Table Register.

LMSW—Load Machine Status Word

Opcode	Instruction	Description
0F 01 /6	LMSW *r/m16*	Loads *r/m16* in machine status word of CR0

Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. The PE flag in the CR0 register is a sticky bit. Once set to 1, the LMSW instruction cannot be used clear this flag and force a switch back to real address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual 8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on processors more recent than the Intel 286 should use the MOV (control registers) instruction to load the machine status word.

This instruction is a serializing instruction.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,LMSW);
CR0[0:3] ← SRC[0:3];
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

LMSW—Load Machine Status Word (continued)

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Description
F0	LOCK	Asserts LOCK# signal for duration of the accompanying instruction

Description

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal insures that the processor has exclusive use of any shared memory while the signal is asserted.

The LOCK prefix can be prepended only to the following instructions and to those forms of the instructions that use a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. An undefined opcode exception will be generated if the LOCK prefix is used with any other instruction. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

Operation

```
IF IA-64 System Environment AND External_Bus_Lock_Required AND DCR.lc
   THEN IA-32_Intercept(LOCK);
AssertLOCK#(DurationOfAccompaningInstruction)
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Intercept	Lock Intercept - If an external atomic bus lock is required to complete this operation and DCR.lc is 1, no atomic transaction occurs, the instruction is faulted and an IA-32_Intercept(Lock) fault is generated. The software lock handler is responsible for the emulation of the instruction.

Protected Mode Exceptions

#UD	If the LOCK prefix is used with an instruction not listed in the "Description" section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

LOCK—Assert LOCK# Signal Prefix (continued)

Real Address Mode Exceptions

#UD	If the LOCK prefix is used with an instruction not listed in the “Description” section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

Virtual 8086 Mode Exceptions

#UD	If the LOCK prefix is used with an instruction not listed in the “Description” section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

LODS/LODSB/LODSW/LODSD—Load String Operand

Opcode	Instruction	Description
AC	LODS DS:(E)SI	Load byte at address DS:(E)SI into AL
AD	LODS DS:SI	Load word at address DS:SI into AX
AD	LODS DS:ESI	Load doubleword at address DS:ESI into EAX
AC	LODSB	Load byte at address DS:(E)SI into AL
AD	LODSW	Load word at address DS:SI into AX
AD	LODSD	Load doubleword at address DS:ESI into EAX

Description

Load a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location at the address DS:ESI. (When the operand-size attribute is 16, the SI register is used as the source-index register.) The DS segment may be overridden with a segment override prefix.

The LODSB, LODSW, and LODSD mnemonics are synonyms of the byte, word, and doubleword versions of the LODS instructions. (For the LODS instruction, “DS:ESI” must be explicitly specified in the instruction.)

After the byte, word, or doubleword is transfer from the memory location into the AL, AX, or EAX register, the ESI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the ESI register is incremented; if the DF flag is 1, the ESI register is decremented.) The ESI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct, because further processing of the data moved into the register is usually necessary before the next transfer can be made. See “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” on page 5-326 for a description of the REP prefix.

Operation

```
IF (byte load)
  THEN
      AL ← SRC; (* byte load *)
          THEN IF DF = 0
              THEN (E)SI ← 1;
              ELSE (E)SI ← –1;
          FI;
  ELSE IF (word load)
      THEN
          AX ← SRC; (* word load *)
              THEN IF DF = 0
                  THEN SI ← 2;
                  ELSE SI ← –2;
              FI;
      ELSE (* doubleword transfer *)
          EAX ← SRC; (* doubleword load *)
              THEN IF DF = 0
```

LODS/LODSB/LODSW/LODSD—Load String Operand (continued)

```
                    THEN ESI ← 4;
                    ELSE ESI ← –4;

                FI;
    FI;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

LOOP/LOOP*cc*—Loop According to ECX Counter

Opcode	Instruction	Description
E2 *cb*	LOOP *rel8*	Decrement count; jump short if count ≠ 0
E1 *cb*	LOOPE *rel8*	Decrement count; jump short if count ≠ 0 and ZF=1
E1 *cb*	LOOPZ *rel8*	Decrement count; jump short if count ≠ 0 and ZF=1
E0 *cb*	LOOPNE *rel8*	Decrement count; jump short if count ≠ 0 and ZF=0
E0 *cb*	LOOPNZ *rel8*	Decrement count; jump short if count ≠ 0 and ZF=0

Description

Performs a loop operation using the ECX or CX register as a counter. Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop. If the address-size attribute is 32 bits, the ECX register is used as the count register; otherwise the CX register is used.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of –128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP*cc*) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the LOOP*cc* instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

All branches are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

Operation

```
IF AddressSize = 32
  THEN
       Count is ECX;
  ELSE (* AddressSize = 16 *)
       Count is CX;
FI;
Count ← Count – 1;

IF instruction in not LOOP
  THEN
       IF (instruction = LOOPE) OR (instruction = LOOPZ)
            THEN
                 IF (ZF =1) AND (Count ≠ 0)
                      THEN BranchCond ← 1;
                      ELSE BranchCond ← 0;
                 FI;
       FI;
       IF (instruction = LOOPNE) OR (instruction = LOOPNZ)
```

```
            THEN
                IF (ZF =0 ) AND (Count ≠ 0)

                    THEN BranchCond ← 1;
                    ELSE BranchCond ← 0;
                FI;
        FI;
    ELSE (* instruction = LOOP *)
        IF (Count ≠ 0)
            THEN BranchCond ← 1;
            ELSE BranchCond ← 0;
        FI;
FI;
IF BranchCond = 1
    THEN
        EIP ← EIP + SignExtend(DEST);
        IF OperandSize = 16
            THEN
                EIP ← EIP AND 0000FFFFH;
        FI;
        IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
    ELSE
        Terminate loop and continue program execution at EIP;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-32_Exception	Taken Branch Debug Exception if PSR.tb is 1

Protected Mode Exceptions

#GP(0)	If the offset jumped to is beyond the limits of the code segment.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

None.

LSL—Load Segment Limit

Opcode	Instruction	Description
0F 03 /*r*	LSL *r16,r/m16*	Load: *r16* ← segment limit, selector *r/m16*
0F 03 /*r*	LSL *r32,r/m32*	Load: *r32* ← segment limit, selector *r/m32*)

Description

Loads the unscrambled segment limit from the segment descriptor specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit). If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit “raw” limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not null.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed.
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

LSL—Load Segment Limit (continued)

Type	Name	Valid
0	Reserved	No
1	Available 16-bit TSS	Yes
2	LDT	Yes
3	Busy 16-bit TSS	Yes
4	16-bit call gate	No
5	16-bit/32-bit task gate	No
6	16-bit trap gate	No
7	16-bit interrupt gate	No
8	Reserved	No
9	Available 32-bit TSS	Yes
A	Reserved	No
B	Busy 32-bit TSS	Yes
C	32-bit call gate	No
D	Reserved	No
E	32-bit trap gate	No
F	32-bit interrupt gate	No

Operation

```
IF SRC(Offset) > descriptor table limit
  THEN ZF ← 0; FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ conforming code segment
  AND (CPL > DPL) OR (RPL > DPL)
  OR Segment type is not valid for instruction
      THEN
          ZF ← 0
      ELSE
          temp ← SegmentLimit([SRC]);
          IF (G = 1)
              THEN
                  temp ← ShiftLeft(12, temp) OR 00000FFFH;
          FI;
          IF OperandSize = 32
              THEN
                  DEST ← temp;
              ELSE (*OperandSize = 16*)
                  DEST ← temp AND FFFFH;
          FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is cleared to 0.

LSL—Load Segment Limit (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

LSS—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

LTR—Load Task Register

Opcode	Instruction	Description
0F 00 /3	LTR *r/m16*	Load *r/m16* into TR

Description

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses to segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,LTR);
IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global
   THEN #GP(segment selector);
FI;
Reat segment descriptor;
IF segment descriptor is not for an available TSS THEN #GP(segment selector); FI;
IF segment descriptor is not present THEN #NP(segment selector);
TSSsegmentDescriptor(busy) ← 1;
(* Locked read-modify-write operation on the entire descriptor when setting busy flag *)
TaskRegister(SegmentSelector) ← SRC;
TaskRegister(SegmentDescriptor) ← TSSSegmentDescriptor;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

LTR—Load Task Register (continued)

#GP(selector)	If the source selector points to a segment that is not a TSS or to one for a task that is already busy.
	If the selector points to LDT or is beyond the GDT limit.
#NP(selector)	If the TSS is marked not present.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

MOV—Move

Opcode	Instruction	Description
88 /*r*	MOV *r/m8,r8*	Move *r8* to *r/m8*
89 /*r*	MOV *r/m16,r16*	Move *r16* to *r/m16*
89 /*r*	MOV *r/m32,r32*	Move *r32* to *r/m32*
8A /*r*	MOV *r8,r/m8*	Move *r/m8* to *r8*
8B /*r*	MOV *r16,r/m16*	Move *r/m16* to *r16*
8B /*r*	MOV *r32,r/m32*	Move *r/m32* to *r32*
8C /*r*	MOV *r/m16,Sreg***	Move segment register to *r/m16*
8E /*r*	MOV *Sreg,r/m16*	Move *r/m16* to segment register
A0	MOV AL,*moffs8**	Move byte at (*seg:offset*) to AL
A1	MOV AX,*moffs16**	Move word at (*seg:offset*) to AX
A1	MOV EAX,*moffs32**	Move doubleword at (*seg:offset*) to EAX
A2	MOV *moffs8**,AL	Move AL to (*seg:offset*)
A3	MOV *moffs16**,AX	Move AX to (*seg:offset*)
A3	MOV *moffs32**,EAX	Move EAX to (*seg:offset*)
B0+ *rb*	MOV *r8,imm8*	Move *imm8* to *r8*
B8+ *rw*	MOV *r16,imm16*	Move *imm16* to *r16*
B8+ *rd*	MOV *r32,imm32*	Move *imm32* to *r32*
C6 /*0*	MOV *r/m8,imm8*	Move *imm8* to *r/m8*
C7 /*0*	MOV *r/m16,imm16*	Move *imm16* to *r/m16*
C7 /*0*	MOV *r/m32,imm32*	Move *imm32* to *r/m32*

Notes:

* The *moffs8*, *moffs16*, and *moffs32* operands specify a simple offset relative to the segment base, where 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

** In 32-bit mode, the assembler may require the use of the 16-bit operand size prefix (a byte with the value 66H preceding the instruction).

Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, or a doubleword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the RET instruction.

MOV—Move (continued)

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the "Operation" algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A null segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a null value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction inhibits all external interrupts and traps until after the execution of the next instruction in the IA-32 System Environment. For the IA-64 System Environment, MOV to SS results in a IA-32_Intercept(SystemFlag) trap after the instruction completes. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, *stack-pointer value*) before an interrupt occurs. The LSS instruction offers a more efficient method of loading the SS and ESP registers.

When moving data in 32-bit mode between a segment register and a 32-bit general-purpose register, the Pentium Pro processor does not require the use of a 16-bit operand size prefix; however, some assemblers do require this prefix. The processor assumes that the sixteen least-significant bits of the general-purpose register are the destination or source operand. When moving a value from a segment selector to a 32-bit register, the processor fills the two high-order bytes of the register with zeros.

Operation

```
DEST ← SRC;
```

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```
IF SS is loaded;
  THEN
      IF segment selector is null
          THEN #GP(0);
      FI;
      IF segment selector index is outside descriptor table limits
          OR segment selector's RPL ≠ CPL
          OR segment is not a writable data segment
          OR DPL ≠ CPL
              THEN #GP(selector);
      FI;
      IF segment not marked present
          THEN #SS(selector);
  ELSE
      SS ← segment selector;
      SS ← segment descriptor;
  FI;
```

MOV—Move (continued)

```
FI;
IF DS, ES, FS or GS is loaded with non-null selector;

THEN
  IF segment selector index is outside descriptor table limits
       OR segment is not a data or readable code segment
       OR ((segment is a data or nonconforming code segment)
            AND (both RPL and CPL > DPL))
                 THEN #GP(selector);
       IF segment not marked present
            THEN #NP(selector);
  ELSE
       SegmentRegister ← segment selector;
       SegmentRegister ← segment descriptor;
  FI;
FI;
IF DS, ES, FS or GS is loaded with a null selector;
  THEN
       SegmentRegister ← null segment selector;
       SegmentRegister ← null segment descriptor;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	System Flag Intercept trap for Move to SS
IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with null segment selector.
	If the destination operand is in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a nonwritable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.

	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If attempt is made to load the CS register.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If attempt is made to load the CS register.

Virtual 8086 Mode Exceptions

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

MOV—Move to/from Control Registers

Opcode	Instruction	Description
0F 22 /*r*	MOV CR0,*r32*	Move *r32* to CR0
0F 22 /*r*	MOV CR2,*r32*	Move *r32* to CR2
0F 22 /*r*	MOV CR3,*r32*	Move *r32* to CR3
0F 22 /*r*	MOV CR4,*r32*	Move *r32* to CR4
0F 20 /*r*	MOV *r32*,CR0	Move CR0 to *r32*
0F 20 /*r*	MOV *r32*,CR2	Move CR2 to *r32*
0F 20 /*r*	MOV *r32*,CR3	Move CR3 to *r32*
0F 20 /*r*	MOV *r32*,CR4	Move CR4 to *r32*

Description

Moves the contents of a control register (CR0, CR2, CR3, or CR4) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits, regardless of the operand-size attribute. (See the *Intel Architecture Software Developer's Manual, Volume 3* for a detailed description of the flags and fields in the control registers.)

When loading a control register, a program should not attempt to change any of the reserved bits; that is, always set reserved bits to the value previously read.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are always 11B. The *r/m* field specifies the general-purpose register loaded or read.

These instructions have the following side effects:

- When writing to control register CR3, all non-global TLB entries are flushed (see the *Intel Architecture Software Developer's Manual, Volume 3*.
- When modifying any of the paging flags in the control registers (PE and PG in register CR0 and PGE, PSE, and PAE in register CR4), all TLB entries are flushed, including global entries. This operation is implementation specific for the Pentium Pro processor. Software should not depend on this functionality in future Intel Architecture processors.
- If the PG flag is set to 1 and control register CR4 is written to set the PAE flag to 1 (to enable the physical address extension mode), the pointers (PDPTRs) in the page-directory pointers table will be loaded into the processor (into internal, non-architectural registers).
- If the PAE flag is set to 1 and the PG flag set to 1, writing to control register CR3 will cause the PDPTRs to be reloaded into the processor.
- If the PAE flag is set to 1 and control register CR0 is written to set the PG flag, the PDPTRs are reloaded into the processor.

Operation

```
IF IA-64 System Environment AND Move To CR Form THEN IA-32_Intercept(INST,MOVCR);
DEST ← SRC;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

MOV—Move to/from Control Registers (continued)

Additional IA-64 System Environment Exceptions

IA-32_Intercept	Move To CR#, Mandatory Instruction Intercept.
	Move From CR#, read the virtualized control register values, CR0{15:6} return zeros.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write reserved bits in the page-directory pointers table (used in the extended physical addressing mode) when the PAE flag in control register CR4 and the PG flag in control register CR0 are set to 1.

Real Address Mode Exceptions

#GP	If an attempt is made to write a 1 to any reserved bit in CR4.

Virtual 8086 Mode Exceptions

#GP(0)	These instructions cannot be executed in virtual 8086 mode.

MOV—Move to/from Debug Registers

Opcode	Instruction	Description
0F 21/*r*	MOV *r32,* DR0-DR3	Move debug registers to *r32*
0F 21/*r*	MOV *r32,* DR4-DR5	Move debug registers to *r32*
0F 21/*r*	MOV *r32,* DR6-DR7	Move debug registers to *r32*
0F 23 /*r*	MOV DR0-DR3, *r32*	Move *r32* to debug registers
0F 23 /*r*	MOV DR4-DR5, *r32*	Move *r32* to debug registers
0F 23 /*r*	MOV DR6-DR7,*r32*	Move *r32* to debug registers

Description

Moves the contents of two or more debug registers (DR0 through DR3, DR4 and DR5, or DR6 and DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits, regardless of the operand-size attribute. (See the *Intel Architecture Software Developer's Manual, Volume 3* for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386™ and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE set in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the debug registers is loaded or read. The two bits in the *mod* field are always 11. The *r/m* field specifies the general-purpose register loaded or read.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,MOVDR);
IF ((DE = 1)  and (SRC or DEST = DR4 or DR5))
THEN
   #UD;
ELSE
   DEST ← SRC;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the DE (debug extensions) bit of CR4 is set and a MOV instruction is executed involving DR4 or DR5.

MOV—Move to/from Debug Registers (continued)

#DB	If any debug register is accessed while the GD flag in debug register DR7 is set.

Real Address Mode Exceptions

#UD	If the DE (debug extensions) bit of CR4 is set and a MOV instruction is executed involving DR4 or DR5.
#DB	If any debug register is accessed while the GD flag in debug register DR7 is set.

Virtual 8086 Mode Exceptions

#GP(0)	The debug registers cannot be loaded or read when in virtual 8086 mode.

MOVS/MOVSB/MOVSW/MOVSD—Move Data from String to String

Opcode	Instruction	Description
A4	MOVS ES:(E)DI, DS:(E)SI	Move byte at address DS:(E)SI to address ES:(E)DI
A5	MOVS ES:DI,DS:SI	Move word at address DS:SI to address ES:DI
A5	MOVS ES:EDI, DS:ESI	Move doubleword at address DS:ESI to address ES:EDI
A4	MOVSB	Move byte at address DS:(E)SI to address ES:(E)DI
A5	MOVSW	Move word at address DS:SI to address ES:DI
A5	MOVSD	Move doubleword at address DS:ESI to address ES:EDI

Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). The source operand specifies the memory location at the address DS:ESI and the destination operand specifies the memory location at address ES:EDI. (When the operand-size attribute is 16, the SI and DI register are used as the source-index and destination-index registers, respectively.) The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

The MOVSB, MOVSW, and MOVSD mnemonics are synonyms of the byte, word, and doubleword versions of the MOVS instructions. They are simpler to use, but provide no type or segment checking. (For the MOVS instruction, “DS:ESI” and “ES:EDI” must be explicitly specified in the instruction.)

After the transfer, the ESI and EDI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the ESI and EDI register are incremented; if the DF flag is 1, the ESI and EDI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The MOVS, MOVSB, MOVSW, and MOVSD instructions can be preceded by the REP prefix (see “REP/REPE/REPZ/REPNE/REPNZ—Repeat Following String Operation” on “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” on page 5-326) for block moves of ECX bytes, words, or doublewords.

Operation

```
DEST ←SRC;
IF (byte move)
  THEN IF DF = 0
       THEN (E)DI ← 1;
       ELSE (E)DI ← –1;
  FI;
  ELSE IF (word move)
       THEN IF DF = 0
            THEN DI ← 2;
            ELSE DI ← –2;
```

```
        FI;
        ELSE (* doubleword move*)
            THEN IF DF = 0

                THEN EDI ← 4;
                ELSE EDI ← –4;
            FI;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

MOVSX—Move with Sign-Extension

Opcode	Instruction	Description
0F BE /*r*	MOVSX *r16,r/m8*	Move byte to word with sign-extension
0F BE /*r*	MOVSX *r32,r/m8*	Move byte to doubleword, sign-extension
0F BF /*r*	MOVSX *r32,r/m16*	Move word to doubleword, sign-extension

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute.

Operation

```
DEST ← SignExtend(SRC);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

MOVZX—Move with Zero-Extend

Opcode	Instruction	Description
0F B6 /*r*	MOVZX *r16,r/m8*	Move byte to word with zero-extension
0F B6 /*r*	MOVZX *r32,r/m8*	Move byte to doubleword, zero-extension
0F B7 /*r*	MOVZX *r32,r/m16*	Move word to doubleword, zero-extension

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute.

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute.

Operation

```
DEST ← ZeroExtend(SRC);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

MOVZX—Move with Zero-Extend (continued)

Virtual 8086 Mode Exceptions

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

MUL—Unsigned Multiplication of AL, AX, or EAX

Opcode	Instruction	Description
F6 /4	MUL *r/m8*	Unsigned multiply (AX ← AL ∗ *r/m8*)
F7 /4	MUL *r/m16*	Unsigned multiply (DX:AX ← AX ∗ *r/m16*)
F7 /4	MUL *r/m32*	Unsigned multiply (EDX:EAX ← EAX ∗ *r/m32*)

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in the following table.

:

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX

The AH, DX, or EDX registers (depending on the operand size) contain the high-order bits of the product. If the contents of one of these registers are 0, the CF and OF flags are cleared; otherwise, the flags are set.

Operation

```
IF byte operation
  THEN
       AX ← AL ∗ SRC
  ELSE (* word or doubleword operation *)
       IF OperandSize = 16
            THEN
                 DX:AX ← AX ∗ SRC
            ELSE (* OperandSize = 32 *)
                 EDX:EAX ← EAX ∗ SRC
       FI;
FI;
```

Flags Affected

The OF and CF flags are cleared to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults — NaT Register Consumption Abort.

IA-64 Mem Faults — VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

MUL—Unsigned Multiplication of AL, AX, or EAX (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

NEG—Two's Complement Negation

Opcode	Instruction	Description
F6 /3	NEG *r/m8*	Two's complement negate *r/m8*
F7 /3	NEG *r/m16*	Two's complement negate *r/m16*
F7 /3	NEG *r/m32*	Two's complement negate *r/m32*

Description

Replaces the value of operand (the destination operand) with its two's complement. The destination operand is located in a general-purpose register or a memory location.

Operation

```
IF DEST = 0
  THEN CF ← 0
  ELSE CF ← 1;
FI;
DEST ←  – (DEST)
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

NEG—Two's Complement Negation (continued)

Virtual 8086 Mode Exceptions

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

NOP—No Operation

Opcode	Instruction	Description
90	NOP	No operation

Description

Performs no operation. This instruction is a one-byte instruction that takes up space in the instruction stream but does not affect the machine context, except the EIP register.

The NOP instruction performs no operation, no registers are accessed and no faults are generated.

Flags Affected

None.

Exceptions (All Operating Modes)

None.

NOT—One's Complement Negation

Opcode	Instruction	Description
F6 /2	NOT *r/m8*	Reverse each bit of *r/m8*
F7 /2	NOT *r/m16*	Reverse each bit of *r/m16*
F7 /2	NOT *r/m32*	Reverse each bit of *r/m32*

Description

Performs a bitwise NOT operation (1's complement) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

Operation

```
DEST ← NOT DEST;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

NOT—One's Complement Negation (continued)

Virtual 8086 Mode Exceptions

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

OR—Logical Inclusive OR

Opcode	Instruction	Description
0C *ib*	OR AL,*imm8*	AL OR *imm8*
0D *iw*	OR AX,*imm16*	AX OR *imm16*
0D *id*	OR EAX,*imm32*	EAXOR *imm32*
80 /1 *ib*	OR *r/m8,imm8*	*r/m8* OR *imm8*
81 /1 *iw*	OR *r/m16,imm16*	*r/m16* OR *imm16*
81 /1 *id*	OR *r/m32,imm32*	*r/m32* OR *imm32*
83 /1 *ib*	OR *r/m16,imm8*	*r/m16* OR *imm8*
83 /1 *ib*	OR *r/m32,imm8*	*r/m32* OR *imm8*
08 /*r*	OR *r/m8,r8*	*r/m8* OR *r8*
09 /*r*	OR *r/m16,r16*	*r/m16* OR *r16*
09 /*r*	OR *r/m32,r32*	*r/m32* OR *r32*
0A /*r*	OR *r8,r/m8*	*r8* OR *r/m8*
0B /*r*	OR *r16,r/m16*	*r16* OR *r/m16*
0B /*r*	OR *r32,r/m32*	*r32* OR *r/m32*

Description

Performs a bitwise OR operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location.

Operation

DEST ← DEST OR SRC;

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

OR—Logical Inclusive OR (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

OUT—Output to Port

Opcode	Instruction	Description
E6 *ib*	OUT *imm8*, AL	Output byte AL to *imm8* I/O port address
E7 *ib*	OUT *imm8*, AX	Output word AX to *imm8* I/O port address
E7 *ib*	OUT *imm8*, EAX	Output doubleword EAX to *imm8* I/O port address
EE	OUT DX, AL	Output byte AL to I/O port address in DX
EF	OUT DX, AX	Output word AX to I/O port address in DX
EF	OUT DX, EAX	Output doubleword EAX to I/O port address in DX

Description

Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space.

I/O transactions are performed after all prior data memory operations. No subsequent data memory operations can pass an I/O transaction.

In the IA-64 System Environment, I/O port references are mapped into the 64-bit virtual address pointed to by the IOBase register, with four ports per 4K-byte virtual page. Operating systems can utilize IA-64 TLBs to grant or deny permission to any four I/O ports. The I/O port space can be mapped into any arbitrary 64-bit physical memory location by operating system code. If CFLG.io is 1 and CPL>IOPL, the TSS is consulted for I/O permission. If CFLG.io is 0 or CPL<=IOPL, permission is granted regardless of the state of the TSS I/O permission bitmap (the bitmap is not referenced).

If the referenced I/O port is mapped to an unimplemented virtual address (via the I/O Base register) or if data translations are disabled (PSR.dt is 0) a GPFault is generated on the referencing OUT instruction.

Operation

```
IF ((PE = 1) AND ((VM = 1) OR (CPL > IOPL)))
  THEN (* Protected mode or virtual-8086 mode with CPL > IOPL *)
       IF (CFLG.io AND Any I/O Permission Bit for I/O port being accessed = 1)
            THEN #GP(0);
       FI;
  ELSE ( * Real-address mode or protected mode with CPL ≤ IOPL *)
  (* or virtual-8086 mode with all I/O permission bits for I/O port cleared *)
```

OUT—Output to Port (continued)

```
FI;
IF (IA-64_System_Environment) THEN
   DEST_VA = IOBase | (Port{15:2}<<12) | Port{11:0};
   DEST_PA = translate(DEST_VA);

   [DEST_PA] ← SRC; (* Writes to selected I/O port *)
FI;

memory_fence();
[DEST_PA] ← SRC; (* Writes to selected I/O port *)
memory_fence();
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Exception	Debug traps for data breakpoints and single step
IA-32_Exception	Alignment faults
#GP(0)	Referenced Port is to an unimplemented virtual address or PSR.dt is zero.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1 **and when CFLG.io is 1**.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

Opcode	Instruction	Description
6E	OUTS DX, DS:(E)SI	Output byte at address DS:(E)SI to I/O port in DX
6F	OUTS DX, DS:SI	Output word at address DS:SI to I/O port in DX
6F	OUTS DX, DS:ESI	Output doubleword at address DS:ESI to I/O port in DX
6E	OUTSB	Output byte at address DS:(E)SI to I/O port in DX
6F	OUTSW	Output word at address DS:SI to I/O port in DX
6F	OUTSD	Output doubleword at address DS:ESI to I/O port in DX

Description

Copies data from the second operand (source operand) to the I/O port specified with the first operand (destination operand). The source operand is a memory location at the address DS:ESI. (When the operand-size attribute is 16, the SI register is used as the source-index register.) The DS register may be overridden with a segment override prefix.

The destination operand must be the DX register, allowing I/O port addresses from 0 to 65,535 to be accessed. When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size.

The OUTSB, OUTSW and OUTSD mnemonics are synonyms of the byte, word, and doubleword versions of the OUTS instructions. (For the OUTS instruction, “DS:ESI” must be explicitly specified in the instruction.)

After the byte, word, or doubleword is transfer from the memory location to the I/O port, the ESI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the ESI register is incremented; if the DF flag is 1, the EDI register is decremented.) The ESI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” on page 5-326 for a description of the REP prefix.

After an OUTS, OUTSB, OUTSW, or OUTSD instruction is executed, the processor waits for the acknowledgment of the OUT transaction before beginning to execute the next instruction. Note that the next instruction may be prefetched, even if the OUT transaction has not completed.

This instruction is only useful for accessing I/O ports located in the processor’s I/O address space.

I/O transactions are performed after all prior data memory operations. No subsequent data memory operations can pass an I/O transaction.

OUTS/OUTSB/OUTSW/OUTSD—Output String to Port (continued)

In the IA-64 System Environment, I/O port references are mapped into the 64-bit virtual address pointed to by the IOBase register, with four ports per 4K-byte virtual page. Operating systems can utilize IA-64 TLBs to grant or deny permission to any four I/O ports. The I/O port space can be mapped into any arbitrary 64-bit physical memory location by operating system code. If CFLG.io is 1 and CPL>IOPL, the TSS is consulted for I/O permission. If CFLG.io is 0 or CPL<=IOPL, permission is granted regardless of the state of the TSS I/O permission bitmap (the bitmap is not referenced).

If the referenced I/O port is mapped to an unimplemented virtual address (via the I/O Base register) or if data translations are disabled (PSR.dt is 0) a GPFault is generated on the referencing OUTS instruction.

Operation

```
IF ((PE = 1) AND ((VM = 1) OR (CPL > IOPL)))
   THEN (* Protected mode or virtual-8086 mode with CPL > IOPL *)
        IF (CFLG.io AND Any I/O Permission Bit for I/O port being accessed = 1)
            THEN #GP(0);
        FI;
   ELSE ( * I/O operation is allowed *)
FI;

IF (IA-64_System_Environment) THEN
   DEST_VA = IOBase | (Port{15:2}<<12) | Port{11:0};
   DEST_PA = translate(DEST_VA);
   [DEST_PA] ← SRC; (* Writes to selected I/O port *)
FI;
memory_fence();
[DEST_PA] ← SRC; (* Writes to selected I/O port *)
memory_fence();

IF (byte operation)
        THEN IF DF = 0
                THEN (E)DI ← 1;
                ELSE (E)DI ← –1;
            FI;
            ELSE IF (word operation)
                THEN IF DF = 0
                    THEN DI ← 2;
                    ELSE DI ← –2;
                FI;
                ELSE (* doubleword operation *)
                    THEN IF DF = 0
                        THEN EDI ← 4;
                        ELSE EDI ← –4;
                    FI;
            FI;
        FI;
FI;
```

Flags Affected

None.

OUTS/OUTSB/OUTSW/OUTSD—Output String to Port (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Exception	Debug traps for data breakpoints and single step
IA-32_Exception	Alignment faults
#GP(0)	Referenced Port is to an unimplemented virtual address or PSR.dt is zero.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1 **and when CFLG.io is 1**.
	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a null segment selector.
	If an illegal memory operand effective address in the ES segments is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

POP—Pop a Value from the Stack

Opcode	Instruction	Description
8F /0	POP *m16*	Pop top of stack into *m16*; increment stack pointer
8F /0	POP *m32*	Pop top of stack into *m32*; increment stack pointer
58+ *rw*	POP *r16*	Pop top of stack into *r16*; increment stack pointer
58+ *rd*	POP *r32*	Pop top of stack into *r32*; increment stack pointer
1F	POP DS	Pop top of stack into DS; increment stack pointer
07	POP ES	Pop top of stack into ES; increment stack pointer
17	POP SS	Pop top of stack into SS; increment stack pointer
0F A1	POP FS	Pop top of stack into FS; increment stack pointer
0F A9	POP GS	Pop top of stack into GS; increment stack pointer

Description

Loads the value from the top of the procedure stack to the location specified with the destination operand and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

The current address-size attribute for the stack segment and the operand-size attribute determine the amount the stack pointer is incremented (see the “Operation” below). For example, if 32-bit addressing and operands are being used, the ESP register (stack pointer) is incremented by 4 and, if 16-bit addressing and operands are being used, the SP register (stack pointer for 16-bit addressing) is incremented by 2. The B flag in the stack segment’s segment descriptor determines the stack’s address-size attribute.

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the “Operation” below).

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a null value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is null.

The POP instruction cannot pop a value into the CS register. To load the CS register, use the RET instruction.

A POP SS instruction inhibits all external interrupts, including the NMI interrupt, and traps until after execution of the next instruction. **in the IA-32 System Environment. For the IA-64 System Environment, POP SS results in an IA-32_Intercept(SystemFlag) trap after the instruction completes.**This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, *stack-pointer value*) before an interrupt occurs. The LSS instruction offers a more efficient method of loading the SS and ESP registers.

POP—Pop a Value from the Stack (continued)

This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instructions computes the effective address of the operand after it increments the ESP register.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

Operation

```
IF StackAddrSize = 32
  THEN
      IF OperandSize = 32
          THEN
              DEST ← SS:ESP; (* copy a doubleword *)
              ESP ← ESP + 4;
          ELSE (* OperandSize = 16*)
              DEST ← SS:ESP; (* copy a word *)
          ESP ← ESP + 2;
      FI;
  ELSE (* StackAddrSize = 16* )
      IF OperandSize = 16
          THEN
              DEST ← SS:SP; (* copy a word *)
              SP ← SP + 2;
          ELSE (* OperandSize = 32 *)
              DEST ← SS:SP; (* copy a doubleword *)
              SP ← SP + 4;
      FI;
FI;
```

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```
IF SS is loaded;
  THEN
      IF segment selector is null
          THEN #GP(0);
      FI;
      IF segment selector index is outside descriptor table limits
          OR segment selector's RPL ≠ CPL
          OR segment is not a writable data segment
          OR DPL ≠ CPL
              THEN #GP(selector);
      FI;
      IF segment not marked present
          THEN #SS(selector);
  ELSE
      SS ← segment selector;
      SS ← segment descriptor;
  FI;
```

POP—Pop a Value from the Stack (continued)

```
FI;
IF DS, ES, FS or GS is loaded with non-null selector;
THEN
  IF segment selector index is outside descriptor table limits
       OR segment is not a data or readable code segment

       OR ((segment is a data or nonconforming code segment)
            AND (both RPL and CPL > DPL))
                 THEN #GP(selector);
       IF segment not marked present
            THEN #NP(selector);
  ELSE
       SegmentRegister ← segment selector;
       SegmentRegister ← segment descriptor;
  FI;
FI;
IF DS, ES, FS or GS is loaded with a null selector;
  THEN
       SegmentRegister ← null segment selector;
       SegmentRegister ← null segment descriptor;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	System Flag Intercept trap for POP SS
IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with null segment selector.
	If the destination operand is in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a nonwritable data segment.

POP—Pop a Value from the Stack (continued)

	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the current top of stack is not within the stack segment.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

POPA/POPAD—Pop All General-Purpose Registers

Opcode	Instruction	Description
61	POPA	Pop DI, SI, BP, BX, DX, CX, and AX
61	POPAD	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX

Description

Pops doublewords (POPAD) or words (POPA) from the procedure stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the current operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). (These instructions reverse the operation of the PUSHA/PUSHAD instructions.) The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded (see the “Operation” below).

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used. Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used.

Operation

```
IF OperandSize = 32 (* instruction = POPAD *)
THEN
   EDI ← Pop();
   ESI ← Pop();
   EBP ← Pop();
   increment ESP by 4 (* skip next 4 bytes of stack *)
   EBX ← Pop();
   EDX ← Pop();
   ECX ← Pop();
   EAX ← Pop();
ELSE (* OperandSize = 16, instruction = POPA *)
   DI ← Pop();
   SI ← Pop();
   BP ← Pop();
   increment ESP by 2 (* skip next 2 bytes of stack *)
   BX ← Pop();
   DX ← Pop();
   CX ← Pop();
   AX ← Pop();
FI;
```

Flags Affected

None.

POPA/POPAD—Pop All General-Purpose Registers (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

POPF/POPFD—Pop Stack into EFLAGS Register

Opcode	Instruction	Description
9D	POPF	Pop top of stack into EFLAGS
9D	POPFD	Pop top of stack into EFLAGS

Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register. (These instructions reverse the operation of the PUSHF/PUSHFD instructions.)

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16 and the POPFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPF is used and to 32 when POPFD is used. Others may treat these mnemonics as synonyms (POPF/POPFD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used.

The effect of the POPF/POPFD instructions on the EFLAGS register changes slightly, depending on the mode of operation of the processor. When the processor is operating in protected mode at privilege level 0 (or in real-address mode, which is equivalent to privilege level 0), all the non-reserved flags in the EFLAGS register except the VIP and VIF flags can be modified. The VIP and VIF flags are cleared.

When operating in protected mode, but with a privilege level greater an 0, all the flags can be modified except the IOPL field and the VIP and VIF flags. Here, the IOPL flags are masked and the VIP and VIF flags are cleared.

When operating in virtual-8086 mode, the I/O privilege level (IOPL) must be equal to 3 to use POPF/POPFD instructions and the VM, RF, IOPL, VIP, and VIF flags are masked. If the IOPL is less than 3, the POPF/POPFD instructions cause a general protection exception (#GP).

The IOPL is altered only when executing at privilege level 0. The interrupt flag is altered only when executing at a level at least as privileged as the IOPL. (Real-address mode is equivalent to privilege level 0.) If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

Operation

```
OLD_IF <- IF; OLD_AC <- AC; OLD_TF <- TF;
IF CR0.PE = 0 (*Real Mode *)
  THEN
      IF OperandSize = 32;
          THEN
              EFLAGS ← Pop();
              (* All non-reserved flags except VM, RF, VIP and VIF can be modified; *)
              ELSE (* OperandSize = 16 *)
              EFLAGS[15:0] ← Pop(); (* All non-reserved flags can be modified; *)
      FI;
  ELSE (*In Protected Mode *)
      IF VM=0 (* Not in Virtual-8086 Mode *)
          THEN
```

POPF/POPFD—Pop Stack into EFLAGS Register (continued)

```
                    IF CPL=0
                        THEN
                            IF OperandSize = 32;
                                THEN
                                    EFLAGS ← Pop();
                                    (* All non-reserved flags except VM, RF, VIP and VIF can be *)
                                    (* modified; *)
                                ELSE (* OperandSize = 16 *)
                                EFLAGS[15:0] ← Pop(); (* All non-reserved flags can be modified; *)
                            FI;
                        ELSE (* CPL > 0 *)
                            IF OperandSize = 32;
                                THEN
                                    EFLAGS ← Pop()
                                    (* All non-reserved bits except IOPL, RF, VM, VIP, and VIF can *)
                                    (* be modified; *)
                                    (* IOPL is masked *)
                                ELSE (* OperandSize = 16 *)
                                    EFLAGS[15:0] ← Pop();
                                    (* All non-reserved bits except IOPL can be modified; IOPL is
masked *)
                            FI;
                    FI;
                ELSE  (* In Virtual-8086 Mode *)
                    IF IOPL=3
                    THEN
                        IF OperandSize=32
                            THEN
                                EFLAGS ← Pop()
                                (* All non-reserved bits except VM, RF, IOPL, VIP, and VIF *)
                                (* can be modified; VM, RF, IOPL, VIP, and VIF are masked*)
                            ELSE
                                EFLAGS[15:0] ← Pop()
                                (* All non-reserved bits except IOPL can be modified; IOPL is *)
                                    (* masked *)
                        FI;
                    ELSE (* IOPL < 3 *)
                        IF CR4.VME = 0
                            THEN #GP(0);
                            ELSE
                                IF ((OperandSize = 32) OR (STACK.TF = 1) OR (EFLAGS.VIP = 1
                                    AND STACK.IF = 1)
                                    THEN #GP(0);
                                    ELSE
                                        TempFlags <- pop();
                                        FLAGS <- TempFlags; (*IF and IOPL bits are unchanged*)
                                        EFLAGS.VIF <- TempFlags.IF;
                                    FI;
                            FI;
                        FI;
            FI;
FI;
```

POPF/POPFD—Pop Stack into EFLAGS Register (continued)

```
IF(IA-64 System Environment AND (AC, TF != OLD_AC, OLD_TF)
   THEN IA-32_Intercept(System_Flag,POPF);
IF IA-64 System Environment AND CFLG.ii AND IF != OLD_IF
   THEN IA-32_Intercept(System_Flag,POPF);
```

Flags Affected

All flags except the reserved bits.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Intercept	System Flag Intercept Trap if CFLG.ii is 1 and the IF flag changes state or if the AC, RF or TF changes state.

Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the I/O privilege level is less than 3. If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

PUSH—Push Word or Doubleword Onto the Stack

Opcode	Instruction	Description
FF /6	PUSH *r/m16*	Push *r/m16*
FF /6	PUSH *r/m32*	Push *r/m32*
50+*rw*	PUSH *r16*	Push *r16*
50+*rd*	PUSH *r32*	Push *r32*
6A	PUSH *imm8*	Push *imm8*
68	PUSH *imm16*	Push *imm16*
68	PUSH *imm32*	Push *imm32*
0E	PUSH CS	Push CS
16	PUSH SS	Push SS
1E	PUSH DS	Push DS
06	PUSH ES	Push ES
0F A0	PUSH FS	Push FS
0F A8	PUSH GS	Push GS

Description

Decrements the stack pointer and then stores the source operand on the top of the procedure stack. The current address-size attribute for the stack segment and the operand-size attribute determine the amount the stack pointer is decremented (see the “Operation” below). For example, if 32-bit addressing and operands are being used, the ESP register (stack pointer) is decremented by 4 and, if 16-bit addressing and operands are being used, the SP register (stack pointer for 16-bit addressing) is decremented by 2. Pushing 16-bit operands when the stack address-size attribute is 32 can result in a misaligned the stack pointer (that is, the stack pointer not aligned on a doubleword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. Thus, if a PUSH instruction uses a memory operand in which the ESP register is used as a base register for computing the operand address, the effective address of the operand is computed before the ESP register is decremented.

In the real-address mode, if the ESP or SP register is 1 when the PUSH instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Operation

```
IF StackAddrSize = 32
THEN
  IF OperandSize = 32
      THEN
          ESP ← ESP – 4;
          SS:ESP ← SRC; (* push doubleword *)
      ELSE (* OperandSize = 16*)
          ESP ← ESP – 2;
          SS:ESP ← SRC; (* push word *)
  FI;
ELSE (* StackAddrSize = 16*)
```

PUSH—Push Word or Doubleword Onto the Stack (continued)

```
  IF OperandSize = 16
       THEN

            SP ← SP – 2;
             SS:SP ← SRC; (* push word *)
       ELSE (* OperandSize = 32*)
            SP ← SP – 4;
            SS:SP ← SRC; (* push doubleword *)
  FI;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
	If the new value of the SP or ESP register is outside the stack segment limit.

Virtual 8086 Mode Exceptions

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

PUSH—Push Word or Doubleword Onto the Stack (continued)

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

Intel Architecture Compatibility

For Intel Architecture processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true in the real-address and virtual-8086 modes.) For the Intel 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	Description
60	PUSHA	Push AX, CX, DX, BX, original SP, BP, SI, and DI
60	PUSHAD	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

Description

Push the contents of the general-purpose registers onto the procedure stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, EBP, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). (These instructions perform the reverse operation of the POPA/POPAD instructions.) The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the “Operation” below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Operation

```
IF OperandSize = 32 (* PUSHAD instruction *)
  THEN
       Temp ← (ESP);
       Push(EAX);
       Push(ECX);
       Push(EDX);
       Push(EBX);
       Push(Temp);
       Push(EBP);
       Push(ESI);
       Push(EDI);
  ELSE (* OperandSize = 16, PUSHA instruction *)
       Temp ← (SP);
       Push(AX);
       Push(CX);
       Push(DX);
       Push(BX);
       Push(Temp);
       Push(BP);
       Push(SI);
       Push(DI);
FI;
```

PUSHA/PUSHAD—Push All General-Purpose Registers (continued)

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs.

Real Address Mode Exceptions

#GP	If the ESP or SP register contains 7, 9, 11, 13, or 15.

Virtual 8086 Mode Exceptions

#GP(0)	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#PF(fault-code)	If a page fault occurs.

PUSHF/PUSHFD—Push EFLAGS Register onto the Stack

Opcode	Instruction	Description
9C	PUSHF	Push EFLAGS
9C	PUSHFD	Push EFLAGS

Description

Decrement the stack pointer by 4 (if the current operand-size attribute is 32) and push the entire contents of the EFLAGS register onto the procedure stack or decrement the stack pointer by 2 (if the operand-size attribute is 16) push the lower 16 bits of the EFLAGS register onto the stack. (These instructions reverse the operation of the POPF/POPFD instructions.)

When copying the entire EFLAGS register to the stack, bits 16 and 17, called the VM and RF flags, are not copied. Instead, the values for these flags are cleared in the EFLAGS image stored on the stack.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

When the I/O privilege level (IOPL) is less than 3 in virtual-8086 mode, the PUSHF/PUSHFD instructions causes a general protection exception (#GP). The IOPL is altered only when executing at privilege level 0. The interrupt flag is altered only when executing at a level at least as privileged as the IOPL. (Real-address mode is equivalent to privilege level 0.) If a PUSHF/PUSHFD instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Operation

```
IF VM=0 (* Not in Virtual-8086 Mode *)
  THEN
      IF OperandSize = 32
          THEN
              push(EFLAGS AND 00FCFFFFH);
              (* VM and RF EFLAG bits are cleared in image stored on the stack*)
          ELSE
              push(EFLAGS); (* Lower 16 bits only *)
      FI;
  ELSE (* In Virtual-8086 Mode *)
      IF IOPL=3
          THEN
              IF OperandSize = 32
                  THEN push(EFLAGS AND 0FCFFFFH);
                  (* VM and RF EFLAGS bits are cleared in image stored on the stack*)
                  ELSE push(EFLAGS); (* Lower 16 bits only *)
```

PUSHF/PUSHFD—Push EFLAGS Register onto the Stack (continued)

```
            FI;
        ELSE (*IOPL < 3*)

            IF OperandSize =32 OR CR$.VME=0
                THEN #GP(0); (* Trap to virtual-8086 monitor *)
                ELSE
                    TempFlags <- FLAGS OR 3000H; (*Set IOPL bits to 11B or IOPL 3 *)
                    TempFlags.IF <- EFLAGS.VIF;
                    push(TempFlags);
            FI;
    FI;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.

RCL/RCR/ROL/ROR-—Rotate

Opcode	Instruction	Description
D0 /2	RCL *r/m8*,1	Rotate 9 bits (CF,*r/m8*) left once
D2 /2	RCL *r/m8*,CL	Rotate 9 bits (CF,*r/m8*) left CL times
C0 /2 *ib*	RCL *r/m8,imm8*	Rotate 9 bits (CF,*r/m8*) left *imm8* times
D1 /2	RCL *r/m16*,1	Rotate 17 bits (CF,*r/m16*) left once
D3 /2	RCL *r/m16*,CL	Rotate 17 bits (CF,*r/m16*) left CL times
C1 /2 *ib*	RCL *r/m16,imm8*	Rotate 17 bits (CF,*r/m16*) left *imm8* times
D1 /2	RCL *r/m32*,1	Rotate 33 bits (CF,*r/m32*) left once
D3 /2	RCL *r/m32*,CL	Rotate 33 bits (CF,*r/m32*) left CL times
C1 /2 *ib*	RCL *r/m32,imm8*	Rotate 33 bits (CF,*r/m32*) left *imm8* times
D0 /3	RCR *r/m8*,1	Rotate 9 bits (CF,*r/m8*) right once
D2 /3	RCR *r/m8*,CL	Rotate 9 bits (CF,*r/m8*) right CL times
C0 /3 *ib*	RCR *r/m8,imm8*	Rotate 9 bits (CF,*r/m8*) right *imm8* times
D1 /3	RCR *r/m16*,1	Rotate 17 bits (CF,*r/m16*) right once
D3 /3	RCR *r/m16*,CL	Rotate 17 bits (CF,*r/m16*) right CL times
C1 /3 *ib*	RCR *r/m16,imm8*	Rotate 17 bits (CF,*r/m16*) right *imm8* times
D1 /3	RCR *r/m32*,1	Rotate 33 bits (CF,*r/m32*) right once
D3 /3	RCR *r/m32*,CL	Rotate 33 bits (CF,*r/m32*) right CL times
C1 /3 *ib*	RCR *r/m32,imm8*	Rotate 33 bits (CF,*r/m32*) right *imm8* times
D0 /0	ROL *r/m8*,1	Rotate 8 bits *r/m8* left once
D2 /0	ROL *r/m8*,CL	Rotate 8 bits *r/m8* left CL times
C0 /0 *ib*	ROL *r/m8,imm8*	Rotate 8 bits *r/m8* left *imm8* times
D1 /0	ROL *r/m16*,1	Rotate 16 bits *r/m16* left once
D3 /0	ROL *r/m16*,CL	Rotate 16 bits *r/m16* left CL times
C1 /0 *ib*	ROL *r/m16,imm8*	Rotate 16 bits *r/m16* left *imm8* times
D1 /0	ROL *r/m32*,1	Rotate 32 bits *r/m32* left once
D3 /0	ROL *r/m32*,CL	Rotate 32 bits *r/m32* left CL times
C1 /0 *ib*	ROL *r/m32,imm8*	Rotate 32 bits *r/m32* left *imm8* times
D0 /1	ROR *r/m8*,1	Rotate 8 bits *r/m8* right once
D2 /1	ROR *r/m8*,CL	Rotate 8 bits *r/m8* right CL times
C0 /1 *ib*	ROR *r/m8,imm8*	Rotate 8 bits *r/m16* right *imm8* times
D1 /1	ROR *r/m16*,1	Rotate 16 bits *r/m16* right once
D3 /1	ROR *r/m16*,CL	Rotate 16 bits *r/m16* right CL times
C1 /1 *ib*	ROR *r/m16,imm8*	Rotate 16 bits *r/m16* right *imm8* times
D1 /1	ROR *r/m32*,1	Rotate 32 bits *r/m32* right once
D3 /1	ROR *r/m32*,CL	Rotate 32 bits *r/m32* right CL times
C1 /1 *ib*	ROR *r/m32,imm8*	Rotate 32 bits *r/m32* right *imm8* times

RCL/RCR/ROL/ROR-—Rotate (continued)

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases. For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

Operation

```
SIZE ← OperandSize
CASE (determine count) OF
   SIZE = 8:     tempCOUNT ← (COUNT AND 1FH) MOD 9;
   SIZE = 16:    tempCOUNT ← (COUNT AND 1FH) MOD 17;
   SIZE = 32:    tempCOUNT ← COUNT AND 1FH;
ESAC;
(* ROL instruction operation *)
WHILE (tempCOUNT ≠ 0)
   DO
        tempCF ← MSB(DEST);
        DEST ← (DEST * 2) + tempCF;
        tempCOUNT ← tempCOUNT – 1;
   OD;
ELIHW;
CF ← tempCF;
IF COUNT = 1
   THEN OF ← MSB(DEST) XOR CF;
   ELSE OF is undefined;
FI;
(* ROR instruction operation *)
WHILE (tempCOUNT ≠ 0)
   DO
        tempCF ← LSB(SRC);
        DEST ← (DEST / 2) + (tempCF * 2^SIZE);
```

```
        tempCOUNT ← tempCOUNT – 1;
   OD;
IF COUNT = 1
   THEN OF ← MSB(DEST) XOR MSB – 1(DEST);
   ELSE OF is undefined;
FI;
(* RCL instruction operation *)
WHILE (tempCOUNT ≠ 0)
   DO
        tempCF ← MSB(DEST);
        DEST ← (DEST ∗ 2) + tempCF;
        tempCOUNT ← tempCOUNT – 1;
   OD;
ELIHW;
CF ← tempCF;
IF COUNT = 1
   THEN OF ← MSB(DEST) XOR CF;
   ELSE OF is undefined;
FI;
(* RCR instruction operation *)
WHILE (tempCOUNT ≠ 0)
   DO
        tempCF ← LSB(SRC);
        DEST ← (DEST / 2) + (tempCF * 2^SIZE);
        tempCOUNT ← tempCOUNT – 1;
   OD;
IF COUNT = 1
IF COUNT = 1
   THEN OF ← MSB(DEST) XOR MSB – 1(DEST);
   ELSE OF is undefined;
FI;
```

Flags Affected

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates (see “Description” above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

RCL/RCR/ROL/ROR-—Rotate (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

Intel Architecture Compatibility

The 8086 does not mask the rotation count. All Intel Architecture processors from the Intel386™ processor on do mask the rotation count in all operating modes.

RDMSR—Read from Model Specific Register

Opcode	Instruction	Description
0F 32	RDMSR	Load MSR specified by ECX into EDX:EAX

Description

Loads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. If less than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors.

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

See model specific instructions for all the MSRs that can be written to with this instruction and their addresses

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,RDMSR);
EDX:EAX ← MSR[ECX];
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.

If the value in ECX specifies a reserved or unimplemented MSR address.

Real Address Mode Exceptions

#GP If the current privilege level is not 0

If the value in ECX specifies a reserved or unimplemented MSR address.

Virtual 8086 Mode Exceptions

#GP(0) The RDMSR instruction is not recognized in virtual 8086 mode.

RDMSR—Read from Model Specific Register (continued)

Intel Architecture Compatibility

The MSRs and the ability to read them with the RDMSR instruction were introduced into the Intel Architecture with the Pentium processor. Execution of this instruction by an Intel Architecture processor earlier than the Pentium processor results in an invalid opcode exception #UD.

RDPMC—Read Performance-Monitoring Counters

Opcode	Instruction	Description
0F 33	RDPMC	Read performance-monitoring counter specified by ECX into EDX:EAX

Description

Loads the contents of the N-bit performance-monitoring counter specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order N-32 bits of the counter and the EAX register is loaded with the low-order 32 bits.

The RDPMC instruction allows application code running at a privilege level of 1, 2, or 3 to read the performance-monitoring counters if the PCE flag in the CR4 register is set for IA-32 System Environment operation or in the IA-64 System Environment if the performance counters have been configured as user level counters. This instruction is provided to allow performance monitoring by application code without incurring the overhead of a call to an operating-system procedure.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads.

The RDPMC instruction does not serialize instruction execution. That is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must use a serializing instruction (such as the CPUID instruction) before and/or after the execution of the RDPCM instruction.

The RDPMC instruction can execute in 16-bit addressing mode or virtual 8086 mode; however, the full contents of the ECX register are used to determine the counter to access and a full N-bit result is returned (the low-order 32 bits in the EAX register and the high-order N-32 bits in the EDX register).

Operation

```
IF (ECX != Implemented Counters) THEN #GP(0)
IF (IA-64 System Environment)
THEN
  SECURED = PSR.sp || CR4.pce==0;
  IF ((PSR.cpl ==0) || (PSR.cpl!=0 && ~PMC[ECX].pm && ~SECURED)))
      THEN
          EDX:EAX ← PMD[ECX+4];
      ELSE
          #GP(0)
  FI;
ELSE
  IF ((CR4.PCE = 1 OR ((CR4.PCE = 0 ) AND (CPL=0)))
      THEN
          EDX:EAX ← PMD[ECX+4];
      ELSE (* CR4.PCE is 0 and CPL is 1, 2, or 3 *)
          #GP(0)
  FI;
FI;
```

RDPMC—Read Performance-Monitoring Counters (continued)

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
#GP(0)	If the current privilege level is not 0 and the selected PMD register's PM bit is 1, or if PSR.sp is 1.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear /*In IA-32 System Environment*/.
	If the value in the ECX register does not match an implemented performance counter.

Real Address Mode Exceptions

#GP	If the PCE flag in the CR4 register is clear. /*In the IA-32 System Environment*/
	If the value in the ECX register does not match an implemented performance counter.

Virtual 8086 Mode Exceptions

#GP(0)	If the PCE flag in the CR4 register is clear. /*In the IA-32 System Environment*/
	If the value in the ECX register does not match an implemented performance counter.

RDTSC—Read Time-Stamp Counter

Opcode	Instruction	Description
0F 31	RDTSC	Read time-stamp counter into EDX:EAX

Description

Loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset.

In the IA-32 System Environment, the time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction. When the TSD flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0. The time-stamp counter can also be read with the RDMSR instruction.

In the IA-64 System Environment, PSR.si and CR4.TSD restricts the use of the RDTSC instruction. When PSR.si is clear and CR4.TSD is clear, the RDTSC instruction can be executed at any privilege level; when PSR.si is set or CR4.TSD is set, the instruction can only be executed at privilege level 0.

The RDTSC instruction is not serializing instruction. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed.

This instruction was introduced into the Intel Architecture in the Pentium processor.

Operation

```
IF (IA-32 System Environement)
  IF (CR4.TSD = 0) OR ((CR4.TSD = 1) AND (CPL=0))
      THEN
          EDX:EAX ← TimeStampCounter;
      ELSE (* CR4 is 1 and CPL is 1, 2, or 3 *)
          #GP(0)
  FI;
ELSE /*IA-64 System Environment*/
  SECURED = PSR.si || CR4.TSD;
  IF (!SECURED) OR (SECURED AND (CPL=0))
      THEN
          EDX:EAX ← TimeStampCounter;
      ELSE (* CR4 is 1 and CPL is 1, 2, or 3 *)
          #GP(0)
  FI;
FI;
```

Flags Affected

None.

RDTSC—Read Time-Stamp Counter (continued)

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
#GP(0)	If PSR.si is 1 or CR4.TSD is 1 and the CPL is greater than 0.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0. /*For the IA-32 System Environment only*/

Real Address Mode Exceptions

#GP	If the TSD flag in register CR4 is set. /*For the IA-32 System Environment only*/

Virtual 8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set. /*For the IA-32 System Environment only*/

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

F3 6C	REP INS *r/m8*, DX	Input ECX bytes from port DX into ES:[EDI]
F3 6D	REP INS *r/m16*,DX	Input ECX words from port DX into ES:[EDI]
F3 6D	REP INS *r/m32*,DX	Input ECX doublewords from port DX into ES:[EDI]
F3 A4	REP MOVS *m8,m8*	Move ECX bytes from DS:[ESI] to ES:[EDI]
F3 A5	REP MOVS *m16,m16*	Move ECX words from DS:[ESI] to ES:[EDI]
F3 A5	REP MOVS *m32,m32*	Move ECX doublewords from DS:[ESI] to ES:[EDI]
F3 6E	REP OUTS DX,*r/m8*	Output ECX bytes from DS:[ESI] to port DX
F3 6F	REP OUTS DX,*r/m16*	Output ECX words from DS:[ESI] to port DX
F3 6F	REP OUTS DX,*r/m32*	Output ECX doublewords from DS:[ESI] to port DX
F3 AC	REP LODS AL	Load ECX bytes from DS:[ESI] to AL
F3 AD	REP LODS AX	Load ECX words from DS:[ESI] to AX
F3 AD	REP LODS EAX	Load ECX doublewords from DS:[ESI] to EAX
F3 AA	REP STOS *m8*	Fill ECX bytes at ES:[EDI] with AL
F3 AB	REP STOS *m16*	Fill ECX words at ES:[EDI] with AX
F3 AB	REP STOS *m32*	Fill ECX doublewords at ES:[EDI] with EAX
F3 A6	REPE CMPS *m8,m8*	Find nonmatching bytes in ES:[EDI] and DS:[ESI]
F3 A7	REPE CMPS *m16,m16*	Find nonmatching words in ES:[EDI] and DS:[ESI]
F3 A7	REPE CMPS *m32,m32*	Find nonmatching doublewords in ES:[EDI] and DS:[ESI]
F3 AE	REPE SCAS *m8*	Find non-AL byte starting at ES:[EDI]
F3 AF	REPE SCAS *m16*	Find non-AX word starting at ES:[EDI]
F3 AF	REPE SCAS *m32*	Find non-EAX doubleword starting at ES:[EDI]
F2 A6	REPNE CMPS *m8,m8*	Find matching bytes in ES:[EDI] and DS:[ESI]
F2 A7	REPNE CMPS *m16,m16*	Find matching words in ES:[EDI] and DS:[ESI]
F2 A7	REPNE CMPS *m32,m32*	Find matching doublewords in ES:[EDI] and DS:[ESI]
F2 AE	REPNE SCAS *m8*	Find AL, starting at ES:[EDI]
F2 AF	REPNE SCAS *m16*	Find AX, starting at ES:[EDI]
F2 AF	REPNE SCAS *m32*	Find EAX, starting at ES:[EDI]

Description

Repeats a string instruction the number of times specified in the count register (ECX) or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix (continued)

All of these repeat prefixes cause the associated instruction to be repeated until the count in register ECX is decremented to 0 (see the following table). The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the ECX register with a JECXZ instruction or by testing the ZF flag with a JZ, JNZ, and JNE instruction.

Table 5-17. Repeat Conditions

Repeat Prefix	Termination Condition 1	Termination Condition 2
REP	ECX=0	None
REPE/REPZ	ECX=0	ZF=0
REPNE/REPNZ	ECX=0	ZF=1

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a page fault occurs during CMPS or SCAS instructions that are prefixed with REPNE, the EFLAGS value may NOT be restored to the state prior to the execution of the instruction. Since SCAS and CMPS do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute.

A REP STOS instruction is the fastest way to initialize a large block of memory.

Operation

```
IF AddressSize = 16
  THEN
       use CX for CountReg;
  ELSE (* AddressSize = 32 *)
       use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
  DO
       service pending interrupts (if any);
       execute associated string instruction;
       CountReg ← CountReg – 1;
       IF CountReg = 0
            THEN exit WHILE loop
       FI;
```

```
        IF (repeat prefix is REPZ or REPE) AND (ZF=0)
        OR (repeat prefix is REPNZ or REPNE) AND (ZF=1)

            THEN exit WHILE loop
        FI;
    OD;
```

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Exceptions (All Operating Modes)

None; however, exceptions can be generated by the instruction a repeat prefix is associated with.

RET—Return from Procedure

Opcode	Instruction	Description
C3	RET	Near return to calling procedure
CB	RET	Far return to calling procedure
C2 *iw*	RET *imm16*	Near return to calling procedure and pop *imm16* bytes from stack
CA *iw*	RET *imm16*	Far return to calling procedure and pop *imm16* bytes from stack

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed.

The RET instruction can be used to execute three different types of returns:

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

The inter-privilege-level return type can only be executed in protected mode.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the procedure stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the procedure stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

RET—Return from Procedure (continued)

Operation

```
(* Near return *)
IF instruction = near return
   THEN;
        IF OperandSize = 32
             THEN
                  IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
                  EIP ← Pop();
             ELSE (* OperandSize = 16 *)
                  IF top 6 bytes of stack not within stack limits
                       THEN #SS(0)
                  FI;
                  tempEIP ← Pop();
                  tempEIP ← tempEIP AND 0000FFFFH;
                  IF tempEIP not within code segment limits THEN #GP(0); FI;
                  EIP ← tempEIP;
        FI;
   IF instruction has immediate operand
        THEN IF StackAddressSize=32
             THEN
                  ESP ← ESP + SRC;
             ELSE (* StackAddressSize=16 *)
                  SP ← SP + SRC;
        FI;
   FI;
   IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) OR (PE = 1 AND VM = 1)) AND instruction = far return
   THEN;
        IF OperandSize = 32
             THEN
                  IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
                  EIP ← Pop();
                  CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
             ELSE (* OperandSize = 16 *)
                  IF top 6 bytes of stack not within stack limits THEN #SS(0); FI;
                  tempEIP ← Pop();
                  tempEIP ← tempEIP AND 0000FFFFH;
                  IF tempEIP not within code segment limits THEN #GP(0); FI;
                  EIP ← tempEIP;
                  CS ← Pop(); (* 16-bit pop *)
        FI;
   IF instruction has immediate operand THEN SP ← SP + (SRC AND FFFFH); FI;
   IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);
FI;

(* Protected mode, not virtual 8086 mode *)
IF (PE = 1 AND VM = 0) AND instruction = far RET
   THEN
        IF OperandSize = 32
             THEN
```

RET—Return from Procedure (continued)

```
                    IF second doubleword on stack is not within stack limits THEN #SS(0); FI;
                ELSE (* OperandSize = 16 *)
                    IF second word on stack is not within stack limits THEN #SS(0); FI;
            FI;
        IF return code segment selector is null THEN GP(0); FI;
        IF return code segment selector addrsses descriptor beyond diescriptor table limit
            THEN GP(selector; FI;
        Obtain descriptor to which return code segment selector points from descriptor table
        IF return code segment descriptor is not a code segment THEN #GP(selector); FI;
        if return code segment selector RPL < CPL THEN #GP(selector); FI;
        IF return code segment descriptor is condorming
            AND return code segment DPL > return code segment selector RPL
                THEN #GP(selector); FI;
        IF return code segment descriptor is not present THEN #NP(selector); FI:
        IF return code segment selector RPL > CPL
            THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
            ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL
        FI;
    END;FI;

    RETURN-SAME-PRIVILEGE-LEVEL:
        IF the return instruction pointer is not within ther return code segment limit
            THEN #GP(0);
        FI;
        IF OperandSize=32
            THEN
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
                ESP ← ESP + SRC;
            ELSE (* OperandSize=16 *)
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop *)
                ESP ← ESP + SRC;
        FI;
        IF IA-64 System Environment AND PSR.tb THEN IA-32_Exception(Debug);


    RETURN-OUTER-PRIVILEGE-LEVEL:

        IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize=32)
            OR top (8 + SRC) bytes of stack are not within stack limits (OperandSize=16)
                THEN #SS(0); FI;
        FI;
        Read return segment selector;
        IF stack segment selector is null THEN #GP(0); FI;
        IF return stack segment selector index is not within its descriptor table limits
                THEN #GP(selector); FI;
        Read segment descriptor pointed to by return segment selector;
        IF stack segment selector RPL ≠ RPL of the return code segment selector
            OR stack segment is not a writable data segment
            OR stack segment descriptor DPL ≠ RPL of the return code segment selector
                THEN #GP(selector); FI;
            IF stack segment not present THEN #SS(StackSegmentSelector); FI;
```

RET—Return from Procedure (continued)

```
IF the return instruction pointer is not within the return code segment limit THEN #GP(0); FI:
 CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize=32
     THEN
          EIP ← Pop();
          CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
           (* segment descriptor information also loaded *)
          CS(RPL) ← CPL;
          ESP ← ESP + SRC;
          tempESP ← Pop();
          tempSS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
           (* segment descriptor information also loaded *)
          ESP ← tempESP;
          SS ← tempSS;
     ELSE (* OperandSize=16 *)
          EIP ← Pop();
          EIP ← EIP AND 0000FFFFH;
          CS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
          CS(RPL) ← CPL;
          ESP ← ESP + SRC;
          tempESP ← Pop();
          tempSS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
           (* segment descriptor information also loaded *)
          ESP ← tempESP;
          SS ← tempSS;
FI;
FOR each of segment register (ES, FS, GS, and DS)
     DO;
          IF segment register points to data or non-conforming code segment
          AND CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
               THEN (* segment register invalid *)
                    SegmentSelector/Descriptor ← 0; (* null segment selector *)
          FI;
     OD;
For each of ES, FS, GS, and DS
DO
     IF segment descriptor indicates the segment is not a data or
               readable code segment
          OR if the segment is a data or non-conforming code segment and the segment
               descriptor's DPL < CPL or RPL of code segment's segment selector
               THEN
                    segment selector register ← null selector;
OD;
```

Flags Affected

None.

RET—Return from Procedure (continued)

Additional IA-64 System Environment Exceptions

IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Exception	Taken Branch Debug Exception if PSR.tb is 1

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector null.
	If the return instruction pointer is not within the return code segment limit
#GP(selector)	If the RPL of the return code segment selector is less then the CPL.
	If the return code or stack segment selector index is not within its descriptor table limits.
	If the return code segment descriptor does not indicate a code segment.
	If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector
	If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits.
	If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

Virtual 8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit
#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

ROL/ROR—Rotate

See entry for RCL/RCR/ROL/ROR.

RSM—Resume from System Management Mode

Opcode	Instruction	Description
0F AA	RSM	Resume operation of interrupted program

Description

Returns program control from system management mode (SMM) to the application program or operating system procedure that was interrupted when the processor received an SSM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486 only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

See Chapter 9 in the *Intel Architecture Software Developer's Manual, Volume 3* for more information about SMM and the behavior of the RSM instruction.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,RSM);
ReturnFromSSM;
ProcessorState ← Restore(SSMDump);
```

Flags Affected

All.

Additional IA-64 System Environment Exceptions

IA-32_Intercept Mandatory Instruction Intercept.

Protected Mode Exceptions

#UD If an attempt is made to execute this instruction when the processor is not in SMM.

Real Address Mode Exceptions

#UD If an attempt is made to execute this instruction when the processor is not in SMM.

Virtual 8086 Mode Exceptions

#UD If an attempt is made to execute this instruction when the processor is not in SMM.

SAHF—Store AH into Flags

Opcode	Instruction	Clocks	Description
9E	SAHF	2	Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register

Description

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS registers are set as shown in the “Operation” below

Operation

```
EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
```

Flags Affected

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are set to 1, 0, and 0, respectively.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

SAL/SAR/SHL/SHR—Shift Instructions

Opcode	Instruction	Description
D0 /4	SAL *r/m8*,1	Multiply *r/m8* by 2, once
D2 /4	SAL *r/m8*,CL	Multiply *r/m8* by 2, CL times
C0 /4 *ib*	SAL *r/m8,imm8*	Multiply *r/m8* by 2, *imm8* times
D1 /4	SAL *r/m16*,1	Multiply *r/m16* by 2, once
D3 /4	SAL *r/m16*,CL	Multiply *r/m16* by 2, CL times
C1 /4 *ib*	SAL *r/m16,imm8*	Multiply *r/m16* by 2, *imm8* times
D1 /4	SAL *r/m32*,1	Multiply *r/m32* by 2, once
D3 /4	SAL *r/m32*,CL	Multiply *r/m32* by 2, CL times
C1 /4 *ib*	SAL *r/m32,imm8*	Multiply *r/m32* by 2, *imm8* times
D0 /7	SAR *r/m8*,1	Signed divide* *r/m8* by 2, once
D2 /7	SAR *r/m8*,CL	Signed divide* *r/m8* by 2, CL times
C0 /7 *ib*	SAR *r/m8,imm8*	Signed divide* *r/m8* by 2, *imm8* times
D1 /7	SAR *r/m16*,1	Signed divide* *r/m16* by 2, once
D3 /7	SAR *r/m16*,CL	Signed divide* *r/m16* by 2, CL times
C1 /7 *ib*	SAR *r/m16,imm8*	Signed divide* *r/m16* by 2, *imm8* times
D1 /7	SAR *r/m32*,1	Signed divide* *r/m32* by 2, once
D3 /7	SAR *r/m32*,CL	Signed divide* *r/m32* by 2, CL times
C1 /7 *ib*	SAR *r/m32,imm8*	Signed divide* *r/m32* by 2, *imm8* times
D0 /4	SHL *r/m8*,1	Multiply *r/m8* by 2, once
D2 /4	SHL *r/m8*,CL	Multiply *r/m8* by 2, CL times
C0 /4 *ib*	SHL *r/m8,imm8*	Multiply *r/m8* by 2, *imm8* times
D1 /4	SHL *r/m16*,1	Multiply *r/m16* by 2, once
D3 /4	SHL *r/m16*,CL	Multiply *r/m16* by 2, CL times
C1 /4 *ib*	SHL *r/m16,imm8*	Multiply *r/m16* by 2, *imm8* times
D1 /4	SHL *r/m32*,1	Multiply *r/m32* by 2, once
D3 /4	SHL *r/m32*,CL	Multiply *r/m32* by 2, CL times
C1 /4 *ib*	SHL *r/m32,imm8*	Multiply *r/m32* by 2, *imm8* times
D0 /5	SHR *r/m8*,1	Unsigned divide *r/m8* by 2, once
D2 /5	SHR *r/m8*,CL	Unsigned divide *r/m8* by 2, CL times
C0 /5 *ib*	SHR *r/m8,imm8*	Unsigned divide *r/m8* by 2, *imm8* times
D1 /5	SHR *r/m16*,1	Unsigned divide *r/m16* by 2, once
D3 /5	SHR *r/m16*,CL	Unsigned divide *r/m16* by 2, CL times
C1 /5 *ib*	SHR *r/m16,imm8*	Unsigned divide *r/m16* by 2, *imm8* times
D1 /5	SHR *r/m32*,1	Unsigned divide *r/m32* by 2, once
D3 /5	SHR *r/m32*,CL	Unsigned divide *r/m32* by 2, CL times
C1 /5 *ib*	SHR *r/m32,imm8*	Unsigned divide *r/m32* by 2, *imm8* times

Note:
* Not the same form of division as IDIV; rounding is toward negative infinity.

SAL/SAR/SHL/SHR—Shift Instructions (continued)

Description

Shift the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or register CL. The count is masked to 5 bits, which limits the count range to from 0 to 31. A special opcode encoding is provide for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared.

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit; the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value.

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is cleared to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

Operation

```
tempCOUNT ← COUNT;
tempDEST ← DEST;
WHILE (tempCOUNT ≠ 0)
DO
  IF instruction is SAL or SHL
      THEN
          CF ← MSB(DEST);
```

SAL/SAR/SHL/SHR—Shift Instructions (continued)

```
        ELSE (* instruction is SAR or SHR *)
            CF ← LSB(DEST);

    FI;
    IF instruction is SAL or SHL
        THEN
            DEST ← DEST * 2;
        ELSE
            IF instruction is SAR
                THEN
                    DEST ← DEST / 2 (*Signed divide, rounding toward negative infinity*);
                ELSE (* instruction is SHR *)
                    DEST ← DEST / 2 ; (* Unsigned divide *);
            FI;
    FI;
    temp ← temp – 1;
OD;
(* Determine overflow for the various instructions *)
IF COUNT = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF ← MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF ← 0;
                    ELSE (* instruction is SHR *)
                        OF ← MSB(tempDEST);
                FI;
        FI;
    ELSE
        OF ← undefined;
FI;
```

Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions count is greater than or equal to the size of the destination operand. The OF flag is affected only for 1-bit shifts (see “Description” above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

Intel Architecture Compatibility

The 8086 does not mask the shift count. All Intel Architecture processors from the Intel386 processor on do mask the rotation count in all operating modes.

SBB—Integer Subtraction with Borrow

Opcode	Instruction	Description
1C *ib*	SBB AL,*imm8*	Subtract with borrow *imm8* from AL
1D *iw*	SBB AX,*imm16*	Subtract with borrow *imm16* from AX
1D *id*	SBB EAX,*imm32*	Subtract with borrow *imm32* from EAX
80 /3 *ib*	SBB *r/m8,imm8*	Subtract with borrow *imm8* from *r/m8*
81 /3 *iw*	SBB *r/m16,imm16*	Subtract with borrow *imm16* from *r/m16*
81 /3 *id*	SBB *r/m32,imm32*	Subtract with borrow *imm32* from *r/m32*
83 /3 *ib*	SBB *r/m16,imm8*	Subtract with borrow sign-extended *imm8* from *r/m16*
83 /3 *ib*	SBB *r/m32,imm8*	Subtract with borrow sign-extended *imm8* from *r/m32*
18 /*r*	SBB *r/m8,r8*	Subtract with borrow *r8* from *r/m8*
19 /*r*	SBB *r/m16,r16*	Subtract with borrow *r16* from *r/m16*
19 /*r*	SBB *r/m32,r32*	Subtract with borrow *r32* from *r/m32*
1A /*r*	SBB *r8,r/m8*	Subtract with borrow *r/m8* from *r8*
1B /*r*	SBB *r16,r/m16*	Subtract with borrow *r/m16* from *r16*
1B /*r*	SBB *r32,r/m32*	Subtract with borrow *r/m32* from *r32*

Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

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

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

Operation

```
DEST ← DEST – (SRC + CF);
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults NaT Register Consumption Abort.

SBB—Integer Subtraction with Borrow (continued)

IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

SCAS/SCASB/SCASW/SCASD—Scan String Data

Opcode	Instruction	Description
AE	SCAS ES:(E)DI	Compare AL with byte at ES:(E)DI and set status flags
AF	SCAS ES:DI	Compare AX with word at ES:DI and set status flags
AF	SCAS ES:EDI	Compare EAX with doubleword at ES:EDI and set status flags
AE	SCASB	Compare AL with byte at ES:(E)DI and set status flags
AF	SCASW	Compare AX with word at ES:DI and set status flags
AF	SCASD	Compare EAX with doubleword at ES:EDI and set status flags

Description

Compares the byte, word, or double word specified with the source operand with the value in the AL, AX, or EAX register, respectively, and sets the status flags in the EFLAGS register according to the results. The source operand specifies the memory location at the address ES:EDI. (When the operand-size attribute is 16, the DI register is used as the source-index register.) The ES segment cannot be overridden with a segment override prefix.

The SCASB, SCASW, and SCASD mnemonics are synonyms of the byte, word, and doubleword versions of the SCAS instructions. They are simpler to use, but provide no type or segment checking. (For the SCAS instruction, “ES:EDI” must be explicitly specified in the instruction.)

After the comparison, the EDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the EDI register is incremented; if the DF flag is 1, the EDI register is decremented.) The EDI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The SCAS, SCASB, SCASW, and SCASD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” on page 5-326 for a description of the REP prefix.

Operation

```
IF (byte cmparison)
  THEN
      temp ← AL – SRC;
      SetStatusFlags(temp);
          THEN IF DF = 0
              THEN (E)DI ← 1;
              ELSE (E)DI ← –1;
          FI;
  ELSE IF (word comparison)
      THEN
          temp ← AX – SRC;
          SetStatusFlags(temp)
              THEN IF DF = 0
```

```
                    THEN DI ← 2;
                    ELSE DI ← –2;
                FI;
        ELSE (* doubleword comparison *)
            temp ← EAX – SRC;
            SetStatusFlags(temp)
                THEN IF DF = 0
                    THEN EDI ← 4;
                    ELSE EDI ← –4;
                FI;
    FI;
FI;
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a null segment selector.
	If an illegal memory operand effective address in the ES segment is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

SETcc—Set Byte on Condition

Opcode	Instruction	Description
0F 97	SETA *r/m8*	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE *r/m8*	Set byte if above or equal (CF=0)
0F 92	SETB *r/m8*	Set byte if below (CF=1)
0F 96	SETBE *r/m8*	Set byte if below or equal (CF=1 or (ZF=1)
0F 92	SETC *r/m8*	Set if carry (CF=1)
0F 94	SETE *r/m8*	Set byte if equal (ZF=1)
0F 9F	SETG *r/m8*	Set byte if greater (ZF=0 and SF=OF)
0F 9D	SETGE *r/m8*	Set byte if greater or equal (SF=OF)
0F 9C	SETL *r/m8*	Set byte if less (SF<>OF)
0F 9E	SETLE *r/m8*	Set byte if less or equal (ZF=1 or SF<>OF)
0F 96	SETNA *r/m8*	Set byte if not above (CF=1 or ZF=1)
0F 92	SETNAE *r/m8*	Set byte if not above or equal (CF=1)
0F 93	SETNB *r/m8*	Set byte if not below (CF=0)
0F 97	SETNBE *r/m8*	Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC *r/m8*	Set byte if not carry (CF=0)
0F 95	SETNE *r/m8*	Set byte if not equal (ZF=0)
0F 9E	SETNG *r/m8*	Set byte if not greater (ZF=1 or SF<>OF)
0F 9C	SETNGE *r/m8*	Set if not greater or equal (SF<>OF)
0F 9D	SETNL *r/m8*	Set byte if not less (SF=OF)
0F 9F	SETNLE *r/m8*	Set byte if not less or equal (ZF=0 and SF=OF)
0F 91	SETNO *r/m8*	Set byte if not overflow (OF=0)
0F 9B	SETNP *r/m8*	Set byte if not parity (PF=0)
0F 99	SETNS *r/m8*	Set byte if not sign (SF=0)
0F 95	SETNZ *r/m8*	Set byte if not zero (ZF=0)
0F 90	SETO *r/m8*	Set byte if overflow (OF=1)
0F 9A	SETP *r/m8*	Set byte if parity (PF=1)
0F 9A	SETPE *r/m8*	Set byte if parity even (PF=1)
0F 9B	SETPO *r/m8*	Set byte if parity odd (PF=0)
0F 98	SETS *r/m8*	Set byte if sign (SF=1)
0F 94	SETZ *r/m8*	Set byte if zero (ZF=1)

Description

Set the destination operand to the value 0 or 1, depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (*cc*) indicates the condition being tested for.

The terms “above” and “below” are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms “greater” and “less” are associated with the SF and OF flags and refer to the relationship between two signed integer values.

SETcc—Set Byte on Condition (continued)

Many of the SET*cc* instruction opcodes have alternate mnemonics. For example, the SETG (set byte if greater) and SETNLE (set if not less or equal) both have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible.

Some languages represent a logical one as an integer with all bits set. This representation can be arrived at by choosing the mutually exclusive condition for the SET*cc* instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

Operation

```
IF condition
  THEN DEST ← 1
  ELSE DEST ← 0;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

SETcc—Set Byte on Condition (continued)

Virtual 8086 Mode Exceptions

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

SGDT/SIDT—Store Global/Interrupt Descriptor Table Register

Opcode	Instruction	Description
0F 01 /0	SGDT *m*	Store GDTR to *m*
0F 01 /1	SIDT *m*	Store IDTR to *m*

Description

Stores the contents of the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR) in the destination operand. The destination operand is a pointer to 6-byte memory location. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the lower 2 bytes of the memory location and the 32-bit base address is stored in the upper 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the lower 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte is filled with 0s.

The SGDT and SIDT instructions are useful only in operating-system software; however, they can be used in application programs.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,SGDT/SIDT);
IF instruction is IDTR
  THEN
      IF OperandSize = 16
          THEN
              DEST[0:15] ← IDTR(Limit);
              DEST[16:39] ← IDTR(Base); (* 24 bits of base address loaded; *)
              DEST[40:47] ← 0;
          ELSE (* 32-bit Operand Size *)
              DEST[0:15] ← IDTR(Limit);
              DEST[16:47] ← IDTR(Base); (* full 32-bit base address loaded *)
      FI;
  ELSE (* instruction is SGDT *)
      IF OperandSize = 16
          THEN
              DEST[0:15] ← GDTR(Limit);
              DEST[16:39] ← GDTR(Base); (* 24 bits of base address loaded; *)
              DEST[40:47] ← 0;
          ELSE (* 32-bit Operand Size *)
              DEST[0:15] ← GDTR(Limit);
              DEST[16:47] ← GDTR(Base); (* full 32-bit base address loaded *)
      FI;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept Instruction Intercept for SIDT and SGDT.

SGDT/SIDT—Store Global/Interrupt Descriptor Table Register (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

Intel Architecture Compatibility

The 16-bit forms of the SGDT and SIDT instructions are compatible with the Intel 286 processor, if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium Pro processor fills these bits with 0s.

SHL/SHR—Shift Instructions

See entry for SAL/SAR/SHL/SHR.

SHLD—Double Precision Shift Left

Opcode	Instruction	Description
0F A4	SHLD *r/m16,r16,imm8*	Shift *r/m16* to left *imm8* places while shifting bits from *r16* in from the right
0F A5	SHLD *r/m16,r16*,CL	Shift *r/m16* to left CL places while shifting bits from *r16* in from the right
0F A4	SHLD *r/m32,r32,imm8*	Shift *r/m32* to left *imm8* places while shifting bits from *r32* in from the right
0F A5	SHLD *r/m32,r32*,CL	Shift *r/m32* to left CL places while shifting bits from *r32* in from the right

Description

Shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand). The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, the flags are not affected.

The SHLD instruction is useful for multi-precision shifts of 64 bits or more.

Operation

```
COUNT ← COUNT MOD 32;
SIZE ← OperandSize
IF COUNT = 0
  THEN
      no operation
  ELSE
      IF COUNT ≥ SIZE
          THEN (* Bad parameters *)
              DEST is undefined;
              CF, OF, SF, ZF, AF, PF are undefined;
          ELSE (* Perform the shift *)
              CF ← BIT[DEST, SIZE – COUNT];
              (* Last bit shifted out on exit *)
              FOR i ← SIZE – 1 DOWNTO COUNT
              DO
                  Bit(DEST, i) ← Bit(DEST, i – COUNT);
              OD;
              FOR i ← COUNT – 1 DOWNTO 0
```

SHLD—Double Precision Shift Left (continued)

```
                DO
                    BIT[DEST, i] ← BIT[SRC, i – COUNT + SIZE];
                OD;
        FI;
FI;
```

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

SHRD—Double Precision Shift Right

Opcode	Instruction	Description
0F AC	SHRD *r/m16,r16,imm8*	Shift *r/m16* to right *imm8* places while shifting bits from *r16* in from the left
0F AD	SHRD *r/m16,r16*,CL	Shift *r/m16* to right CL places while shifting bits from *r16* in from the left
0F AC	SHRD *r/m32,r32,imm8*	Shift *r/m32* to right *imm8* places while shifting bits from *r32* in from the left
0F AD	SHRD *r/m32,r32*,CL	Shift *r/m32* to right CL places while shifting bits from *r32* in from the left

Description

Shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand). The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, the flags are not affected.

The SHRD instruction is useful for multiprecision shifts of 64 bits or more.

Operation

```
COUNT ← COUNT MOD 32;
SIZE ← OperandSize
IF COUNT = 0
  THEN
      no operation
  ELSE
      IF COUNT ≥ SIZE
          THEN (* Bad parameters *)
              DEST is undefined;
              CF, OF, SF, ZF, AF, PF are undefined;
          ELSE (* Perform the shift *)
              CF ← BIT[DEST, COUNT – 1]; (* last bit shifted out on exit *)
              FOR i ← 0 TO SIZE – 1 – COUNT
                  DO
                      BIT[DEST, i] ← BIT[DEST, i – COUNT];
                  OD;
              FOR i ← SIZE – COUNT TO SIZE – 1
                  DO
                      BIT[DEST,i] ← BIT[inBits,i+COUNT – SIZE];
                  OD;
      FI;
FI;
```

SHRD—Double Precision Shift Right (continued)

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

SIDT—Store Interrupt Descriptor Table Register

See entry for SGDT/SIDT.

SLDT—Store Local Descriptor Table Register

Opcode	Instruction	Description
0F 00 /0	SLDT *r/m16*	Stores segment selector from LDTR in *r/m16*
0F 00 /0	SLDT *r/m32*	Store segment selector from LDTR in low-order 16 bits of *r/m32*; high-order 16 bits are undefined

Description

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the LDT.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared to 0s. With the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

The SLDT instruction is only useful in operating-system software; however, it can be used in application programs. Also, this instruction can only be executed in protected mode.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,SLDT);
DEST ← LDTR(SegmentSelector);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	SLDT results in an IA-32 Intercept

Protected Mode Exceptions

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

SLDT—Store Local Descriptor Table Register (continued)

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

SMSW—Store Machine Status Word

Opcode	Instruction	Description
0F 01 /4	SMSW *r32/m16*	Store machine status word in low-order 16 bits of *r32/m16*; high-order 16 bits of *r32* are undefined

Description

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a 16-bit general-purpose register or a memory location.

When the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the upper 16 bits of the register are undefined. With the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

The SMSW instruction is only useful in operating-system software; however, it is not a privileged instruction and can be used in application programs.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on processors more recent than the Intel 286 should use the MOV (control registers) instruction to load the machine status word.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,SMSW);
DEST ← CR0[15:0]; (* MachineStatusWord *);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

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

SMSW—Store Machine Status Word (continued)

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

STC—Set Carry Flag

Opcode	Instruction	Description
F9	STC	Set CF flag

Description

Sets the CF flag in the EFLAGS register.

Operation

CF ← 1;

Flags Affected

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

Exceptions (All Operating Modes)

None.

STD—Set Direction Flag

Opcode	Instruction	Description
FD	STD	Set DF flag

Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI).

Operation

```
DF ← 1;
```

Flags Affected

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

Operation

```
DF ← 1;
```

Exceptions (All Operating Modes)

None.

STI—Set Interrupt Flag

Opcode	Instruction	Description
FB	STI	Set interrupt flag; interrupts enabled at the end of the next instruction

Description

Sets the interrupt flag (IF) in the EFLAGS register. **In the IA-32 System Environment,** after the IF flag is set, the processor begins responding to external maskable interrupts after the next instruction is executed. If the STI instruction is followed by a CLI instruction (which clears the IF flag) the effect of the STI instruction is negated. **In the IA-64 System Environment, the processor will immediately respond do interrupts after STI, unless execution of STI results in a trap or intercept. External interrupts are enabled for IA-32 instructions if PSR.i and (~CFLG.if or EFLAG.if).**

The IF flag and the STI and CLI instruction have no affect on the generation of exceptions and NMI interrupts.

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

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

Notes:
X Don't care.
N Action in Column 1 not taken.
Y Action in Column 1 taken.

Operation

```
OLD_IF <- IF;
IF PE=0  (* Executing in real-address mode *)
  THEN
      IF ← 1;  (* Set Interrupt Flag *)
  ELSE  (* Executing in protected mode or virtual-8086 mode *)
      IF VM=0  (* Executing in protected mode*)
          THEN
              IF CR4.PVI = 0
                  THEN
                      IF CPL <= IOPL
                      THEN IF <- 1
                      ELSE #GP(0);
                      FI;
                  ELSE (*PVI is 1 *)
```

STI—Set Interrupt Flag (continued)

```
                    IF CPL = 3
                    THENSTI—Set Interrupt Flag (continued)

                        IF IOPL < 3
                        THEN
                            IF VIP = 0
                            THEN VIF <- 1;
                            ELSE #GP(0);
                            FI;
                        ELSE (*IOPL = 3 *)
                            IF <- 1;
                        FI;
                    ELSE (*CPL < 3*)
                            IF IOPL < CPL THEN #GP(0); FI;
                            IF IOPL>=CPL OR IOPL=3 THEN IF <-1; FI;
                        FI;
                FI;
            ELSE (*Executing in Virtual-8086 Mode*)
                IF IOPL = 3
                    THEN IF <- 1;
                ELSE
                    IF CR4.VME = 0
                    THEN #GP(0);
                    ELSE
                        IF VIP = 1      (*virtual interrupt is pending*)
                        THEN #GP(0);
                        ELSE VIF <- 1;
                        FI;
                    FI;
                FI;
        FI;
    FI;
FI;
```

IF IA-64 System Environment AND CFLG.ii AND IF != OLD_IF
THEN IA-32_Intercept(System_Flag,STI);

Flags Affected

The IF flag is set to 1.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	System Flag Intercept Trap if CFLG.ii is 1 and the IF flag changes state.

Protected Mode Exceptions

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

STI—Set Interrupt Flag (continued)

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

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

STOS/STOSB/STOSW/STOSD—Store String Data

Opcode	Instruction	Description
AA	STOS ES:(E)DI	Store AL at address ES:(E)DI
AB	STOS ES:DI	Store AX at address ES:DI
AB	STOS ES:EDI	Store EAX at address ES:EDI
AA	STOSB	Store AL at address ES:(E)DI
AB	STOSW	Store AX at address ES:DI
AB	STOSD	Store EAX at address ES:EDI

Description

Stores a byte, word, or doubleword from the AL, AX, or EAX register, respectively, into the destination operand. The destination operand is a memory location at the address ES:EDI. (When the operand-size attribute is 16, the DI register is used as the source-index register.) The ES segment cannot be overridden with a segment override prefix.

The STOSB, STOSW, and STOSD mnemonics are synonyms of the byte, word, and doubleword versions of the STOS instructions. They are simpler to use, but provide no type or segment checking. (For the STOS instruction, “ES:EDI” must be explicitly specified in the instruction.)

After the byte, word, or doubleword is transfer from the AL, AX, or EAX register to the memory location, the EDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the EDI register is incremented; if the DF flag is 1, the EDI register is decremented.) The EDI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The STOS, STOSB, STOSW, and STOSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct, because data needs to be moved into the AL, AX, or EAX register before it can be stored. See “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” on page 5-326 for a description of the REP prefix.

Operation

```
IF (byte store)
  THEN
      DEST ← AL;
          THEN IF DF = 0
              THEN (E)DI ← 1;
              ELSE (E)DI ← –1;
          FI;
  ELSE IF (word store)
      THEN
          DEST ← AX;
              THEN IF DF = 0
                  THEN DI ← 2;
                  ELSE DI ← –2;
              FI;
      ELSE (* doubleword store *)
          DEST ← EAX;
              THEN IF DF = 0
```

```
                    THEN EDI ← 4;
                    ELSE EDI ← –4;

                FI;
    FI;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

STR—Store Task Register

Opcode	Instruction	Description
0F 00 /1	STR *r/m16*	Stores segment selector from TR in *r/m16*

Description

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared to 0s. With the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,STR);
DEST ← TR(SegmentSelector);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

SUB—Integer Subtraction

Opcode	Instruction	Description
2C *ib*	SUB AL,*imm8*	Subtract *imm8* from AL
2D *iw*	SUB AX,*imm16*	Subtract *imm16* from AX
2D *id*	SUB EAX,*imm32*	Subtract *imm32* from EAX
80 /5 *ib*	SUB *r/m8,imm8*	Subtract *imm8* from *r/m8*
81 /5 *iw*	SUB *r/m16,imm16*	Subtract *imm16* from *r/m16*
81 /5 *id*	SUB *r/m32,imm32*	Subtract *imm32* from *r/m32*
83 /5 *ib*	SUB *r/m16,imm8*	Subtract sign-extended *imm8* from *r/m16*
83 /5 *ib*	SUB *r/m32,imm8*	Subtract sign-extended *imm8* from *r/m32*
28 /*r*	SUB *r/m8,r8*	Subtract *r8* from *r/m8*
29 /*r*	SUB *r/m16,r16*	Subtract *r16* from *r/m16*
29 /*r*	SUB *r/m32,r32*	Subtract *r32* from *r/m32*
2A /*r*	SUB *r8,r/m8*	Subtract *r/m8* from *r8*
2B /*r*	SUB *r16,r/m16*	Subtract *r/m16* from *r16*
2B /*r*	SUB *r32,r/m32*	Subtract *r/m32* from *r32*

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

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

Operation

DEST ← DEST – SRC;

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

SUB—Integer Subtraction (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

TEST—Logical Compare

Opcode	Instruction	Description
A8 *ib*	TEST AL,*imm8*	AND *imm8* with AL; set SF, ZF, PF according to result
A9 *iw*	TEST AX,*imm16*	AND *imm16* with AX; set SF, ZF, PF according to result
A9 *id*	TEST EAX,*imm32*	AND *imm32* with EAX; set SF, ZF, PF according to result
F6 /0 *ib*	TEST *r/m8,imm8*	AND *imm8* with *r/m8*; set SF, ZF, PF according to result
F7 /0 *iw*	TEST *r/m16,imm16*	AND *imm16* with *r/m16*; set SF, ZF, PF according to result
F7 /0 *id*	TEST *r/m32,imm32*	AND *imm32* with *r/m32*; set SF, ZF, PF according to result
84 /*r*	TEST *r/m8,r8*	AND *r8* with *r/m8*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m16,r16*	AND *r16* with *r/m16*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m32,r32*	AND *r32* with *r/m32*; set SF, ZF, PF according to result

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

Operation

```
TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);
IF TEMP = 0
   THEN ZF ← 0;
   ELSE ZF ← 1;
FI:
PF ← BitwiseXNOR(TEMP[0:7]);
CF ← 0;
OF ← 0;
(*AF is Undefined*)
```

Flags Affected

The OF and CF flags are cleared to 0. The SF, ZF, and PF flags are set according to the result (see “Operation” above). The state of the AF flag is undefined.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

TEST—Logical Compare (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

UD2—Undefined Instruction

Opcode	Instruction	Description
0F 0B	UD2	Raise invalid opcode exception

Description

Generates an invalid opcode. This instruction is provided for software testing to explicitly generate an invalid opcode. The opcode for this instruction is reserved for this purpose.

Other than raising the invalid opcode exception, this instruction is the same as the NOP instruction.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,0F0B);
#UD (* Generates invalid opcode exception *);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept Mandatory Instruction Intercept.

Exceptions (All Operating Modes)

#UD Instruction is guaranteed to raise an invalid opcode exception in all operating modes).

VERR, VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	Description
0F 00 /4	VERR *r/m16*	Set ZF=1 if segment specified with *r/m16* can be read
0F 00 /5	VERW *r/m16*	Set ZF=1 if segment specified with *r/m16* can be written

Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not null.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable; the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment's DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector's RPL.

The validation performed is the same as if the segment were loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) were performed. The selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

Operation

```
IF SRC(Offset) > (GDTR(Limit) OR (LDTR(Limit))
        THEN
            ZF ← 0
Read segment descriptor;
IF SegmentDescriptor(DescriptorType) = 0 (* system segment *)
   OR (SegmentDescriptor(Type) ≠ conforming code segment)
   AND (CPL > DPL) OR (RPL > DPL)
        THEN
            ZF ← 0
        ELSE
            IF ((Instruction = VERR) AND (segment = readable))
                OR ((Instruction = VERW) AND (segment = writable))
                THEN
                    ZF ← 1;
            FI;
FI;
```

VERR, VERW—Verify a Segment for Reading or Writing (continued)

Flags Affected

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is cleared to 0.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

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

Real Address Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in real address mode.

Virtual 8086 Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in virtual 8086 mode.

WAIT/FWAIT—Wait

Opcode	Instruction	Description
9B	WAIT	Check pending unmasked floating-point exceptions.
9B	FWAIT	Check pending unmasked floating-point exceptions.

Description

Causes the processor to check for and handle pending unmasked floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for the WAIT).

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction insures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results.

Operation

```
CheckPendingUnmaskedFloatingPointExceptions;
```

FPU Flags Affected

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

Floating-point Exceptions

None.

Protected Mode Exceptions

#NM	MP and TS in CR0 is set.

Real Address Mode Exceptions

#NM	MP and TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	MP and TS in CR0 is set.

WBINVD—Write-Back and Invalidate Cache

Opcode	Instruction	Description
0F 09	WBINVD	Write-back and flush Internal caches; initiate writing-back and flushing of external caches.

Description

Writes back all modified cache lines in the processor's internal cache to main memory, invalidates (flushes) the internal caches, and issues a special-function bus cycle that directs external caches to also write back modified data.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction.

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,WBINVD);
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.

Real Address Mode Exceptions

None.

WBINVD—Write-Back and Invalidate Cache (continued)

Virtual 8086 Mode Exceptions

#GP(0)	The WBINVD instruction cannot be executed at the virtual 8086 mode.

Intel Architecture Compatibility

The WDINVD instruction implementation-dependent; its function may be implemented differently on future Intel Architecture processors. The instruction is not supported on Intel Architecture processors earlier than the Intel486 processor.

WRMSR—Write to Model Specific Register

Opcode	Instruction	Description
0F 30	WRMSR	Write the value in EDX:EAX to MSR specified by ECX

Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. The high-order 32 bits are copied from EDX and the low-order 32 bits are copied from EAX. Always set undefined or reserved bits in an MSR to the values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated, including the global entries see the *Intel Architecture Software Developer's Manual, Volume 3*).

The MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. See model specific instructions for all the MSRs that can be written to with this instruction and their addresses.

The WRMSR instruction is a serializing instruction.

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

Operation

```
IF IA-64 System Environment THEN IA-32_Intercept(INST,WRMSR);
MSR[ECX] ← EDX:EAX;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-32_Intercept — Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0) — If the current privilege level is not 0.

If the value in ECX specifies a reserved or unimplemented MSR address.

Real Address Mode Exceptions

#GP — If the current privilege level is not 0

If the value in ECX specifies a reserved or unimplemented MSR address.

WRMSR—Write to Model Specific Register (continued)

Virtual 8086 Mode Exceptions

#GP(0)	The WRMSR instruction is not recognized in virtual 8086 mode.

Intel Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the Intel Architecture with the Pentium processor. Execution of this instruction by an Intel Architecture processor earlier than the Pentium processor results in an invalid opcode exception #UD.

XADD—Exchange and Add

Opcode	Instruction	Description
0F C0/r	XADD *r/m8,r8*	Exchange *r8* and *r/m8*; load sum into *r/m8*.
0F C1/r	XADD *r/m16,r16*	Exchange *r16* and *r/m16*; load sum into *r/m16*.
0F C1/r	XADD *r/m32,r32*	Exchange *r32* and *r/m32*; load sum into *r/m32*.

Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

This instruction can be used with a LOCK prefix.

Operation

```
IF IA-64 System Environment AND External_Bus_Lock_Required AND DCR.lc
   THEN IA-32_Intercept(LOCK,XADD);
TEMP ← SRC + DEST
SRC ← DEST
DEST ← TEMP
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Intercept	Lock Intercept - If an external atomic bus lock is required to complete this operation and DCR.lc is 1, no atomic transaction occurs, this instruction is faulted and an IA-32_Intercept(Lock) fault is generated. The software lock handler is responsible for the emulation of this instruction.

Protected Mode Exceptions

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

XADD—Exchange and Add (continued)

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

Intel Architecture Compatibility

Intel Architecture processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Description
90+*rw*	XCHG AX,*r16*	Exchange *r16* with AX
90+*rw*	XCHG *r16*,AX	Exchange *r16* with AX
90+*rd*	XCHG EAX,*r32*	Exchange *r32* with EAX
90+*rd*	XCHG *r32*,EAX	Exchange *r32* with EAX
86 /*r*	XCHG *r/m8,r8*	Exchange byte register with EA byte
86 /*r*	XCHG *r8,r/m8*	Exchange byte register with EA byte
87 /*r*	XCHG *r/m16,r16*	Exchange *r16* with EA word
87 /*r*	XCHG *r16,r/m16*	Exchange *r16* with EA word
87 /*r*	XCHG *r/m32,r32*	Exchange *r32* with EA doubleword
87 /*r*	XCHG *r32,r/m32*	Exchange *r32* with EA doubleword

Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. When the operands are two registers, one of the registers must be the EAX or AX register. If a memory operand is referenced, the LOCK# signal is automatically asserted for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See Chapter 5, *Processor Management and Initialization*, in the *Intel Architecture Software Developer's Manual, Volume 3* for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

Operation

```
IF IA-64 System Environment AND External_Atomic_Lock_Required AND DCR.lc
   THEN IA-32_Intercept(LOCK,XCHG);

TEMP ← DEST
DEST ← SRC
SRC ← TEMP
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Intercept	Lock Intercept - If an external atomic bus lock is required to complete this operation and DCR.lc is 1, no atomic transaction occurs, this instruction is faulted and an IA-32_Intercept(Lock) fault is generated. The software lock handler is responsible for the emulation of this instruction.

XCHG—Exchange Register/Memory with Register (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Description
D7	XLAT *m8*	Set AL to memory byte DS:[(E)BX + unsigned AL]
D7	XLATB	Set AL to memory byte DS:[(E)BX + unsigned AL]

Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from the DS:EBX registers (or the DS:BX registers when the address-size attribute of 16 bits.) The XLAT instruction allows a different segment register to be specified with a segment override. When assembled, the XLAT and XLATB instructions produce the same machine code.

Operation

```
IF AddressSize = 16
  THEN
       AL ← (DS:BX + ZeroExtend(AL))
  ELSE (* AddressSize = 32 *)
       AL ← (DS:EBX + ZeroExtend(AL));
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

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

XLAT/XLATB—Table Look-up Translation (continued)

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

XOR—Logical Exclusive OR

Opcode	Instruction	Description
34 *ib*	XOR AL,*imm8*	AL XOR *imm8*
35 *iw*	XOR AX,*imm16*	AX XOR *imm16*
35 *id*	XOR EAX,*imm32*	EAX XOR *imm32*
80 /6 *ib*	XOR *r/m8,imm8*	*r/m8* XOR *imm8*
81 /6 *iw*	XOR *r/m16,imm16*	*r/m16* XOR *imm16*
81 /6 *id*	XOR *r/m32,imm32*	*r/m32* XOR *imm32*
83 /6 *ib*	XOR *r/m16,imm8*	*r/m16* XOR *imm8*
83 /6 *ib*	XOR *r/m32,imm8*	*r/m32* XOR *imm8*
30 /*r*	XOR *r/m8,r8*	*r/m8* XOR *r8*
31 /*r*	XOR *r/m16,r16*	*r/m16* XOR *r16*
31 /*r*	XOR *r/m32,r32*	*r/m32* XOR *r32*
32 /*r*	XOR *r8,r/m8*	*r8* XOR *r/m8*
33 /*r*	XOR *r16,r/m16*	*r8* XOR *r/m8*
33 /*r*	XOR *r32,r/m32*	*r8* XOR *r/m8*

Description

Performs a bitwise exclusive-OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location.

Operation

```
DEST ← DEST XOR SRC;
```

Flags Affected

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

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

XOR—Logical Exclusive OR (continued)

Protected Mode Exceptions

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

Real Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

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

IA-32 MMX™ Technology Instruction Reference 6

This section lists the IA-32 MMX technology instructions designed to increase performance of multimedia intensive applications.

EMMS—Empty MMX State

Opcode	Instruction	Description
0F 77	EMMS	Set the FP tag word to empty.

Description

Sets the values of all the tags in the FPU tag word to empty (all ones). This operation marks the MMX technology registers as available, so they can subsequently be used by floating-point instructions. (See Figure 7-11 in the *Intel Architecture Software Developer's Manual, Volume 1*, for the format of the FPU tag word.) All other MMX instructions (other than the EMMS instruction) set all the tags in FPU tag word to valid (all zeros).

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

Operation

```
FPUTagWord ← FFFFH;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.

Protected Mode Exceptions

#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Real-Address Mode Exceptions

#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

MOVD—Move 32 Bits

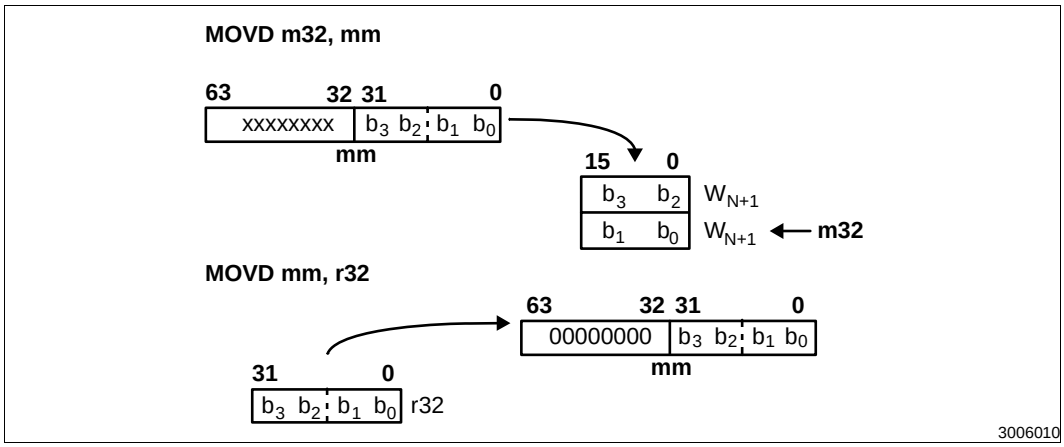
Opcode	Instruction	Description
0F 6E /r	MOVD *mm, r/m32*	Move doubleword from *r/m32* to *mm*.
0F 7E /r	MOVD *r/m32, mm*	Move doubleword from *mm* to *r/m32*.

Description

Copies doubleword from the source operand (second operand) to the destination operand (first operand). Source and destination operands can be MMX technology registers, memory locations, or 32-bit general-purpose registers; however, data cannot be transferred from an MMX technology register to an MMX technology register, from one memory location to another memory location, or from one general-purpose register to another general-purpose register.

When the destination operand is an MMX technology register, the 32-bit source value is written to the low-order 32 bits of the 64-bit MMX technology register and zero-extended to 64 bits (see Figure 6-1). When the source operand is an MMX technology register, the low-order 32 bits of the MMX technology register are written to the 32-bit general-purpose register or 32-bit memory location selected with the destination operand.

Figure 6-1. Operation of the MOVD Instruction



Operation

```
IF DEST is MMX register
  THEN
      DEST ← ZeroExtend(SRC);
  ELSE (* SRC is MMX register *)
      DEST ← LowOrderDoubleword(SRC);
```

MOVD—Move 32 Bits (continued)

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If the destination operand is in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

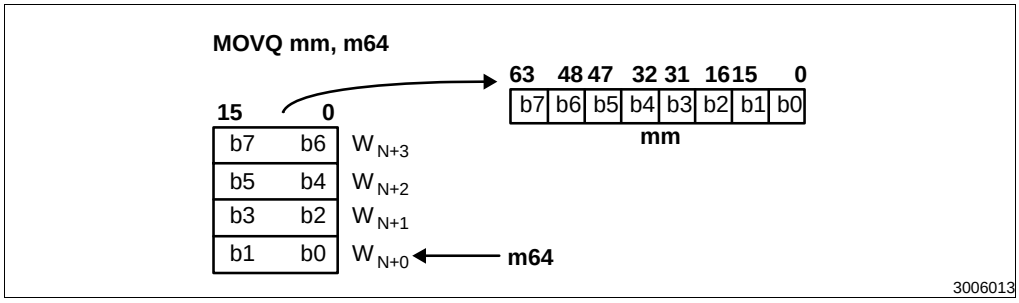
MOVQ—Move 64 Bits

Opcode	Instruction	Description
0F 6F /r	MOVQ *mm, mm/m64*	Move quadword from *mm/m64* to *mm*.
0F 7F /r	MOVQ *mm/m64, mm*	Move quadword from *mm* to *mm/m64*.

Description

Copies quadword from the source operand (second operand) to the destination operand (first operand). (See Figure 6-2.) A source or destination operand can be either an MMX technology register or a memory location; however, data cannot be transferred from one memory location to another memory location. Data can be transferred from one MMX technology register to another MMX technology register.

Figure 6-2. Operation of the MOVQ Instruction



Operation

DEST ← SRC;

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

MOVQ—Move 64 Bits (continued)

Protected Mode Exceptions

#GP(0)	If the destination operand is in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode	Instruction	Description
0F 63 /r	PACKSSWB *mm, mm/m64*	Packs and saturate pack 4 signed words from *mm* and 4 signed words from *mm/m64* into 8 signed bytes in *mm*.
0F 6B /r	PACKSSDW *mm, mm/m64*	Pack and saturate 2 signed doublewords from *mm* and 2 signed doublewords from *mm/m64* into 4 signed words in *mm*.

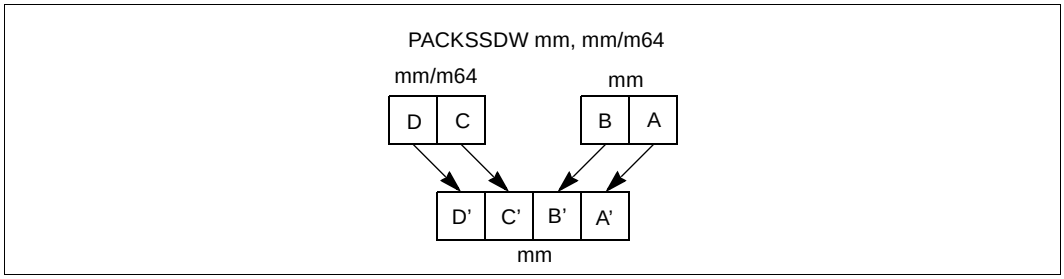
Description

Packs and saturates signed words into bytes (PACKSSWB) or signed doublewords into words (PACKSSDW). The PACKSSWB instruction packs 4 signed words from the destination operand (first operand) and 4 signed words from the source operand (second operand) into 8 signed bytes in the destination operand. If the signed value of a word is beyond the range of a signed byte (that is, greater than 7FH or less than 80H), the saturated byte value of 7FH or 80H, respectively, is stored into the destination.

The PACKSSDW instruction packs 2 signed doublewords from the destination operand (first operand) and 2 signed doublewords from the source operand (second operand) into 4 signed words in the destination operand (see Figure 6-3). If the signed value of a doubleword is beyond the range of a signed word (that is, greater than 7FFFH or less than 8000H), the saturated word value of 7FFFH or 8000H, respectively, is stored into the destination.

The destination operand for either the PACKSSWB or PACKSSDW instruction must be an MMX technology register; the source operand may be either an MMX technology register or a quadword memory location.

Figure 6-3. Operation of the PACKSSDW Instruction



Operation

```
IF instruction is PACKSSWB
  THEN
      DEST(7..0) ← SaturateSignedWordToSignedByte DEST(15..0);
      DEST(15..8) ← SaturateSignedWordToSignedByte DEST(31..16);
      DEST(23..16) ← SaturateSignedWordToSignedByte DEST(47..32);
      DEST(31..24) ← SaturateSignedWordToSignedByte DEST(63..48);
      DEST(39..32) ← SaturateSignedWordToSignedByte SRC(15..0);
      DEST(47..40) ← SaturateSignedWordToSignedByte SRC(31..16);
      DEST(55..48) ← SaturateSignedWordToSignedByte SRC(47..32);
      DEST(63..56) ← SaturateSignedWordToSignedByte SRC(63..48);
```

PACKSSWB/PACKSSDW—Pack with Signed Saturation (continued)

```
    ELSE (* instruction is PACKSSDW *)
        DEST(15..0) ← SaturateSignedDoublewordToSignedWord DEST(31..0);
        DEST(31..16) ← SaturateSignedDoublewordToSignedWord DEST(63..32);
        DEST(47..32) ← SaturateSignedDoublewordToSignedWord SRC(31..0);
        DEST(63..48) ← SaturateSignedDoublewordToSignedWord SRC(63..32);
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

PACKSSWB/PACKSSDW—Pack with Signed Saturation (continued)

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PACKUSWB—Pack with Unsigned Saturation

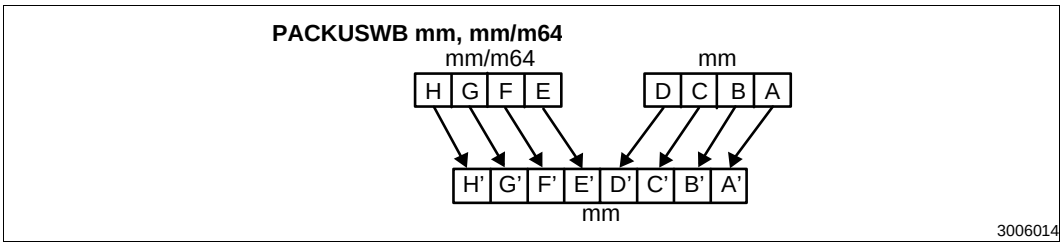
Opcode	Instruction	Description
0F 67 /r	PACKUSWB *mm, mm/m64*	Pack and saturate 4 signed words from *mm* and 4 signed words from *mm/m64* into 8 unsigned bytes in *mm*.

Description

Packs and saturates 4 signed words from the destination operand (first operand) and 4 signed words from the source operand (second operand) into 8 unsigned bytes in the destination operand (see Figure 6-4). If the signed value of a word is beyond the range of an unsigned byte (that is, greater than FFH or less than 00H), the saturated byte value of FFH or 00H, respectively, is stored into the destination.

The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a quadword memory location.

Figure 6-4. Operation of the PACKUSWB Instruction



Operation

```
DEST(7..0) ← SaturateSignedWordToUnsignedByte DEST(15..0);
DEST(15..8) ← SaturateSignedWordToUnsignedByte DEST(31..16);
DEST(23..16) ← SaturateSignedWordToUnsignedByte DEST(47..32);
DEST(31..24) ← SaturateSignedWordToUnsignedByte DEST(63..48);
DEST(39..32) ← SaturateSignedWordToUnsignedByte SRC(15..0);
DEST(47..40) ← SaturateSignedWordToUnsignedByte SRC(31..16);
DEST(55..48) ← SaturateSignedWordToUnsignedByte SRC(47..32);
DEST(63..56) ← SaturateSignedWordToUnsignedByte SRC(63..48);
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PACKUSWB—Pack with Unsigned Saturation (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PADDB/PADDW/PADDD—Packed Add

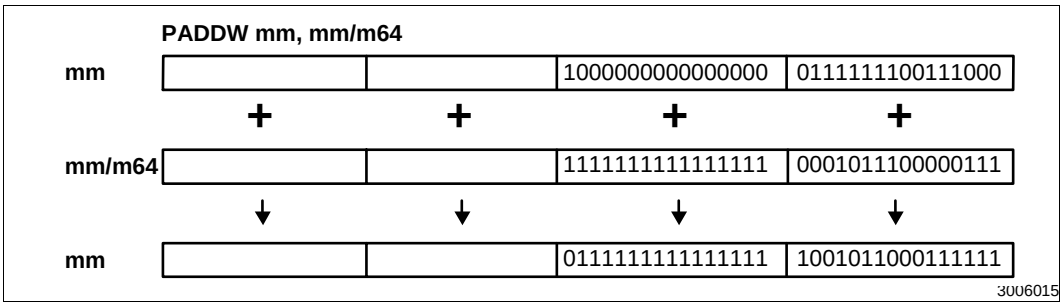
Opcode	Instruction	Description
0F FC /r	PADDB *mm, mm/m64*	Add packed bytes from *mm/m64* to packed bytes in *mm*.
0F FD /r	PADDW *mm, mm/m64*	Add packed words from *mm/m64* to packed words in *mm*.
0F FE /r	PADDD *mm, mm/m64*	Add packed doublewords from *mm/m64* to packed doublewords in *mm*.

Description

Adds the individual data elements (bytes, words, or doublewords) of the source operand (second operand) to the individual data elements of the destination operand (first operand). (See Figure 6-5.) If the result of an individual addition exceeds the range for the specified data type (overflows), the result is wrapped around, meaning that the result is truncated so that only the lower (least significant) bits of the result are returned (that is, the carry is ignored).

The destination operand must be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 6-5. Operation of the PADDW Instruction



The PADDB instruction adds the bytes of the source operand to the bytes of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 8 bits, the lower 8 bits of the result are written to the destination operand and therefore the result wraps around.

The PADDW instruction adds the words of the source operand to the words of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 16 bits, the lower 16 bits of the result are written to the destination operand and therefore the result wraps around.

The PADDD instruction adds the doublewords of the source operand to the doublewords of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 32 bits, the lower 32 bits of the result are written to the destination operand and therefore the result wraps around.

PADDB/PADDW/PADDD—Packed Add (continued)

Note that like the integer ADD instruction, the PADDB, PADDW, and PADDD instructions can operate on either unsigned or signed (two's complement notation) packed integers. Unlike the integer instructions, none of the MMX instructions affect the EFLAGS register. With MMX instructions, there are no carry or overflow flags to indicate when overflow has occurred, so the software must control the range of values or else use the "with saturation" MMX instructions.

Operation

```
IF instruction is PADDB
  THEN
      DEST(7..0) ← DEST(7..0) + SRC(7..0);
      DEST(15..8) ← DEST(15..8) + SRC(15..8);
      DEST(23..16) ← DEST(23..16)+ SRC(23..16);
      DEST(31..24) ← DEST(31..24) + SRC(31..24);
      DEST(39..32) ← DEST(39..32) + SRC(39..32);
      DEST(47..40) ← DEST(47..40)+ SRC(47..40);
      DEST(55..48) ← DEST(55..48) + SRC(55..48);
      DEST(63..56) ← DEST(63..56) + SRC(63..56);
ELSEIF instruction is PADDW
  THEN
      DEST(15..0) ← DEST(15..0) + SRC(15..0);
      DEST(31..16) ← DEST(31..16) + SRC(31..16);
      DEST(47..32) ← DEST(47..32) + SRC(47..32);
      DEST(63..48) ← DEST(63..48) + SRC(63..48);
  ELSE (* instruction is PADDD *)
      DEST(31..0) ← DEST(31..0) + SRC(31..0);
      DEST(63..32) ← DEST(63..32) + SRC(63..32);
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PADDB/PADDW/PADDD—Packed Add (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

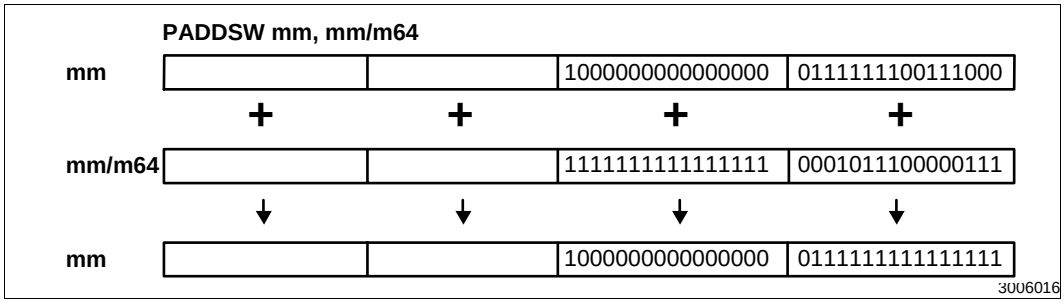
PADDSB/PADDSW—Packed Add with Saturation

Opcode	Instruction	Description
0F EC /r	PADDSB *mm, mm/m64*	Add signed packed bytes from *mm/m64* to signed packed bytes in *mm* and saturate.
0F ED /r	PADDSW *mm, mm/m64*	Add signed packed words from *mm/m64* to signed packed words in *mm* and saturate.

Description

Adds the individual signed data elements (bytes or words) of the source operand (second operand) to the individual signed data elements of the destination operand (first operand). (See Figure 6-6.) If the result of an individual addition exceeds the range for the specified data type, the result is saturated. The destination operand must be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 6-6. Operation of the PADDSW Instruction



The PADDSB instruction adds the signed bytes of the source operand to the signed bytes of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of a signed byte (that is, greater than 7FH or less than 80H), the saturated byte value of 7FH or 80H, respectively, is written to the destination operand.

The PADDSW instruction adds the signed words of the source operand to the signed words of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of a signed word (that is, greater than 7FFFH or less than 8000H), the saturated word value of 7FFFH or 8000H, respectively, is written to the destination operand.

Operation

```
IF instruction is PADDSB
  THEN
      DEST(7..0) ← SaturateToSignedByte(DEST(7..0) + SRC (7..0)) ;
      DEST(15..8) ← SaturateToSignedByte(DEST(15..8) + SRC(15..8) );
      DEST(23..16) ← SaturateToSignedByte(DEST(23..16)+ SRC(23..16) );
      DEST(31..24) ← SaturateToSignedByte(DEST(31..24) + SRC(31..24) );
      DEST(39..32) ← SaturateToSignedByte(DEST(39..32) + SRC(39..32) );
      DEST(47..40) ← SaturateToSignedByte(DEST(47..40)+ SRC(47..40) );
      DEST(55..48) ← SaturateToSignedByte(DEST(55..48) + SRC(55..48) );
      DEST(63..56) ← SaturateToSignedByte(DEST(63..56) + SRC(63..56) );
  ELSE { (* instruction is PADDSW *)
```

PADDSB/PADDSW—Packed Add with Saturation (continued)

```
        DEST(15..0) ← SaturateToSignedWord(DEST(15..0) + SRC(15..0) );

        DEST(31..16) ← SaturateToSignedWord(DEST(31..16) + SRC(31..16) );
        DEST(47..32) ← SaturateToSignedWord(DEST(47..32) + SRC(47..32) );
        DEST(63..48) ← SaturateToSignedWord(DEST(63..48) + SRC(63..48) );
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

PADDSB/PADDSW—Packed Add with Saturation (continued)

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

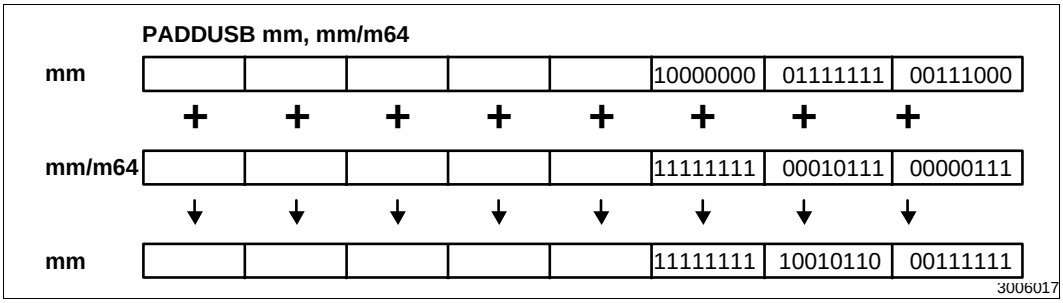
PADDUSB/PADDUSW—Packed Add Unsigned with Saturation

Opcode	Instruction	Description
0F DC /r	PADDUSB *mm, mm/m64*	Add unsigned packed bytes from *mm/m64* to unsigned packed bytes in *mm* and saturate.
0F DD /r	PADDUSW *mm, mm/m64*	Add unsigned packed words from *mm/m64* to unsigned packed words in *mm* and saturate.

Description

Adds the individual unsigned data elements (bytes or words) of the packed source operand (second operand) to the individual unsigned data elements of the packed destination operand (first operand). (See Figure 6-7.) If the result of an individual addition exceeds the range for the specified unsigned data type, the result is saturated. The destination operand must be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 6-7. Operation of the PADDUSB Instruction



The PADDUSB instruction adds the unsigned bytes of the source operand to the unsigned bytes of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of an unsigned byte (that is, greater than FFH), the saturated unsigned byte value of FFH is written to the destination operand.

The PADDUSW instruction adds the unsigned words of the source operand to the unsigned words of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of an unsigned word (that is, greater than FFFFH), the saturated unsigned word value of FFFFH is written to the destination operand.

PADDUSB/PADDUSW—Packed Add Unsigned with Saturation (continued)

Operation

```
IF instruction is PADDUSB
  THEN
      DEST(7..0) ← SaturateToUnsignedByte(DEST(7..0) + SRC (7..0) );
      DEST(15..8) ← SaturateToUnsignedByte(DEST(15..8) + SRC(15..8) );
      DEST(23..16) ← SaturateToUnsignedByte(DEST(23..16)+ SRC(23..16) );
      DEST(31..24) ← SaturateToUnsignedByte(DEST(31..24) + SRC(31..24) );
      DEST(39..32) ← SaturateToUnsignedByte(DEST(39..32) + SRC(39..32) );
      DEST(47..40) ← SaturateToUnsignedByte(DEST(47..40)+ SRC(47..40) );
      DEST(55..48) ← SaturateToUnsignedByte(DEST(55..48) + SRC(55..48) );
      DEST(63..56) ← SaturateToUnsignedByte(DEST(63..56) + SRC(63..56) );
  ELSE { (* instruction is PADDUSW *)
      DEST(15..0) ← SaturateToUnsignedWord(DEST(15..0) + SRC(15..0) );
      DEST(31..16) ← SaturateToUnsignedWord(DEST(31..16) + SRC(31..16) );
      DEST(47..32) ← SaturateToUnsignedWord(DEST(47..32) + SRC(47..32) );
      DEST(63..48) ← SaturateToUnsignedWord(DEST(63..48) + SRC(63..48) );
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

PADDUSB/PADDUSW—Packed Add Unsigned with Saturation (continued)

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

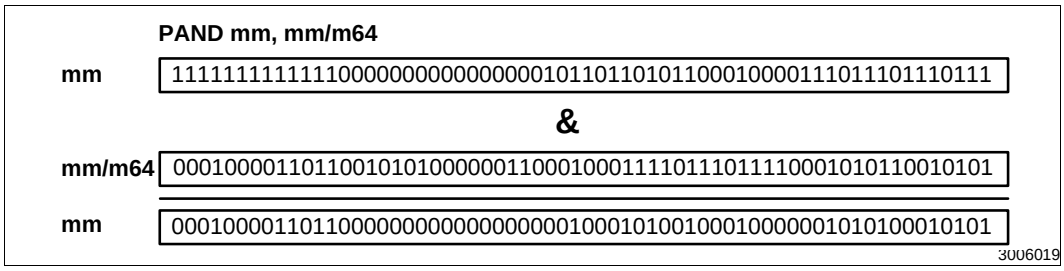
PAND—Logical AND

Opcode	Instruction	Description
0F DB /r	PAND *mm, mm/m64*	AND quadword from *mm/m64* to quadword in mm.

Description

Performs a bitwise logical AND operation on the quadword source (second) and destination (first) operands and stores the result in the destination operand location (see Figure 6-8). The source operand can be an MMX technology register or a quadword memory location; the destination operand must be an MMX technology register. Each bit of the result of the PAND instruction is set to 1 if the corresponding bits of the operands are both 1; otherwise it is made zero

Figure 6-8. Operation of the PAND Instruction



Operation

```
DEST ← DEST AND SRC;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PAND—Logical AND (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PANDN—Logical AND NOT

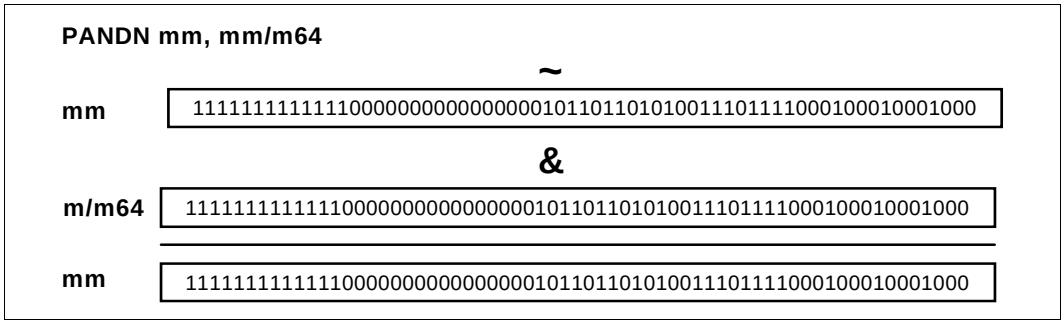
Opcode	Instruction	Description
0F DF /r	PANDN *mm, mm/m64*	AND quadword from *mm/m64* to NOT quadword in *mm*.

Description

Performs a bitwise logical NOT on the quadword destination operand (first operand). Then, the instruction performs a bitwise logical AND operation on the inverted destination operand and the quadword source operand (second operand). (See Figure 6-9.) Each bit of the result of the AND operation is set to one if the corresponding bits of the source and inverted destination bits are one; otherwise it is set to zero. The result is stored in the destination operand location.

The source operand can be an MMX technology register or a quadword memory location; the destination operand must be an MMX technology register.

Figure 6-9. Operation of the PANDN Instruction



Operation

```
DEST ← (NOT DEST) AND SRC;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PANDN—Logical AND NOT (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

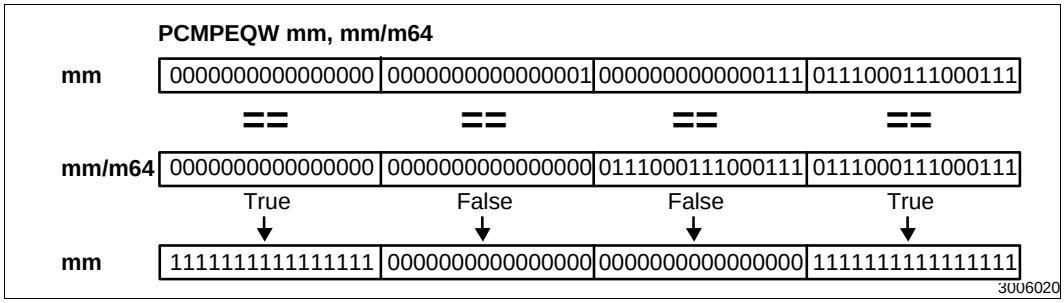
PCMPEQB/PCMPEQW/PCMPEQD—Packed Compare for Equal

Opcode	Instruction	Description
0F 74 /r	PCMPEQB *mm, mm/m64*	Compare packed bytes in *mm/m64* with packed bytes in *mm* for equality.
0F 75 /r	PCMPEQW *mm, mm/m64*	Compare packed words in *mm/m64* with packed words in *mm* for equality.
0F 76 /r	PCMPEQD *mm, mm/m64*	Compare packed doublewords in *mm/m64* with packed doublewords in *mm* for equality.

Description

Compares the individual data elements (bytes, words, or doublewords) in the destination operand (first operand) to the corresponding data elements in the source operand (second operand). (See Figure 6-10.) If a pair of data elements are equal, the corresponding data element in the destination operand is set to all ones; otherwise, it is set to all zeros. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location.

Figure 6-10. Operation of the PCMPEQW Instruction



The PCMPEQB instruction compares the bytes in the destination operand to the corresponding bytes in the source operand, with the bytes in the destination operand being set according to the results.

The PCMPEQW instruction compares the words in the destination operand to the corresponding words in the source operand, with the words in the destination operand being set according to the results.

The PCMPEQD instruction compares the doublewords in the destination operand to the corresponding doublewords in the source operand, with the doublewords in the destination operand being set according to the results.

PCMPEQB/PCMPEQW/PCMPEQD—Packed Compare for Equal (continued)

Operation

```
IF instruction is PCMPEQB
  THEN
      IF DEST(7..0) = SRC(7..0)
          THEN DEST(7  0) ← FFH;
          ELSE DEST(7..0) ← 0;
      * Continue comparison of second through seventh bytes in DEST and SRC *
      IF DEST(63..56) = SRC(63..56)
          THEN DEST(63..56) ← FFH;
          ELSE DEST(63..56) ← 0;
ELSE IF instruction is PCMPEQW
  THEN
      IF DEST(15..0) = SRC(15..0)
          THEN DEST(15..0) ← FFFFH;
          ELSE DEST(15..0) ← 0;
      * Continue comparison of second and third words in DEST and SRC *
      IF DEST(63..48) = SRC(63..48)
          THEN DEST(63..48) ← FFFFH;
          ELSE DEST(63..48) ← 0;
  ELSE (* instruction is PCMPEQD *)
      IF DEST(31..0) = SRC(31..0)
          THEN DEST(31..0) ← FFFFFFFFH;
          ELSE DEST(31..0) ← 0;
      IF DEST(63..32) = SRC(63..32)
          THEN DEST(63..32) ← FFFFFFFFH;
          ELSE DEST(63..32) ← 0;
FI;
```

Flags Affected

None:

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.

PCMPEQB/PCMPEQW/PCMPEQD—Packed Compare for Equal (continued)

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

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

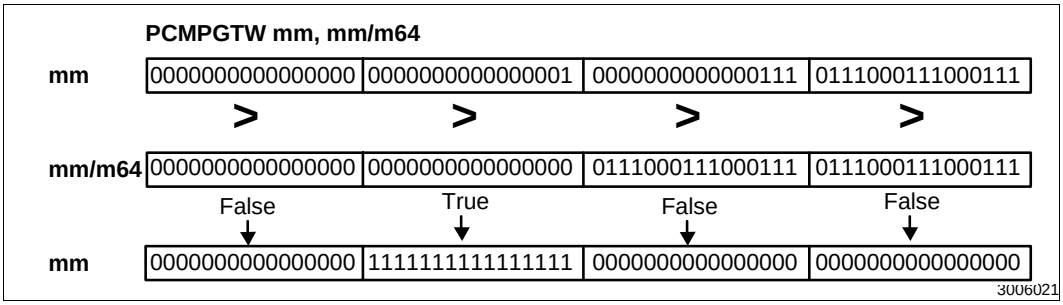
PCMPGTB/PCMPGTW/PCMPGTD—Packed Compare for Greater Than

Opcode	Instruction	Description
0F 64 /r	PCMPGTB *mm, mm/m64*	Compare packed bytes in *mm* with packed bytes in *mm/m64* for greater value.
0F 65 /r	PCMPGTW *mm, mm/m64*	Compare packed words in *mm* with packed words in *mm/m64* for greater value.
0F 66 /r	PCMPGTD *mm, mm/m64*	Compare packed doublewords in *mm* with packed doublewords in *mm/m64* for greater value.

Description

Compare the individual signed data elements (bytes, words, or doublewords) in the destination operand (first operand) to the corresponding signed data elements in the source operand (second operand). (See Figure 6-11.) If a data element in the destination operand is greater than its corresponding data element in the source operand, the data element in the destination operand is set to all ones; otherwise, it is set to all zeros. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location.

Figure 6-11. Operation of the PCMPGTW Instruction



The PCMPGTB instruction compares the signed bytes in the destination operand to the corresponding signed bytes in the source operand, with the bytes in the destination operand being set according to the results.

The PCMPGTW instruction compares the signed words in the destination operand to the corresponding signed words in the source operand, with the words in the destination operand being set according to the results.

The PCMPGTD instruction compares the signed doublewords in the destination operand to the corresponding signed doublewords in the source operand, with the doublewords in the destination operand being set according to the results.

PCMPGTB/PCMPGTW/PCMPGTD—Packed Compare for Greater Than (continued)

Operation

```
IF instruction is PCMPGTB
  THEN
      IF DEST(7..0) > SRC(7..0)
          THEN DEST(7  0) ← FFH;
          ELSE DEST(7..0) ← 0;
      * Continue comparison of second through seventh bytes in DEST and SRC *
      IF DEST(63..56) > SRC(63..56)
          THEN DEST(63..56) ← FFH;
          ELSE DEST(63..56) ← 0;
ELSE IF instruction is PCMPGTW
  THEN
      IF DEST(15..0) > SRC(15..0)
          THEN DEST(15..0) ← FFFFH;
          ELSE DEST(15..0) ←0;
      * Continue comparison of second and third bytes in DEST and SRC *
      IF DEST(63..48) > SRC(63..48)
          THEN DEST(63..48) ← FFFFH;
          ELSE DEST(63..48) ← 0;
  ELSE { (* instruction is PCMPGTD *)
      IF DEST(31..0) > SRC(31..0)
          THEN DEST(31..0) ← FFFFFFFFH;
          ELSE DEST(31..0) ← 0;
      IF DEST(63..32) > SRC(63..32)
          THEN DEST(63..32) ← FFFFFFFFH;
          ELSE DEST(63..32) ← 0;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PCMPGTB/PCMPGTW/PCMPGTD—Packed Compare for Greater Than (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PMADDWD—Packed Multiply and Add

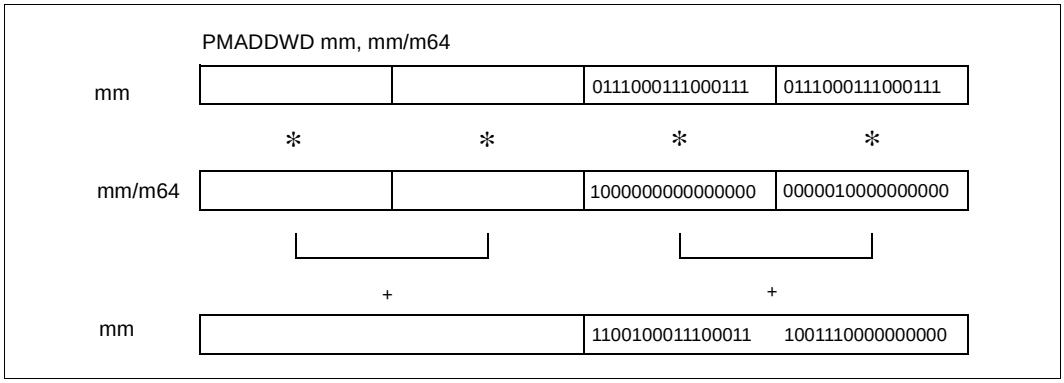
Opcode	Instruction	Description
0F F5 /r	PMADDWD *mm, mm/m64*	Multiply the packed words in *mm* by the packed words in *mm/m64*. Add the 32-bit pairs of results and store in *mm* as doubleword

Description

Multiplies the individual signed words of the destination operand by the corresponding signed words of the source operand, producing four signed, doubleword results (see Figure 6-12). The two doubleword results from the multiplication of the high-order words are added together and stored in the upper doubleword of the destination operand; the two doubleword results from the multiplication of the low-order words are added together and stored in the lower doubleword of the destination operand. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location.

The PMADDWD instruction wraps around to 80000000H only when all four words of both the source and destination operands are 8000H.

Figure 6-12. Operation of the PMADDWD Instruction



Operation

```
DEST(31..0) ← (DEST(15..0) * SRC(15..0)) + (DEST(31..16) * SRC(31..16));
DEST(63..32) ← (DEST(47..32) * SRC(47..32)) + (DEST(63..48) * SRC(63..48));
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PMADDWD—Packed Multiply and Add (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

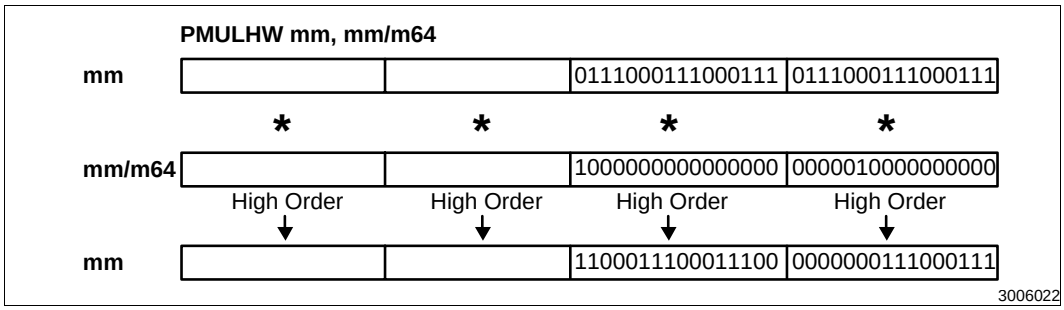
PMULHW—Packed Multiply High

Opcode	Instruction	Description
0F E5 /r	PMULHW *mm, mm/m64*	Multiply the signed packed words in *mm* by the signed packed words in *mm/m64*, then store the high-order word of each doubleword result in *mm*.

Description

Multiplies the four signed words of the source operand (second operand) by the four signed words of the destination operand (first operand), producing four signed, doubleword, intermediate results (see Figure 6-13). The high-order word of each intermediate result is then written to its corresponding word location in the destination operand. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location.

Figure 6-13. Operation of the PMULHW Instruction



Operation

```
DEST(15..0) ← HighOrderWord(DEST(15..0) * SRC(15..0));
DEST(31..16) ← HighOrderWord(DEST(31..16) * SRC(31..16));
DEST(47..32) ← HighOrderWord(DEST(47..32) * SRC(47..32));
DEST(63..48) ← HighOrderWord(DEST(63..48) * SRC(63..48));
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PMULHW—Packed Multiply High (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

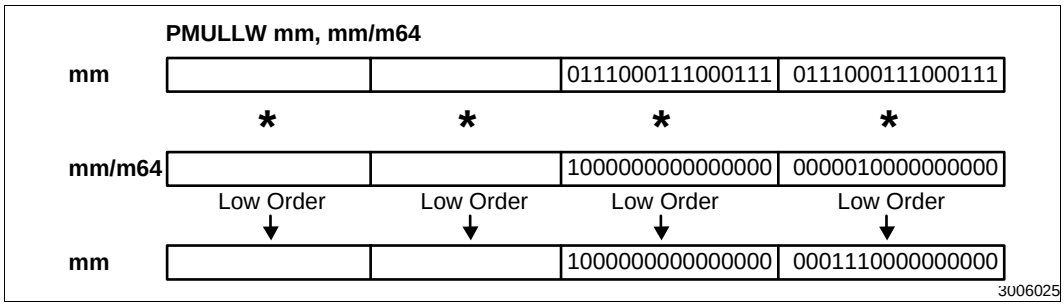
PMULLW—Packed Multiply Low

Opcode	Instruction	Description
0F D5 /r	PMULLW *mm, mm/m64*	Multiply the packed words in *mm* with the packed words in *mm/m64*, then store the low-order word of each doubleword result in *mm*.

Description

Multiplies the four signed or unsigned words of the source operand (second operand) with the four signed or unsigned words of the destination operand (first operand), producing four doubleword, intermediate results (see Figure 6-14). The low-order word of each intermediate result is then written to its corresponding word location in the destination operand. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location.

Figure 6-14. Operation of the PMULLW Instruction



Operation

```
DEST(15..0) ← LowOrderWord(DEST(15..0) * SRC(15..0));
DEST(31..16) ← LowOrderWord(DEST(31..16) * SRC(31..16));
DEST(47..32) ← LowOrderWord(DEST(47..32) * SRC(47..32));
DEST(63..48) ← LowOrderWord(DEST(63..48) * SRC(63..48));
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PMULLW—Packed Multiply Low (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

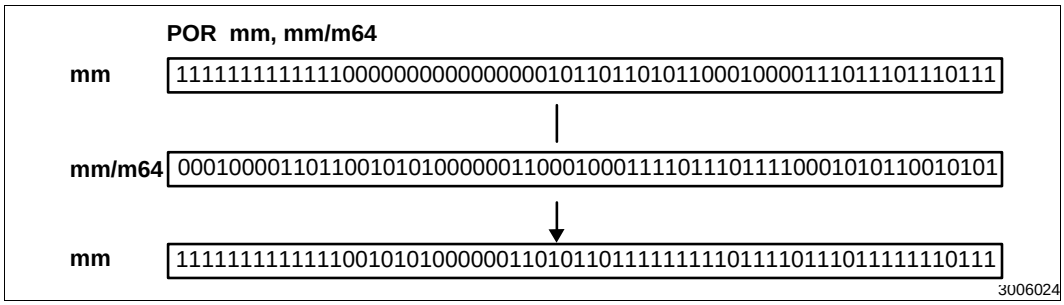
POR—Bitwise Logical OR

Opcode	Instruction	Description
0F EB /r	POR *mm, mm/m64*	OR quadword from mm/m64 to quadword in mm.

Description

Performs a bitwise logical OR operation on the quadword source (second) and destination (first) operands and stores the result in the destination operand location (see Figure 6-15). The source operand can be an MMX technology register or a quadword memory location; the destination operand must be an MMX technology register. Each bit of the result is made 0 if the corresponding bits of both operands are 0; otherwise the bit is set to 1.

Figure 6-15. Operation of the POR Instruction.



Operation

DEST ← DEST OR SRC;

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

POR—Bitwise Logical OR (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PSLLW/PSLLD/PSLLQ—Packed Shift Left Logical

Opcode	Instruction	Description
0F F1 /r	PSLLW *mm, mm/m64*	Shift words in *mm* left by amount specified in *mm/m64*, while shifting in zeros.
0F 71 /6, ib	PSLLW *mm, imm8*	Shift words in *mm* left by *imm8*, while shifting in zeros.
0F F2 /r	PSLLD *mm, mm/m64*	Shift doublewords in *mm* left by amount specified in *mm/m64*, while shifting in zeros.
0F 72 /6 ib	PSLLD *mm, imm8*	Shift doublewords in *mm* by *imm8*, while shifting in zeros.
0F F3 /r	PSLLQ *mm, mm/m64*	Shift *mm* left by amount specified in *mm/m64*, while shifting in zeros.
0F 73 /6 ib	PSLLQ *mm, imm8*	Shift *mm* left by Imm8, while shifting in zeros.

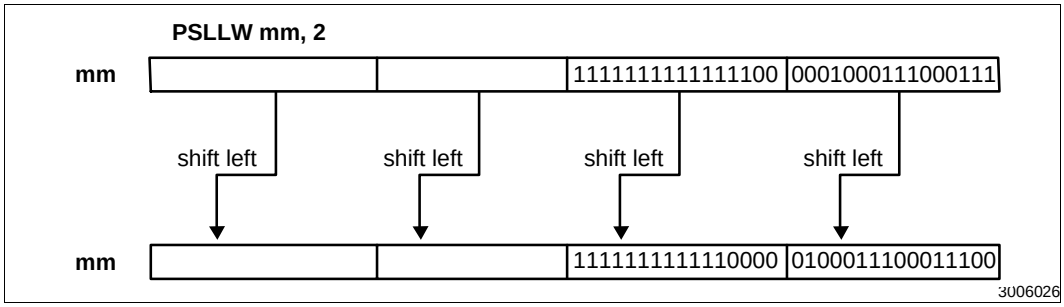
Description

Shifts the bits in the data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the unsigned count operand (second operand). (See Figure 6-16.) The result of the shift operation is written to the destination operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to zero). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all zeros.

The destination operand must be an MMX technology register; the count operand can be either an MMX technology register, a 64-bit memory location, or an 8-bit immediate.

The PSLLW instruction shifts each of the four words of the destination operand to the left by the number of bits specified in the count operand; the PSLLD instruction shifts each of the two doublewords of the destination operand; and the PSLLQ instruction shifts the 64-bit quadword in the destination operand. As the individual data elements are shifted left, the empty low-order bit positions are filled with zeros.

Figure 6-16. Operation of the PSLLW Instruction



PSLLW/PSLLD/PSLLQ—Packed Shift Left Logical (continued)

Operation

```
IF instruction is PSLLW
  THEN
       DEST(15..0) ← DEST(15..0) << COUNT;
       DEST(31..16) ← DEST(31..16) << COUNT;
       DEST(47..32) ← DEST(47..32) << COUNT;
       DEST(63..48) ← DEST(63..48) << COUNT;
  ELSE IF instruction is PSLLD
       THEN {
            DEST(31..0) ← DEST(31..0) << COUNT;
            DEST(63..32) ← DEST(63..32) << COUNT;
       ELSE  (* instruction is PSLLQ *)
            DEST ← DEST << COUNT;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

PSLLW/PSLLD/PSLLQ—Packed Shift Left Logical (continued)

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PSRAW/PSRAD—Packed Shift Right Arithmetic

Opcode	Instruction	Description
0F E1 /r	PSRAW *mm, mm/m64*	Shift words in *mm* right by amount specified in *mm/m64* while shifting in sign bits.
0F 71 /4 ib	PSRAW *mm, imm8*	Shift words in *mm* right by *imm8* while shifting in sign bits
0F E2 /r	PSRAD *mm, mm/m64*	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in sign bits.
0F 72 /4 ib	PSRAD *mm, imm8*	Shift doublewords in *mm* right by *imm8* while shifting in sign bits.

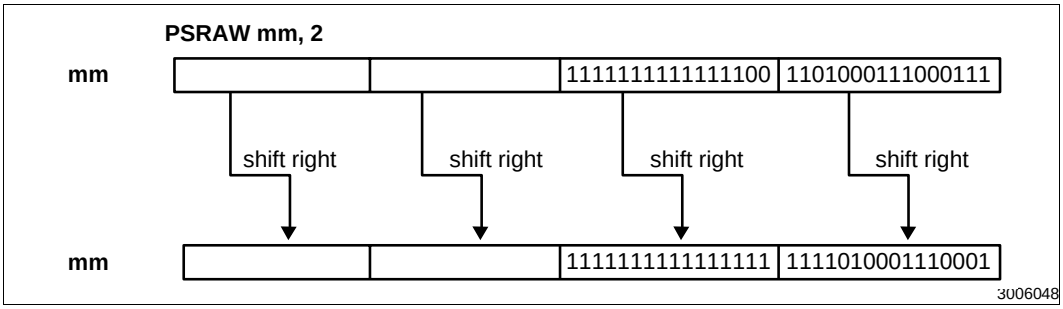
Description

Shifts the bits in the data elements (words or doublewords) in the destination operand (first operand) to the right by the amount of bits specified in the unsigned count operand (second operand). (See Figure 6-17.) The result of the shift operation is written to the destination operand. The empty high-order bits of each element are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words) or 31 (for doublewords), each destination data element is filled with the initial value of the sign bit of the element.

The destination operand must be an MMX technology register; the count operand (source operand) can be either an MMX technology register, a 64-bit memory location, or an 8-bit immediate.

The PSRAW instruction shifts each of the four words in the destination operand to the right by the number of bits specified in the count operand; the PSRAD instruction shifts each of the two doublewords in the destination operand. As the individual data elements are shifted right, the empty high-order bit positions are filled with the sign value.

Figure 6-17. Operation of the PSRAW Instruction



PSRAW/PSRAD—Packed Shift Right Arithmetic (continued)

Operation

```
IF instruction is PSRAW
  THEN
      DEST(15..0) ← SignExtend (DEST(15..0) >> COUNT);
      DEST(31..16) ← SignExtend (DEST(31..16) >> COUNT);
      DEST(47..32) ← SignExtend (DEST(47..32) >> COUNT);
      DEST(63..48) ← SignExtend (DEST(63..48) >> COUNT);
  ELSE { (*instruction is PSRAD *)
      DEST(31..0) ← SignExtend (DEST(31..0) >> COUNT);
      DEST(63..32) ← SignExtend (DEST(63..32) >> COUNT);
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

PSRAW/PSRAD—Packed Shift Right Arithmetic (continued)

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PSRLW/PSRLD/PSRLQ—Packed Shift Right Logical

Opcode	Instruction	Description
0F D1 /r	PSRLW *mm, mm/m64*	Shift words in *mm* right by amount specified in *mm/m64* while shifting in zeros.
0F 71 /2 ib	PSRLW *mm, imm8*	Shift words in *mm* right by *imm8*.
0F D2 /r	PSRLD *mm, mm/m64*	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in zeros.
0F 72 /2 ib	PSRLD *mm, imm8*	Shift doublewords in *mm* right by *imm8*.
0F D3 /r	PSRLQ *mm, mm/m64*	Shift *mm* right by amount specified in *mm/m64* while shifting in zeros.
0F 73 /2 ib	PSRLQ *mm, imm8*	Shift *mm* right by *imm8* while shifting in zeros.

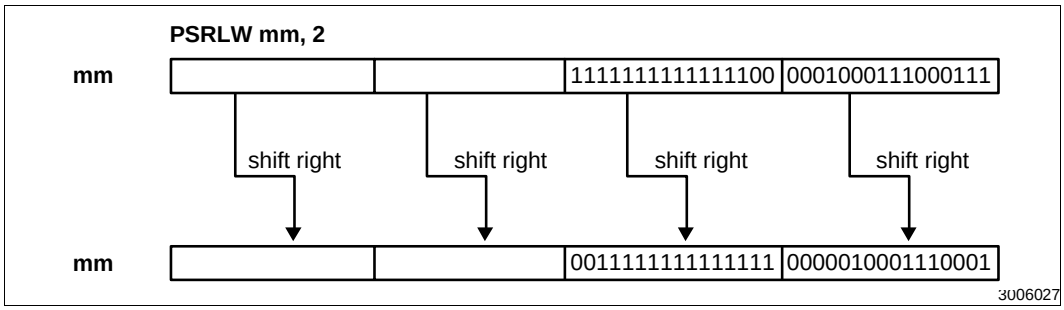
Description

Shifts the bits in the data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the unsigned count operand (second operand). (See Figure 6-18.) The result of the shift operation is written to the destination operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to zero). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all zeros.

The destination operand must be an MMX technology register; the count operand can be either an MMX technology register, a 64-bit memory location, or an 8-bit immediate.

The PSRLW instruction shifts each of the four words of the destination operand to the right by the number of bits specified in the count operand; the PSRLD instruction shifts each of the two doublewords of the destination operand; and the PSRLQ instruction shifts the 64-bit quadword in the destination operand. As the individual data elements are shifted right, the empty high-order bit positions are filled with zeros.

Figure 6-18. Operation of the PSRLW Instruction



PSRLW/PSRLD/PSRLQ—Packed Shift Right Logical (continued)

Operation

```
IF instruction is PSRLW
  THEN {
       DEST(15..0) ← DEST(15..0) >> COUNT;
       DEST(31..16) ← DEST(31..16) >> COUNT;
       DEST(47..32) ← DEST(47..32) >> COUNT;
       DEST(63..48) ← DEST(63..48) >> COUNT;
ELSE IF instruction is PSRLD
  THEN {
       DEST(31..0) ← DEST(31..0) >> COUNT;
       DEST(63..32) ← DEST(63..32) >> COUNT;
  ELSE  (* instruction is PSRLQ *)
       DEST ← DEST >> COUNT;
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

PSRLW/PSRLD/PSRLQ—Packed Shift Right Logical (continued)

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PSUBB/PSUBW/PSUBD—Packed Subtract

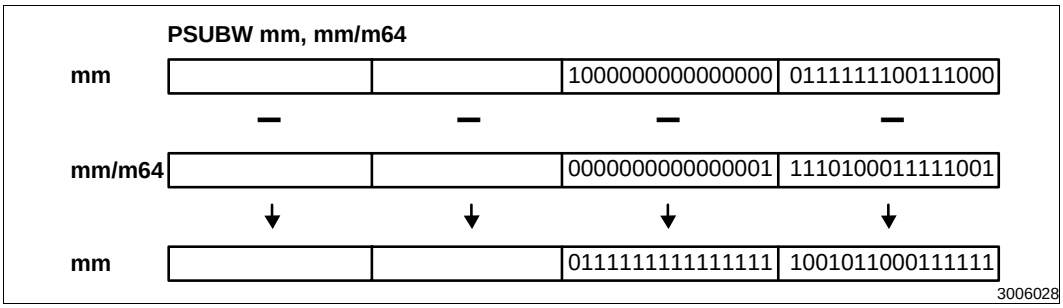
Opcode	Instruction	Description
0F F8 /r	PSUBB *mm, mm/m64*	Subtract packed bytes in *mm/m64* from packed bytes in *mm.*
0F F9 /r	PSUBW *mm, mm/m64*	Subtract packed words in*mm/m64* from packed words in *mm.*
0F FA /r	PSUBD *mm, mm/m64*	Subtract packed doublewords in *mm/m64* from packed doublewords in *mm.*

Description

Subtracts the individual data elements (bytes, words, or doublewords) of the source operand (second operand) from the individual data elements of the destination operand (first operand). (See Figure 6-19.) If the result of a subtraction exceeds the range for the specified data type (overflows), the result is wrapped around, meaning that the result is truncated so that only the lower (least significant) bits of the result are returned (that is, the carry is ignored).

The destination operand must be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 6-19. Operation of the PSUBW Instruction



The PSUBB instruction subtracts the bytes of the source operand from the bytes of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 8 bits, the lower 8 bits of the result are written to the destination operand and therefore the result wraps around.

The PSUBW instruction subtracts the words of the source operand from the words of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 16 bits, the lower 16 bits of the result are written to the destination operand and therefore the result wraps around.

The PSUBD instruction subtracts the doublewords of the source operand from the doublewords of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 32 bits, the lower 32 bits of the result are written to the destination operand and therefore the result wraps around.

PSUBB/PSUBW/PSUBD—Packed Subtract (continued)

Note that like the integer SUB instruction, the PSUBB, PSUBW, and PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers. Unlike the integer instructions, none of the MMX instructions affect the EFLAGS register. With MMX instructions, there are no carry or overflow flags to indicate when overflow has occurred, so the software must control the range of values or else use the "with saturation" MMX instructions.

Operation

```
IF instruction is PSUBB
  THEN
      DEST(7..0) ← DEST(7..0) – SRC(7..0);
      DEST(15..8) ← DEST(15..8) – SRC(15..8);
      DEST(23..16) ← DEST(23..16) – SRC(23..16);
      DEST(31..24) ← DEST(31..24) – SRC(31..24);
      DEST(39..32) ← DEST(39..32) – SRC(39..32);
      DEST(47..40) ← DEST(47..40) – SRC(47..40);
      DEST(55..48) ← DEST(55..48) – SRC(55..48);
      DEST(63..56) ← DEST(63..56) – SRC(63..56);
ELSEIF instruction is PSUBW
  THEN
      DEST(15..0) ← DEST(15..0) – SRC(15..0);
      DEST(31..16) ← DEST(31..16) – SRC(31..16);
      DEST(47..32) ← DEST(47..32) – SRC(47..32);
      DEST(63..48) ← DEST(63..48) – SRC(63..48);
  ELSE { (* instruction is PSUBD *)
      DEST(31..0) ← DEST(31..0) – SRC(31..0);
      DEST(63..32) ← DEST(63..32) – SRC(63..32);
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PSUBB/PSUBW/PSUBD—Packed Subtract (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

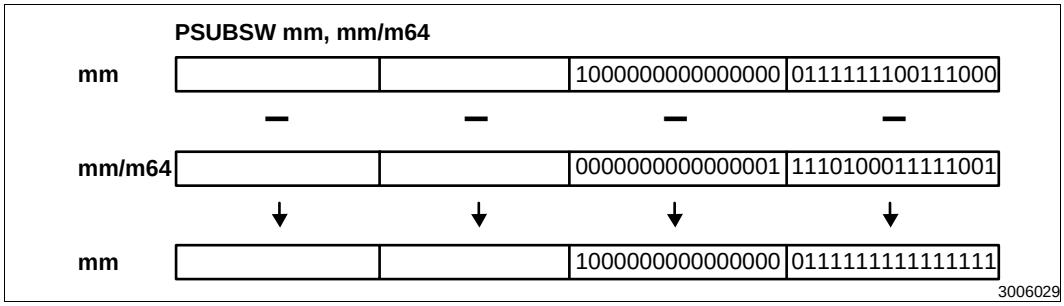
PSUBSB/PSUBSW—Packed Subtract with Saturation

Opcode	Instruction	Description
0F E8 /r	PSUBSB *mm, mm/m64*	Subtract signed packed bytes in *mm/m64* from signed packed bytes in *mm* and saturate.
0F E9 /r	PSUBSW *mm, mm/m64*	Subtract signed packed words in *mm/m64* from signed packed words in *mm* and saturate.

Description

Subtracts the individual signed data elements (bytes or words) of the source operand (second operand) from the individual signed data elements of the destination operand (first operand). (See Figure 6-20.) If the result of a subtraction exceeds the range for the specified data type, the result is saturated. The destination operand must be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 6-20. Operation of the PSUBSW Instruction



The PSUBSB instruction subtracts the signed bytes of the source operand from the signed bytes of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of a signed byte (that is, greater than 7FH or less than 80H), the saturated byte value of 7FH or 80H, respectively, is written to the destination operand.

The PSUBSW instruction subtracts the signed words of the source operand from the signed words of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of a signed word (that is, greater than 7FFFH or less than 8000H), the saturated word value of 7FFFH or 8000H, respectively, is written to the destination operand.

PSUBSB/PSUBSW—Packed Subtract with Saturation (continued)

Operation

```
IF instruction is PSUBSB
  THEN
      DEST(7..0) ← SaturateToSignedByte(DEST(7..0) – SRC (7..0));
      DEST(15..8) ← SaturateToSignedByte(DEST(15..8) – SRC(15..8));
      DEST(23..16) ← SaturateToSignedByte(DEST(23..16) – SRC(23..16));
      DEST(31..24) ← SaturateToSignedByte(DEST(31..24) – SRC(31..24));
      DEST(39..32) ← SaturateToSignedByte(DEST(39..32) – SRC(39..32));
      DEST(47..40) ← SaturateToSignedByte(DEST(47..40) – SRC(47..40));
      DEST(55..48) ← SaturateToSignedByte(DEST(55..48) – SRC(55..48));
      DEST(63..56) ← SaturateToSignedByte(DEST(63..56) – SRC(63..56))
  ELSE (* instruction is PSUBSW *)
      DEST(15..0) ← SaturateToSignedWord(DEST(15..0) – SRC(15..0));
      DEST(31..16) ← SaturateToSignedWord(DEST(31..16) – SRC(31..16));
      DEST(47..32) ← SaturateToSignedWord(DEST(47..32) – SRC(47..32));
      DEST(63..48) ← SaturateToSignedWord(DEST(63..48) – SRC(63..48));
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

PSUBSB/PSUBSW—Packed Subtract with Saturation (continued)

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

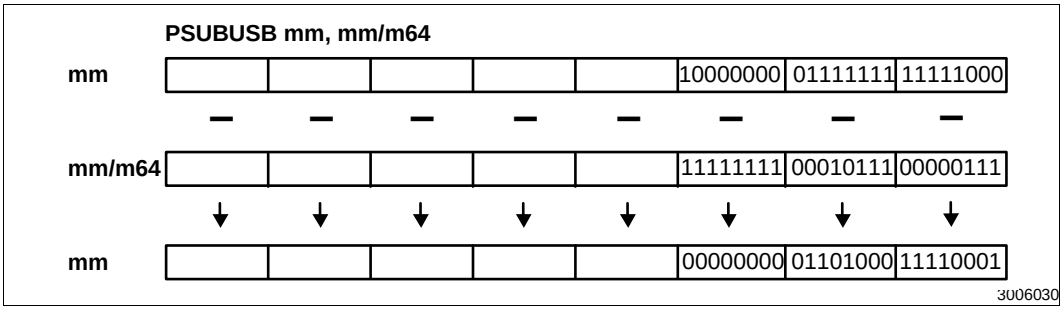
PSUBUSB/PSUBUSW—Packed Subtract Unsigned with Saturation

Opcode	Instruction	Description
0F D8 /r	PSUBUSB *mm, mm/m64*	Subtract unsigned packed bytes in *mm/m64* from unsigned packed bytes in *mm* and saturate.
0F D9 /r	PSUBUSW *mm, mm/m64*	Subtract unsigned packed words in *mm/m64* from unsigned packed words in *mm* and saturate.

Description

Subtracts the individual unsigned data elements (bytes or words) of the source operand (second operand) from the individual unsigned data elements of the destination operand (first operand). (See Figure 6-21.) If the result of an individual subtraction exceeds the range for the specified unsigned data type, the result is saturated. The destination operand musts be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 6-21. Operation of the PSUBUSB Instruction



The PSUBUSB instruction subtracts the unsigned bytes of the source operand from the unsigned bytes of the destination operand and stores the results to the destination operand. When an individual result is less than zero (a negative value), the saturated unsigned byte value of 00H is written to the destination operand.

The PSUBUSW instruction subtracts the unsigned words of the source operand from the unsigned words of the destination operand and stores the results to the destination operand. When an individual result is less than zero (a negative value), the saturated unsigned word value of 0000H is written to the destination operand.

PSUBUSB/PSUBUSW—Packed Subtract Unsigned with Saturation (continued)

Operation

```
IF instruction is PSUBUSB
  THEN
      DEST(7..0) ← SaturateToUnsignedByte (DEST(7..0 – SRC (7..0) );
      DEST(15..8) ← SaturateToUnsignedByte ( DEST(15..8) – SRC(15..8) );
      DEST(23..16) ← SaturateToUnsignedByte (DEST(23..16) – SRC(23..16) );
      DEST(31..24) ← SaturateToUnsignedByte (DEST(31..24) – SRC(31..24) );
      DEST(39..32) ← SaturateToUnsignedByte (DEST(39..32) – SRC(39..32) );
      DEST(47..40) ← SaturateToUnsignedByte (DEST(47..40) – SRC(47..40) );
      DEST(55..48) ← SaturateToUnsignedByte (DEST(55..48) – SRC(55..48) );
      DEST(63..56) ← SaturateToUnsignedByte (DEST(63..56) – SRC(63..56) );
  ELSE { (* instruction is PSUBUSW *)
      DEST(15..0) ← SaturateToUnsignedWord (DEST(15..0) – SRC(15..0) );
      DEST(31..16) ← SaturateToUnsignedWord (DEST(31..16) – SRC(31..16) );
      DEST(47..32) ← SaturateToUnsignedWord (DEST(47..32) – SRC(47..32) );
      DEST(63..48) ← SaturateToUnsignedWord (DEST(63..48) – SRC(63..48) );
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

PSUBUSB/PSUBUSW—Packed Subtract Unsigned with Saturation (continued)

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

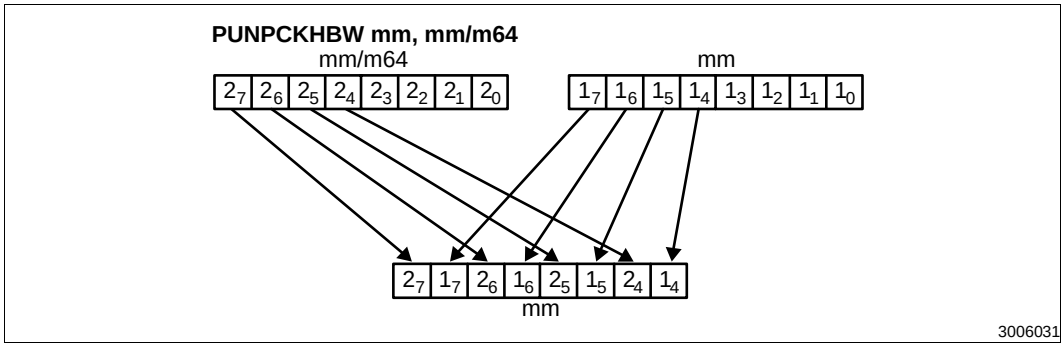
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ—Unpack High Packed Data

Opcode	Instruction	Description
0F 68 /r	PUNPCKHBW *mm, mm/m64*	Interleave high-order bytes from *mm* and *mm/m64* into *mm.*
0F 69 /r	PUNPCKHWD *mm, mm/m64*	Interleave high-order words from *mm* and *mm/m64* into *mm.*
0F 6A /r	PUNPCKHDQ *mm, mm/m64*	Interleave high-order doublewords from *mm* and *mm/m64* into *mm.*

Description

Unpacks and interleaves the high-order data elements (bytes, words, or doublewords) of the destination operand (first operand) and source operand (second operand) into the destination operand (see Figure 6-22). The low-order data elements are ignored. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location. When the source data comes from a memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits.

Figure 6-22. High-order Unpacking and Interleaving of Bytes with the PUNPCKHBW Instruction



The PUNPCKHBW instruction interleaves the four high-order bytes of the source operand and the four high-order bytes of the destination operand and writes them to the destination operand.

The PUNPCKHWD instruction interleaves the two high-order words of the source operand and the two high-order words of the destination operand and writes them to the destination operand.

The PUNPCKHDQ instruction interleaves the high-order doubleword of the source operand and the high-order doubleword of the destination operand and writes them to the destination operand.

If the source operand is all zeros, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. With the PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned words), and with the PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doublewords).

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ—Unpack High Packed Data (continued)

Operation

```
IF instruction is PUNPCKHBW
  THEN
       DEST(7..0) ← DEST(39..32);
       DEST(15..8) ← SRC(39..32);
       DEST(23..16) ← DEST(47..40);
       DEST(31..24) ← SRC(47..40);
       DEST(39..32) ← DEST(55..48);
       DEST(47..40) ← SRC(55..48);
       DEST(55..48) ← DEST(63..56);
       DEST(63..56) ← SRC(63..56);
ELSE IF instruction is PUNPCKHW
  THEN
       DEST(15..0) ← DEST(47..32);
       DEST(31..16) ← SRC(47..32);
       DEST(47..32) ← DEST(63..48);
       DEST(63..48) ← SRC(63..48);
  ELSE (* instruction is PUNPCKHDQ *)
       DEST(31..0) ← DEST(63..32)
       DEST(63..32) ← SRC(63..32);
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ—Unpack High Packed Data
(continued)

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

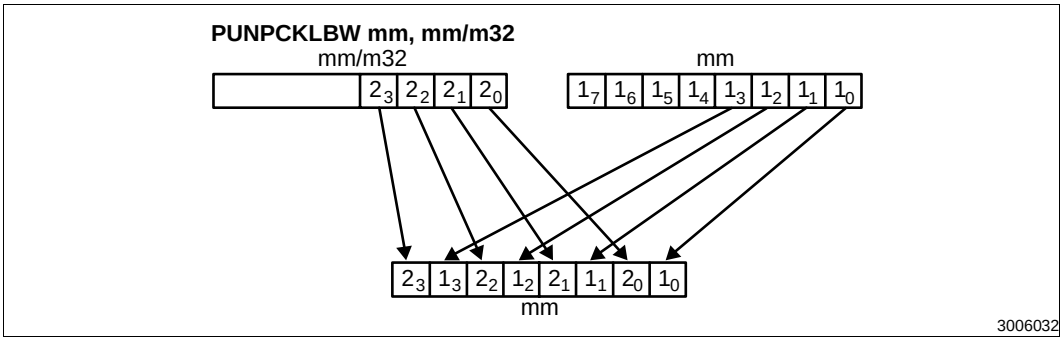
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ—Unpack Low Packed Data

Opcode	Instruction	Description
0F 60 /r	PUNPCKLBW *mm, mm/m32*	Interleave low-order bytes from *mm* and *mm/m64* into *mm.*
0F 61 /r	PUNPCKLWD *mm, mm/m32*	Interleave low-order words from *mm* and *mm/m64* into *mm.*
0F 62 /r	PUNPCKLDQ *mm, mm/m32*	Interleave low-order doublewords from *mm* and *mm/m64* into *mm.*

Description

Unpacks and interleaves the low-order data elements (bytes, words, or doublewords) of the destination and source operands into the destination operand (see Figure 6-23). The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a memory location. When source data comes from an MMX technology register, the upper 32 bits of the register are ignored. When the source data comes from a memory, only 32-bits are accessed from memory.

Figure 6-23. Low-order Unpacking and Interleaving of Bytes with the PUNPCKLBW Instruction



The PUNPCKLBW instruction interleaves the four low-order bytes of the source operand and the four low-order bytes of the destination operand and writes them to the destination operand.

The PUNPCKLWD instruction interleaves the two low-order words of the source operand and the two low-order words of the destination operand and writes them to the destination operand.

The PUNPCKLDQ instruction interleaves the low-order doubleword of the source operand and the low-order doubleword of the destination operand and writes them to the destination operand.

If the source operand is all zeros, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. With the PUNPCKLBW instruction the low-order bytes are zero extended (that is, unpacked into unsigned words), and with the PUNPCKLWD instruction, the low-order words are zero extended (unpacked into unsigned doublewords).

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ—Unpack Low Packed Data (continued)

Operation

```
IF instruction is PUNPCKLBW
  THEN
       DEST(63..56) ← SRC(31..24);
       DEST(55..48) ← DEST(31..24);
       DEST(47..40) ← SRC(23..16);
       DEST(39..32) ← DEST(23..16);
       DEST(31..24) ← SRC(15..8);
       DEST(23..16) ← DEST(15..8);
       DEST(15..8) ← SRC(7..0);
       DEST(7..0) ← DEST(7..0);
ELSE IF instruction is PUNPCKLWD
  THEN
       DEST(63..48) ← SRC(31..16);
       DEST(47..32) ← DEST(31..16);
       DEST(31..16) ← SRC(15..0);
       DEST(15..0) ← DEST(15..0);
  ELSE (* instruction is PUNPCKLDQ *)
       DEST(63..32) ← SRC(31..0);
       DEST(31..0) ← DEST(31..0);
FI;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ—Unpack Low Packed Data (continued)

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

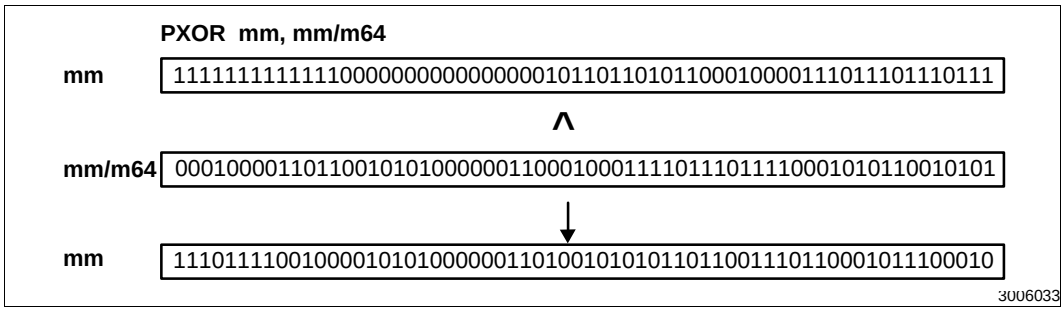
PXOR—Logical Exclusive OR

Opcode	Instruction	Description
0F EF /r	PXOR *mm, mm/m64*	XOR quadword from *mm/m64* to quadword in *mm*.

Description

Performs a bitwise logical exclusive-OR (XOR) operation on the quadword source (second) and destination (first) operands and stores the result in the destination operand location (see Figure 6-24). The source operand can be an MMX technology register or a quadword memory location; the destination operand must be an MMX technology register. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

Figure 6-24. Operation of the PXOR Instruction



Operation

```
DEST ← DEST XOR SRC;
```

Flags Affected

None.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
IA-64 Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PXOR—Logical Exclusive OR (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

IA-32 Streaming SIMD Extension Instruction Reference 7

7.1 IA-32 Streaming SIMD Extension Instructions

This section lists the IA-32 Streaming SIMD Extension instructions designed to increase performance of IA-32 3D and floating-point intensive applications. For details on Streaming SIMD Extension please refer to the *Intel Architecture Software Developer's Manual*.

7.2 About the Intel Architecture Streaming SIMD Extensions

The Streaming SIMD Extensions for the Intel Architecture (IA) accelerates performance of 3D graphics applications over the current P6 generation of the Pentium Pro, Pentium II and Pentium III processors. The programming model is similar to the MMX technology model except that instructions now operate on new packed floating-point data types which contain four single-precision floating-point numbers.

The Streaming SIMD Extensions introduces new general purpose floating-point instructions, which operate on a new set of eight 128-bit Streaming SIMD Extension registers. This gives the programmer the ability to develop algorithms that can finely mix packed single-precision floating-point and integer using both Streaming SIMD Extension and MMX instructions respectively. In addition to these instructions, Streaming SIMD Extensions also provides new instructions to control cacheability of all MMX technology data types. These include ability to stream data into and from the processor while minimizing pollution of the caches and the ability to prefetch data before it is actually used. The main focus of packed floating-point instructions is the acceleration of 3D geometry. The new definition also contains additional SIMD Integer instructions to accelerate 3D rendering and video encoding and decoding. Together with the cacheability control instruction, this combination enables the development of new algorithms that can significantly accelerate 3D graphics.

The new Streaming SIMD Extension state requires OS support for saving and restoring the new state during a context switch. A new set of extended FSAVE/FRSTOR instructions will permit saving/restoring new and existing state for applications and OS. To make use of these new instructions, an application must verify that the processor supports Streaming SIMD Extensions extensions and the operating system supports this new extension. If both the extension and support is enabled, then the software application can use the new features.

The Streaming SIMD Extension instruction set is fully compatible with all software written for Intel Architecture microprocessors. All existing software continues to run correctly, without modification, on microprocessors that incorporate the Streaming SIMD Extensions, as well as in the presence of existing and new applications that incorporate this technology.

7.3 Single Instruction Multiple Data

The Streaming SIMD Extensions uses the Single Instruction Multiple Data (SIMD) technique. This technique speeds up software performance by processing multiple data elements in parallel, using a single instruction. The Streaming SIMD Extensions supports operations on packed single-precision floating-point data types, and the additional SIMD Integer instructions support operations on packed quadrate data types (byte, word, or double-word). This approach was chosen because most 3D graphics and DSP applications have the following characteristics:

- Inherently parallel
- Wide dynamic range, hence floating-point based
- Regular and re-occurring memory access patterns
- Localized re-occurring operations performed on the data
- Data independent control flow

Streaming SIMD Extensions is 100% compatible with the IEEE Standard 754 for Binary Floating-point Arithmetic. The Streaming SIMD Extension instructions are accessible from all IA execution modes: Protected mode, Real address mode, and Virtual 8086 mode.New Features

Streaming SIMD Extensions provides the following new features, while maintaining backward compatibility with all existing Intel Architecture microprocessors, IA applications and operating systems.

- New data type
- Eight Streaming SIMD Extension registers
- Enhanced instruction set

Streaming SIMD Extensions can enhance the performance of applications that use these features.

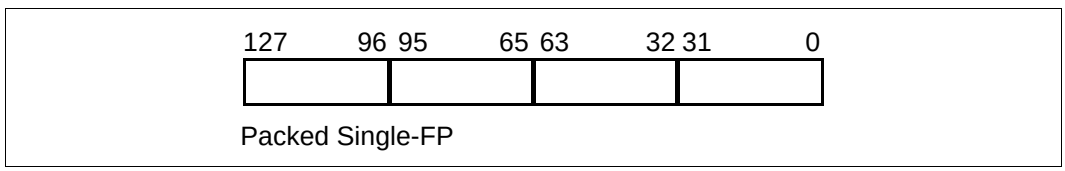
7.4 New Data Types

The principal data type of the Streaming SIMD Extensions is a packed single-precision floating-point operand, specifically:

- Four 32-bit single-precision (SP) floating-point numbers (Figure 7-1).

The SIMD Integer instructions will operate on the packed byte, word or doubleword data types. The prefetch instruction works on typeless data of size 32 bytes or greater.

Figure 7-1. Packed Single-FP Data Type



7.5 Streaming SIMD Extension Registers

The Streaming SIMD Extensions provides eight 128-bit general purpose registers, each of which can be directly addressed. These registers are new state, and require support from the operating system to use them.

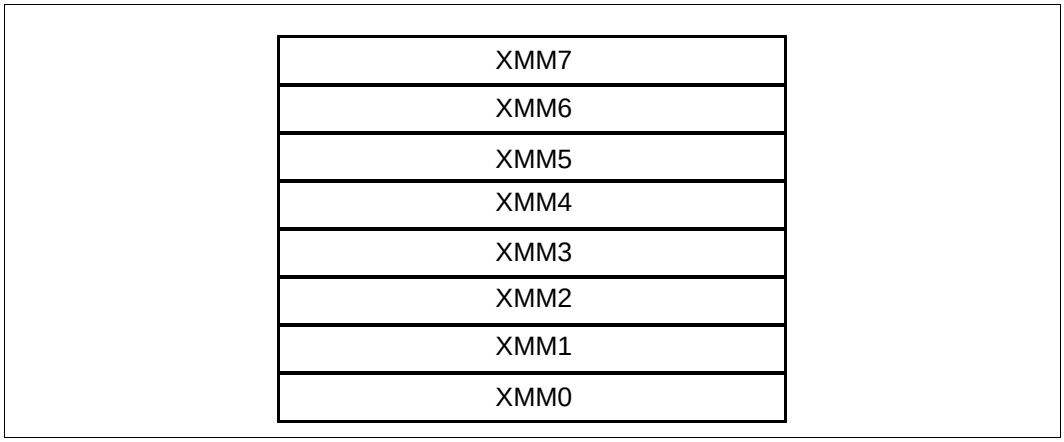
The Streaming SIMD Extension registers can hold packed 128-bit data. The Streaming SIMD Extension instructions access the Streaming SIMD Extension registers directly using the registers names XMM0 to XMM7 (Figure 7-2).

Streaming SIMD Extension registers can be used to perform calculation on data. They cannot be used to address memory; addressing is accomplished by using the integer registers and existing IA addressing modes.

The contents of Streaming SIMD Extension registers are cleared upon reset.

There is a new control/status register MXCSR which is used to mask/unmask numerical exception handling, to set rounding modes, to set flush-to-zero mode, and to view status flags.

Figure 7-2. Streaming SIMD Extension Register Set



7.6 Extended Instruction Set

The Streaming SIMD Extensions supplies a rich set of instructions that operate on either all or the least significant pairs of packed data operands, in parallel. The packed instructions operate on a pair of operands as shown in Figure 7-3 while scalar instructions always operate on the least significant pair of the two operands as shown in Figure 7-4; for scalar operations, the three upper components from the first operand are passed through to the destination. In general, the address of a memory operand has to be aligned on a 16-byte boundary for all instructions, except for unaligned loads and stores.

Figure 7-3. Packed Operation

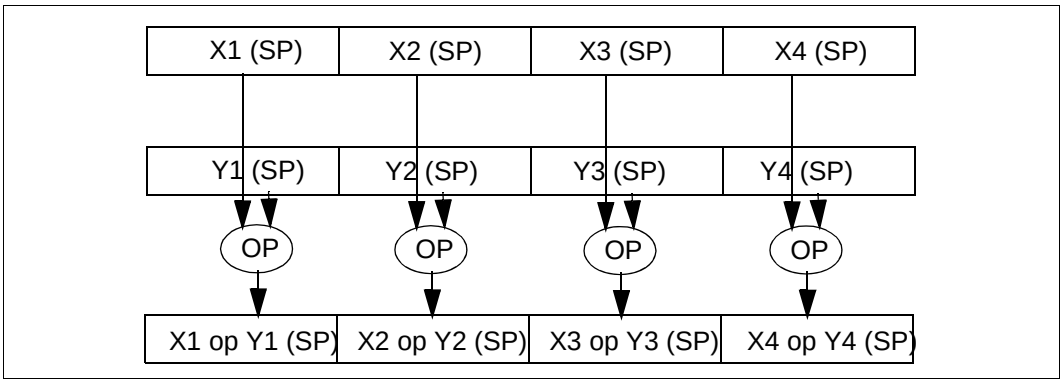
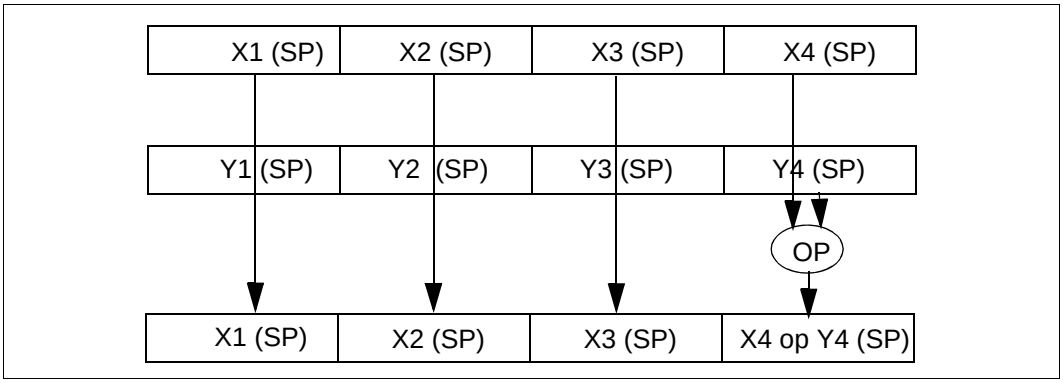


Figure 7-4. Scalar Operation



7.6.1 Instruction Group Review

7.6.1.1 Arithmetic Instructions

Packed/Scalar Addition and Subtraction
The ADDPS (Add packed single-precision floating-point) and SUBPS (Subtract packed single-precision floating-point) instructions add or subtract four pairs of packed single-precision floating-point operands.

The ADDSS (Add scalar single-precision floating-point) and SUBSS (Subtract scalar single-precision floating-point) instructions add or subtract the least significant pair of packed single-precision floating-point operands; the upper three fields are passed through from the source operand.

Packed/Scalar Multiplication and Division
The MULPS (Multiply packed single-precision floating-point) instruction multiplies four pairs of packed single-precision floating-point operands.

The MULSS (Multiply scalar single-precision floating-point) instruction multiplies the least significant pair of packed single-precision floating-point operands; the upper three fields are passed through from the source operand.

The DIVPS (Divide packed single-precision floating-point) instruction divides four pairs of packed single-precision floating-point operands.

The DIVSS (Divide scalar single-precision floating-point) instruction divides the least significant pair of packed single-precision floating-point operands; the upper three fields are passed through from the source operand.

Packed/Scalar Square Root
The SQRTPS (Square root packed single-precision floating-point) instruction returns the square root of the packed four single-precision floating-point numbers from the source to a destination register.

The SQRTSS (Square root scalar single-precision floating-point) instruction returns the square root of the least significant component of the packed single-precision floating-point numbers from source to a destination register; the upper three fields are passed through from the source operand.

Packed Maximum/Minimum
The MAXPS (Maximum packed single-precision floating-point) instruction returns the maximum of each pair of packed single-precision floating-point numbers into the destination register.

The MAXSS (Maximum scalar single-precision floating-point) instructions returns the maximum of the least significant pair of packed single-precision floating-point numbers into the destination register; the upper three fields are passed through from the source operand, to the destination register.

The MINPS (Minimum packed single-precision floating-point) instruction returns the minimum of each pair of packed single-precision floating-point numbers into the destination register.

The MINSS (Minimum scalar single-precision floating-point) instruction returns the minimum of the least significant pair of packed single-precision floating-point numbers into the destination register; the upper three fields are passed through from the source operand, to the destination register

7.6.1.2 Logical Instructions

The ANDPS (Bit-wise packed logical AND for single-precision floating-point) instruction returns a bitwise AND between the two operands.

The ANDNPS (Bit-wise packed logical AND NOT for single-precision floating-point) instruction returns a bitwise AND NOT between the two operands.

The ORPS (Bit-wise packed logical OR for single-precision floating-point) instruction returns a bitwise OR between the two operands.

The XORPS (Bit-wise packed logical XOR for single-precision floating-point) instruction returns a bitwise XOR between the two operands.

7.6.1.3 Compare Instructions

The CMPPS (Compare packed single-precision floating-point) instruction compares four pairs of packed single-precision floating-point numbers using the immediate operand as a predicate, returning per SP field an all “1” 32-bit mask or an all “0” 32-bit mask as a result. The instruction supports a full set of 12 conditions: equal, less than, less than equal, greater than, greater than or equal, unordered, not equal, not less than, not less than or equal, not greater than, not greater than or equal, ordered.

The CMPSS (Compare scalar single-precision floating-point) instruction compares the least significant pairs of packed single-precision floating-point numbers using the immediate operand as a predicate (same as CMPPS), returning per SP field an all “1” 32-bit mask or an all “0” 32-bit mask as a result.

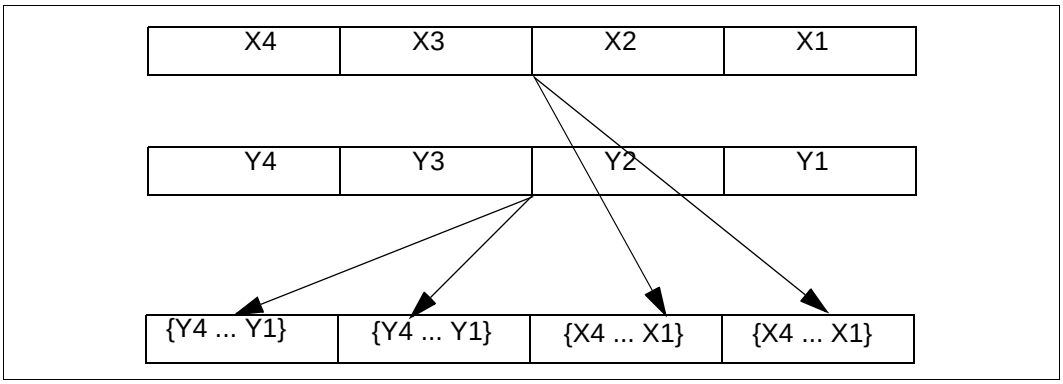
The COMISS (Compare scalar single-precision floating-point ordered and set EFLAGS) instruction compares the least significant pairs of packed single-precision floating-point numbers and sets the ZF,PF,CF bits in the EFLAGS register (the OF, SF and AF bits are cleared).

The UCOMISS (Unordered compare scalar single-precision floating-point ordered and set EFLAGS) instruction compares the least significant pairs of packed single-precision floating-point numbers and sets the ZF,PF,CF bits in the EFLAGS register as described above (the OF, SF and AF bits are cleared).

7.6.1.4 Shuffle Instructions

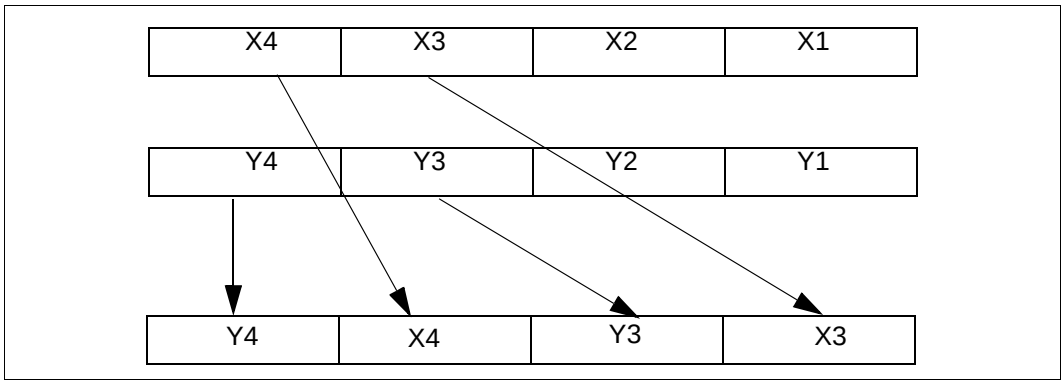
The SHUFPS (Shuffle packed single-precision floating-point) instruction is able to shuffle any of the packed four single-precision floating-point numbers from one source operand to the lower two destination fields; the upper two destination fields are generated from a shuffle of any of the four SP FP numbers from the second source operand (Figure 7-5). By using the same register for both sources, SHUFPS can return any combination of the four SP FP numbers from this register.

Figure 7-5. Packed Shuffle Operation



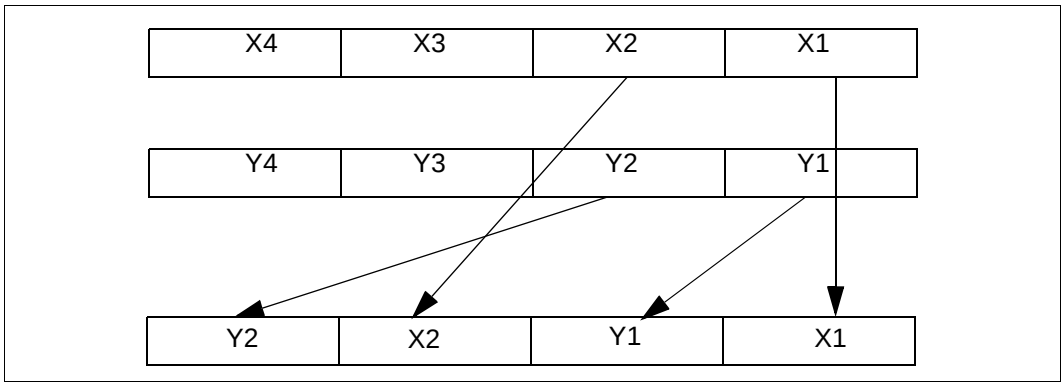
The UNPCKHPS (Unpacked high packed single-precision floating-point) instruction performs an interleaved unpack of the high-order data elements of first and second packed single-precision floating-point operands. It ignores the lower half part of the sources (Figure 7-6). When unpacking from a memory operand, the full 128-bit operand is accessed from memory but only the high order 64 bits are utilized by the instruction.

Figure 7-6. Unpack High Operation



The UNPCKLPS (Unpacked low packed single-precision floating-point) instruction performs an interleaved unpack of the low-order data elements of first and second packed single-precision floating-point operands. It ignores the higher half part of the sources (Figure 7-7). When unpacking from a memory operand, the full 128-bit operand is accessed from memory but only the low order 64 bits are utilized by the instruction.

Figure 7-7. Unpack Low Operation



7.6.1.5 Conversion Instructions

These instructions support packed and scalar conversions between 128-bit Streaming SIMD Extension registers and either 64-bit integer MMX technology registers or 32-bit integer IA-32 registers. The packed versions behave identically to original MMX instructions, in the presence of x87-FP instructions, including:

- Transition from x87-FP to MMX technology (TOS=0, FP valid bits set to all valid).
- MMX instructions write ones (1's) to the exponent part of the corresponding x87-FP register.
- Use of EMMS for transition from MMX technology to x87-FP.

The CVTPI2PS (Convert packed 32-bit integer to packed single-precision floating-point) instruction converts two 32-bit signed integers in a MMX technology register to the two least significant single-precision floating-point numbers; when the conversion is inexact, the rounded value according to the rounding mode in MXCSR is returned. The upper two significant numbers in the destination register are retained.

The CVTSI2SS (Convert scalar 32-bit integer to scalar single-precision floating-point) instruction converts a 32-bit signed integer in a MMX technology register to the least significant single-precision floating-point number; when the conversion is inexact, the rounded value according to the rounding mode in MXCSR is returned. The upper three significant numbers in the destination register are retained.

The CVTPS2PI (Convert packed single-precision floating-point to packed 32-bit integer) instruction converts the two least significant single-precision floating-point numbers to two 32-bit signed integers in a MMX technology register; when the conversion is inexact, the rounded value according to the rounding mode in MXCSR is returned. The CVTTPS2PI (Convert truncate packed single-precision floating-point to packed 32-bit integer) instruction is similar to CVTPS2PI except if the conversion is inexact, in which case the truncated result is returned.

The CVTSS2SI (Convert scalar single-precision floating-point to a 32-bit integer) instruction converts the least significant single-precision floating-point number to a 32-bit signed integer in an Intel Architecture 32-bit integer register; when the conversion is inexact, the rounded value according to the rounding mode in MXCSR is returned.The CVTTSS2SI (Convert truncate scalar single-precision floating-point to scalar 32-bit integer) instruction is similar to CVTSS2SI except if the conversion is inexact, the truncated result is returned.

7.6.1.6 Data Movement Instructions

The MOVAPS (Move aligned packed single-precision floating-point) instruction transfers 128-bits of packed data from memory to Streaming SIMD Extension registers and vice versa, or between Streaming SIMD Extension registers. The memory address is aligned to 16-byte boundary; if not then a general protection exception will occur.

The MOVUPS (Move unaligned packed single-precision floating-point) instruction transfers 128-bits of packed data from memory to Streaming SIMD Extension registers and vice versa, or between Streaming SIMD Extension registers. No assumption is made for alignment.

The MOVHPS (Move aligned high packed single-precision floating-point) instruction transfers 64-bits of packed data from memory to the upper two fields of a Streaming SIMD Extension register and vice versa. The lower field is left unchanged.

The MOVLPS (Move aligned low packed single-precision floating-point) instruction transfers 64-bits of packed data from memory to the lower two fields of a Streaming SIMD Extension register and vice versa. The upper field is left unchanged.

The MOVMSKPS (Move mask packed single-precision floating-point) instruction transfers the most significant bit of each of the four packed single-precision floating-point number to an IA integer register. This 4-bit value can then be used as a condition to perform branching.

The MOVSS (Move scalar single-precision floating-point) instruction transfers a single 32-bit floating-point number from memory to a Streaming SIMD Extension register or vice versa, and between registers.

7.6.1.7 State Management Instructions

The LDMXCSR (Load Streaming SIMD Extension Control and Status Register) instruction loads the Streaming SIMD Extension control and status register from memory. STMXCSR (Store Streaming SIMD Extension Control and Status Register) instruction stores the Streaming SIMD Extension control and status word to memory.

The FXSAVE instruction saves FP and MMX technology state and Streaming SIMD Extension state to memory. Unlike FSAVE, FXSAVE does not clear the x87-FP state. FXRSTOR loads FP and MMX technology state and Streaming SIMD Extension state from memory.

7.6.1.8 Additional SIMD Integer Instructions

Similar to the conversions instructions discussed in Section 7.6.1.5, these SIMD Integer instructions also behave identically to original MMX instructions, in the presence of x87-FP instructions.

The PAVGB/PAVGW (Average unsigned source sub-operands, without incurring a loss in precision) instructions add the unsigned data elements of the source operand to the unsigned data elements of the destination register. The results of the add are then each independently right shifted right by one bit position. The high order bits of each element are filled with the carry bits of the sums. To prevent cumulative round-off errors, an averaging is performed. The low order bit of each final shifted result is set to 1 if at least one of the two least significant bits of the intermediate unshifted shifted sum is 1.

The PEXTRW (Extract 16-bit word from MMX technology register) instruction moves the word in a MMX technology register selected by the two least significant bits of the immediate operand to the lower half of a 32-bit integer register; the upper word in the integer register is cleared.

The PINSRW (Insert 16-bit word into MMX technology register) instruction moves the lower word in a 32-bit integer register or 16-bit word from memory into one of the four word locations in a MMX technology register, selected by the two least significant bits of the immediate operand.

The PMAXUB/PMAXSW (Maximum of packed unsigned integer bytes or signed integer words) instruction returns the maximum of each pair of packed elements into the destination register.

The PMINUB/PMINSW (Minimum of packed unsigned integer bytes or signed integer words) instructions returns the minimum of each pair of packed data elements into the destination register.

The PMOVMSKB (Move Byte Mask from MMX technology register) instruction returns an 8-bit mask formed of the most significant bits of each byte of its source operand in a MMX technology register to an IA integer register.

The PMULHUW (Unsigned high packed integer word multiply in MMX technology register) instruction performs an unsigned multiply on each word field of the two source MMX technology registers, returning the high word of each result to a MMX technology register.

The PSADBW (Sum of absolute differences) instruction computes the absolute difference for each pair of sub-operand byte sources and then accumulates the 8 differences into a single 16-bit result.

The PSHUFW (Shuffle packed integer word in MMX technology register) instruction performs a full shuffle of any source word field to any result word field, using an 8-bit immediate operand.

7.6.1.9 Cacheability Control Instructions

Data referenced by a programmer can have temporal (data will be used again) or spatial (data will be in adjacent locations, e.g. same cache line) locality. Some multimedia data types, such as the display list in a 3D graphics application, are referenced once and not reused in the immediate future. We will refer to this data type as non-temporal data. Thus the programmer does not want the application's cached code and data to be overwritten by this non-temporal data. The cacheability control instructions enable the programmer to control caching so that non-temporal accesses will minimize cache pollution.

In addition, the execution engine needs to be fed such that it does not become stalled waiting for data. Streaming SIMD Extension instructions allow the programmer to prefetch data long before it's final use. These instructions are not architectural since they do not update any architectural state, and are specific to each implementation. The programmer may have to tune his application for each implementation to take advantage of these instructions. These instructions merely provide a hint to the hardware, and they will not generate exceptions or faults. Excessive use of prefetch instructions may be throttled by the processor.

The following four instructions provide hints to the cache hierarchy which enables the data to be prefetched to different levels of the cache hierarchy and avoid polluting cache with non-temporal data.

The MASKMOVQ (Non-temporal byte mask store of packed integer in a MMX technology register) instruction stores data from a MMX technology register to the location specified by the EDI register. The most significant bit in each byte of the second MMX technology mask register is used to selectively write the data of the first register on a per-byte basis. The instruction is implicitly weakly-ordered, with all of the characteristics of the WC memory type; successive non-temporal stores may not write memory in program-order, do not write-allocate (i.e. the processor will not fetch the corresponding cache line into the cache hierarchy, prior to performing the store), write combine/collapse, and minimize cache pollution.

The MOVNTQ (Non-temporal store of packed integer in a MMX technology register) instruction stores data from a MMX technology register to memory. The instruction is implicitly weakly-ordered, does not write-allocate and minimizes cache pollution.

The MOVNTPS (Non-temporal store of packed single-precision floating-point) instruction stores data from a Streaming SIMD Extension register to memory. The memory address must be aligned to a 16-byte boundary; if it is not aligned, a general protection exception will occur. The instruction is implicitly weakly-ordered, does not write-allocate and minimizes cache pollution.

The main difference between a non-temporal store and a regular cacheable store is in the write-allocation policy. The memory type of the region being written to can override the non-temporal hint, leading to the following considerations:

- If the programmer specifies a non-temporal store to uncacheable memory, then the store behaves like an uncacheable store; the non-temporal hint is ignored and the memory type for the region is retained. Uncacheable as referred to here means that the region being written to has been mapped with either a UC or WP memory type. If the memory region has been mapped as WB, WT or WC, the non-temporal store will implement weakly-ordered (WC) semantic behavior.

- If the programmer specifies a non-temporal store to cacheable memory, two cases may result:
  - If the data is present in the cache hierarchy, the instruction will ensure consistency. A given processor may choose different ways to implement this; some examples include: updating data in-place in the cache hierarchy while preserving the memory type semantics assigned to that region, or evicting the data from the caches and writing the new non-temporal data to memory (with WC semantics).
  - If the data is not present in the cache hierarchy, and the destination region is mapped as WB, WT or WC, the transaction will be weakly ordered, and is subject to all WC memory semantics. The non-temporal store will not write allocate. Different implementations may choose to collapse and combine these stores.
- In general, WC semantics require software to ensure coherence, with respect to other processors and other system agents (such as graphics cards). Appropriate use of synchronization and a fencing operation (see SFENCE, below) must be performed for producer-consumer usage models. Fencing ensures that all system agents have global visibility of the stored data; for instance, failure to fence may result in a written cache line staying within a processor, and the line would not be visible to other agents. For processors which implement non-temporal stores by updating data in-place that already resides in the cache hierarchy, the destination region should also be mapped as WC. Otherwise if mapped as WB or WT, there is the potential for speculative processor reads to bring the data into the caches; in this case, non-temporal stores would then update in place, and data would not be flushed from the processor by a subsequent fencing operation.
- The memory type visible on the bus in the presence of memory type aliasing is implementation specific. As one possible example, the memory type written to the bus may reflect the memory type for the first store to this line, as seen in program order; other alternatives are possible. This behavior should be considered reserved, and dependency on the behavior of any particular implementation risks future incompatibility.

The PREFETCH (Load 32 or greater number of bytes) instructions load either non-temporal data or temporal data in the specified cache level. This access and the cache level are specified as a hint. The prefetch instructions do not affect functional behavior of the program and will be implementation specific.

The SFENCE (Store Fence) instruction guarantees that every store instruction that precedes the store fence instruction in program order is globally visible before any store instruction which follows the fence. The SFENCE instruction provides an efficient way of ensuring ordering between routines that produce weakly-ordered results and routines that consume this data.

7.7 IEEE Compliance

Streaming SIMD Extension floating-point computation is IEEE-754 compliant except when the control word is set to flush to zero mode. IEEE-754 compliance includes support for single-precision signed infinities, QNaNs, SNaNs, integer indefinite, signed zeros, denormals, masked and unmasked exceptions. single-precision floating-point values are represented identically both internally and in memory, and are of the following form:

Sign	Exponent	Significand
31	30...23	22...0

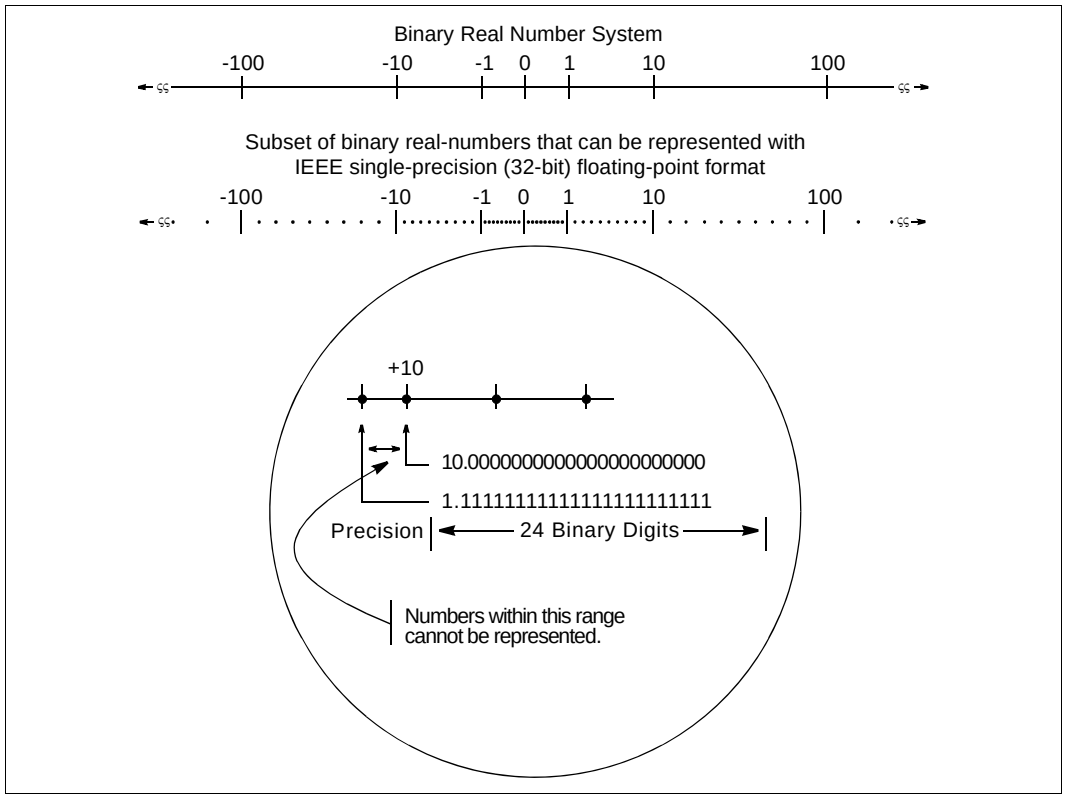
This is a change from x87 floating-point which internally represents all numbers in 80-bit extended format. This change implies that x87-FP libraries re-written to use Streaming SIMD Extension instructions may not produce results that are identical to the those of the x87-FP implementation.Real Numbers and Floating-point Formats.

This section describes how real numbers are represented in floating-point format in the processor. 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.

7.7.1 Real Number System

As shown in Figure 7-8, the real-number system comprises the continuum of real numbers from minus infinity (–∞) to plus infinity (+∞).

Figure 7-8. 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 processor 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 processor uses to represent real numbers.

7.7.1.1 Floating-point Format

To increase the speed and efficiency of real-number computations, computers 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 7-9 shows the binary floating-point format that Streaming SIMD Extension data 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 7-9. Binary Floating-point Format

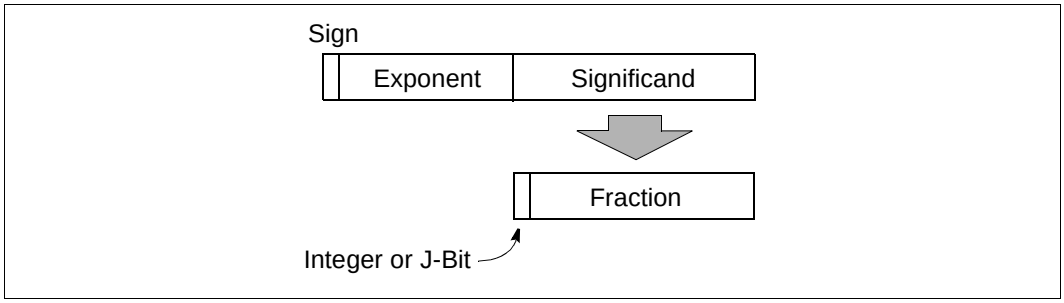


Table 7-1 shows how the real number 178.125 (in ordinary decimal format) is stored in floating-point format. The table lists a progression of real number notations that leads to the format that the processor uses. In this format, the binary real number is normalized and the exponent is biased.

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 Format (Normalized)	Sign	Biased Exponent	Significand
	0	10000110	01100100010000000000000 1 (Implied)

7.7.1.2 Normalized Numbers

In most cases, the processor 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.)

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.

7.7.1.3 Biased Exponent

The processor 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.

7.7.1.4 Real Number and Non-Number Encodings

A variety of real numbers and special values can be encoded in the processor'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 7-10 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 processor 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.

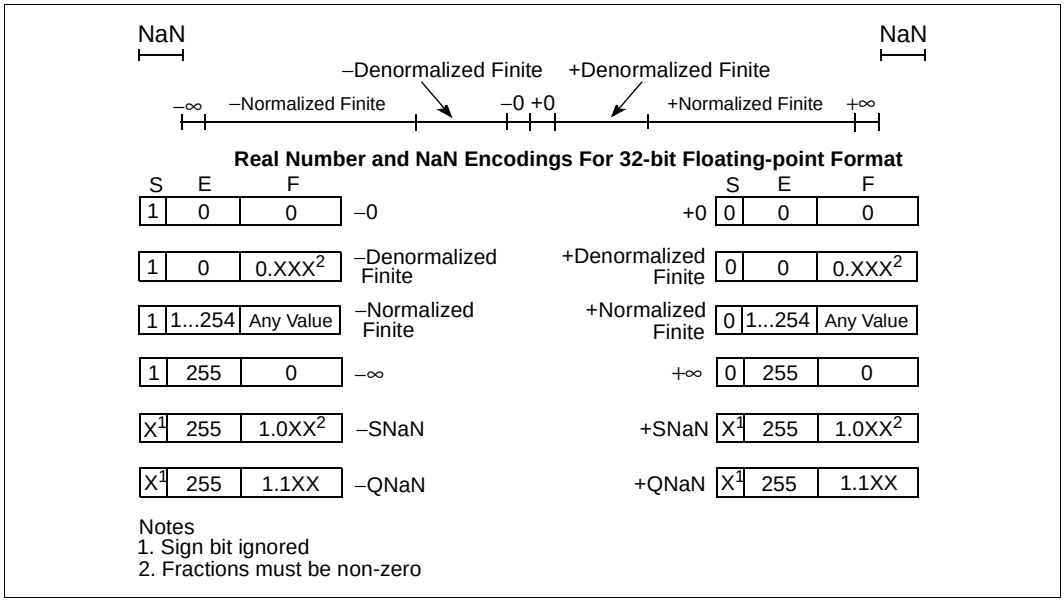
7.7.1.5 Signed Zeros

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.

7.7.1.6 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 format shown in Figure 7-10, 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 7-10. 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, a processor 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[a]	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

a. 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 processor 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.

7.7.1.7 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.

7.7.1.8 NaNs

Since NaNs are non-numbers, they are not part of the real number line. In Figure 7-10, the encoding space for NaNs in the processor 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, as well as detailed information on how the processor handles NaNs, are discussed in Section 7.7.2.

7.7.1.9 Indefinite

In response to a masked invalid-operation floating-point exceptions, the indefinite value QNAN is produced. The integer indefinite, which can be produced during conversion from single-precision floating-point to 32-bit integer, is defined to be 80000000H.

7.7.2 Operating on NaNs

As was described in Section 7.7.1.8, Streaming SIMD Extension 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.

The invalid operation exception has a flag and a mask bit associated with it in MXCSR. The mask bit determines how the an SNaN value is handled. If the 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 invalid operation flag is set. If the invalid operation mask is clear, an 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 7-3. The exceptions to the behavior described in Table 7-3 are the MINPS and MAXPS instructions. If only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 7-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. This approach for MINPS/MAXPS allows NaN data to be screened out of the bounds-checking portion of an algorithm. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

In general Src1 and Src2 relate to an Streaming SIMD Extension instruction as follows:

ADDPS Src1, Src2/m128

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 7-3. Results of Operations with NAN Operands

Source Operands	NaN Result (invalid operation exception is masked)
An SNaN and a QNaN.	Src1 NaN (converted to QNaN if Src1 is an SNaN).
Two SNaNs.	Src1 NaN (converted to QNaN)
Two QNaNs.	Src1 QNaN
An SNaN and a real value.	The SNaN converted into a QNaN.
A QNaN and a real value.	The QNaN source operand.
An SNaN/QNaN value (for instructions which take only one operand i.e. RCPPS, RCPSS, RSQRTPS, RSQRTSS)	The SNaN converted into a QNaN/the source QNaN.
Neither source operand is a NaN and a floating-point invalid-operation exception is signaled.	The default QNaN *real indefinite*.

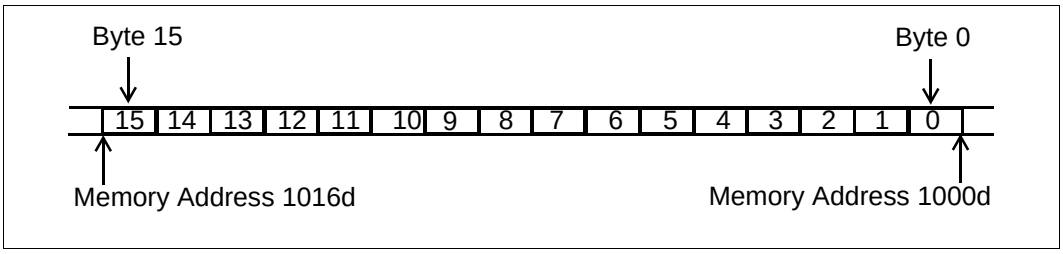
7.8 Data Formats

7.8.1 Memory Data Formats

The Intel Architecture Streaming SIMD Extension introduces a new packed 128-bit data type which consists of 4 single-precision floating-point numbers. The 128 bits are numbered 0 through 127. Bit 0 is the least significant bit (LSB), and bit 127 is the most significant bit (MSB).

Bytes in the new data type format have consecutive memory addresses. The ordering is always little endian, that is, the bytes with the lower addresses are less significant than the bytes with the higher addresses.

Figure 7-11. Four Packed FP Data in Memory (at address 1000H)



7.8.2 Streaming SIMD Extension Register Data Formats

Values in Streaming SIMD Extension registers have the same format as a 128-bit quantity in memory. They have two data access modes: 128-bit access mode and 32-bit access mode. The data type corresponds directly to the single-precision format in the IEEE standard. Table 7-4 gives the

precision and range of this data type. 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. The exponent of the single-precision data type is encoded in biased format. The biasing constant is 127 for the single-precision format.

Table 7-4. Precision and Range of Streaming SIMD Extension Datatype

Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Single-precision	32	24	2^{-126} to 2^{127}	1.18×10^{-38} to 3.40×10^{38}

Table 7-5 shows the encodings for all the classes of real numbers (that is, zero, denormalized-finite, normalized-finite, and ∞) and NaNs for the single-real data-type. It also gives the format for the real indefinite value, which is a QNaN encoding that is generated by several Streaming SIMD Extension instructions in response to a masked floating-point invalid-operation exception.

Table 7-5. 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			⟵ 8 Bits ⟶		⟵ 23 Bits ⟶

When storing real values in memory, single-real values are stored in 4 consecutive bytes in memory. The 128-bit access mode is used for 128-bit memory accesses, 128-bit transfers between Streaming SIMD Extension registers, and all logical, unpack and arithmetic instructions.The 32-bit access mode is used for 32-bit memory access, 32-bit transfers between Streaming SIMD Extension registers, and all arithmetic instructions.

There are sixty-eight new instructions in Streaming SIMD Extension instruction set. This chapter describes the packed and scalar floating-point instructions in alphabetical order, with a full description of each instruction. The last two sections of this chapter describe the SIMD Integer instructions and the cacheability control instructions.

7.9 Instruction Formats

The nature of Streaming SIMD Extension allows the use of existing instruction formats. Instructions use the ModR/M format and are preceded by the 0F prefix byte. In general, operations are not duplicated to provide two directions (i.e. separate load and store variants).

7.10 Instruction Prefixes

The Streaming SIMD Extension instruction uses prefixes as specified in Table 7-6, Table 7-7, and Table 7-8. The effect of multiple prefixes (more than one prefix from a group) is unpredictable and may vary from processor to processor.

Applying a prefix, in a manner not defined in this document, is considered reserved behavior. For example, Table 7-6 shows general behavior for most Streaming SIMD Extension instructions; however, the application of a prefix (Repeat, Repeat NE, Operand Size) is reserved for the following instructions:

ANDPS, ANDNPS, COMISS, FXRSTOR, FXSAVE, ORPS, LDMXCSR, MOVAPS, MOVHPS, MOVLPS, MOVMSKPS, MOVNTPS, MOVUPS, SHUFPS, STMXCSR, UCOMISS, UNPCKHPS, UNPCKLPS, XORPS.

Table 7-6. Streaming SIMD Extension Instruction Behavior with Prefixes

Prefix Type	Effect on Streaming SIMD Extension Instructions
Address Size Prefix (67H)	Affects Streaming SIMD Extension instructions with memory operand
	Ignored by Streaming SIMD Extension instructions without memory operand.
Operand Size (66H)	Reserved and may result in unpredictable behavior.
Segment Override (2EH,36H,3EH,26H,64H,65H)	Affects Streaming SIMD Extension instructions with mem.operand
	Ignored by Streaming SIMD Extension instructions without mem operand
Repeat Prefix (F3H)	Affects Streaming SIMD Extension instructions
Repeat NE Prefix(F2H)	Reserved and may result in unpredictable behavior.
Lock Prefix (0F0H)	Generates invalid opcode exception.

Table 7-7. SIMD Integer Instructions – Behavior with Prefixes

Prefix Type	Effect on MMX™ Instructions
Address Size Prefix (67H)	Affects MMX instructions with mem. operand
	Ignored by MMX instructions without mem. operand.
Operand Size (66H)	Reserved and may result in unpredictable behavior.
Segment Override (2EH,36H,3EH,26H,64H,65H)	Affects MMX instructions with mem. operand
	Ignored by MMX instructions without mem operand
Repeat Prefix (F3H)	Reserved and may result in unpredictable behavior.
Repeat NE Prefix(F2H)	Reserved and may result in unpredictable behavior.
Lock Prefix (0F0H)	Generates invalid opcode exception.

Table 7-8. Cacheability Control Instruction Behavior with Prefixes

Prefix Type	Effect on Streaming SIMD Extension Instructions
Address Size Prefix (67H)	Affects cacheability control instruction with a mem. operand
	Ignored by cacheability control instruction w/o a mem. operand.
Operand Size (66H)	Reserved and may result in unpredictable behavior.
Segment Override (2EH,36H,3EH,26H,64H,65H)	Affects cacheability control instructions with mem. operand
	Ignored by cacheability control instruction without mem operand
Repeat Prefix(F3H)	Reserved and may result in unpredictable behavior.
Repeat NE Prefix(F2H)	Reserved and may result in unpredictable behavior.
Lock Prefix (0F0H)	Generates an invalid opcode exception for all cacheability instructions.

7.11 Reserved Behavior 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 reserved, but unpredictable. In general, reserved behavior may also be applied in other areas. Software should follow these guidelines in dealing with reserved behavior:

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

Note: Avoid any software dependency upon the reserved state/behavior. Depending upon reserved behavior will make the software dependent upon the unspecified manner in which the processor handles this behavior and risks incompatibility with future processors.

7.12 Notations

Besides opcodes, two kinds of notations are found which both describe information found in the ModR/M byte:

1. **/digit:** (digit between 0 and 7) indicates that the instruction uses only the r/m (register and memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
2. **/r**: indicates that the ModR/M byte of an instruction contains both a register operand and an r/m operand.

In addition, the following abbreviations are used:

- **r32**: Intel Architecture 32-bit integer register.
- **xmm/m128**: Indicates a 128-bit multimedia register or a 128-bit memory location.
- **xmm/m64**: Indicates a 128-bit multimedia register or a 64-bit memory location.
- **xmm/m32:** Indicates a 128-bit multimedia register or a 32-bit memory location.
- **mm/m64**: Indicates a 64-bit multimedia register or a 64-bit memory location.
- **imm8**: Indicates an immediate 8-bit operand.
- **ib**: Indicates that an immediate byte operand follows the opcode, ModR/M byte or scaled-indexing byte.

When there is ambiguity, xmm1 indicates the first source operand and xmm2 the second source operand.

Table 7-9 describes the naming conventions used in the Streaming SIMD Extension instruction mnemonics.

Table 7-9. Key to Streaming SIMD Extension Naming Convention

Mnemonic	Description
PI	Packed integer qword (e.g. mm0)
PS	Packed single FP (e.g. xmm0)
SI	Scalar integer (e.g. eax)
SS	Scalar single-FP (e.g. low 32 bits of xmm0)

ADDPS: Packed Single-FP Add

Opcode	Instruction	Description
0F,58,/r	ADDPS xmm1, xmm2/m128	Add packed SP FP numbers from XMM2/Mem to XMM1.

Operation:

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

Description: The ADDPS instruction adds the packed SP FP numbers of both their operands.

Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0)

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

ADDSS: Scalar Single-FP Add

Opcode	Instruction	Description
F3,0F,58, /r	ADDSS xmm1, xmm2/m32	Add the lower SP FP number from XMM2/Mem to XMM1.

Operation:

```
xmm1[31-0]   = xmm1[31-0] + xmm2/m32[31-0];

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The ADDSS instruction adds the lower SP FP numbers of both their operands; the upper 3 fields are passed through from xmm1.

FP Exceptions: None.

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

ANDNPS: Bit-wise Logical And Not for Single-FP

Opcode	Instruction	Description
0F,55,/r	ANDNPS xmm1, xmm2/m128	Invert the 128 bits in XMM1and then AND the result with 128 bits from XMM2/Mem.

Operation:

```
xmm1[127-0] = ~(xmm1[127-0]) & xmm2/m128[127-0];
```

Description: The ANDNPS instructions returns a bit-wise logical AND between the complement of XMM1 and XMM2/Mem.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with ANDNPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with ANDNPS risks incompatibility with future processors.

ANDPS: Bit-wise Logical And for Single-FP

Opcode	Instruction	Description
0F,54,/r	ANDPS xmm1, xmm2/m128	Logical AND of 128 bits from XMM2/Mem to XMM1 register.

Operation: `xmm1[127-0] &= xmm2/m128[127-0];`

Description: The ANDPS instruction returns a bit-wise logical AND between XMM1 and XMM2/Mem.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with ANDPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with ANDPS risks incompatibility with future processors.

CMPPS: Packed Single-FP Compare

Opcode	Instruction	Description
0F,C2,/r,ib	CMPPS xmm1, xmm2/m128, imm8	Compare packed SP FP numbers from XMM2/Mem to packed SP FP numbers in XMM1 register using imm8 as predicate.

Operation:

```
switch (imm8) {
    case eq:      op = eq;
    case lt:      op = lt;
    case le:      op = le;
    case unord:   op = unord;
    case neq:     op = neq;
    case nlt:     op = nlt;
    case nle:     op = nle;
    case ord:     op = ord;
    default:      Reserved;
    }

cmp0 = op(xmm1[31-0],xmm2/m128[31-0]);
cmp1 = op(xmm1[63-32],xmm2/m128[63-32]);
cmp2 = op(xmm1[95-64],xmm2/m128[95-64]);
cmp3 = op(xmm1[127-96],xmm2/m128[127-96]);

xmm1[31-0]    = (cmp0) ? 0xffffffff : 0x00000000;
xmm1[63-32]    = (cmp1) ? 0xffffffff : 0x00000000;
xmm1[95-64]    = (cmp2) ? 0xffffffff : 0x00000000;
xmm1[127-96]    = (cmp3) ? 0xffffffff : 0x00000000;
```

Description: For each individual pairs of SP FP numbers, the CMPPS instruction returns an all “1” 32-bit mask or an all “0” 32-bit mask, using the comparison predicate specified by imm8; note that a subsequent computational instruction which uses this mask as an input operand will not generate a fault, since a mask of all “0’s” corresponds to a FP value of +0.0 and a mask of all “1’s” corresponds to a FP value of -qNaN. Some of the comparisons can be achieved only through software emulation. For these comparisons the programmer must swap the operands, copying registers when necessary to protect the data that will now be in the destination, and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in under the heading “Emulation”. The following table shows the different comparison types:

Predicate	Description[a]	Relation	Emulation	imm8 Encoding	Result if NaN Operand	QNaN Operand Signals Invalid
eq	equal	xmm1 == xmm2		000B	False	No
lt	less-than	xmm1 < xmm2		001B	False	Yes
le	less-than-or-equal	xmm1 <= xmm2		010B	False	Yes
	greater than	xmm1 > xmm2	swap, protect, lt		False	Yes
	greater-than-or-equal	xmm1 >= xmm2	swap protect, le		False	Yes
unord	unordered	xmm1 ? xmm2		011B	True	No
neq	not-equal	!(xmm1 == xmm2)		100B	True	No
nlt	not-less-than	!(xmm1 < xmm2)		101B	True	Yes
nle	not-less-than-or-equal	!(xmm1 <= xmm2)		110B	True	Yes
	not-greater-than	!(xmm1 > xmm2)	swap, protect, nlt		True	Yes
	not-greater-than-or-equal	!(xmm1 >= xmm2)	swap, protect, nle		True	Yes
ord	ordered	!(xmm1 ? xmm2)		111B	False	No

a. The greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations are not directly implemented in hardware.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Invalid if sNaN operand, invalid if qNaN and predicate as listed in above table, denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Compilers and assemblers should implement the following 2-operand pseudo-ops in addition to the 3-operand CMPPS instruction:

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

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

Bits 7-4 of the immediate field are reserved. Different processors may handle them differently. Usage of these bits risks incompatibility with future processors.

CMPSS: Scalar Single-FP Compare

Opcode	Instruction	Description
F3,0F,C2,/r,ib	CMPSS xmm1, xmm2/m32, imm8	Compare lowest SP FP number from XMM2/Mem to lowest SP FP number in XMM1 register using imm8 as predicate.

Operation:

```
switch (imm8) {
    case eq:      op = eq;
    case lt:      op = lt;
    case le:      op = le;
    case unord:   op = unord;
    case neq:     op = neq;
    case nlt:     op = nlt;
    case nle:     op = nle;
    case ord:     op = ord;
    default:      Reserved;
    }

cmp0 = op(xmm1[31-0],xmm2/m32[31-0]);

xmm1[31-0]   = (cmp0) ? 0xffffffff : 0x00000000;
xmm1[63-32]   = xmm1[63-32];
xmm1[95-64]   = xmm1[95-64];
xmm1[127-96]   = xmm1[127-96];
```

Description: For the lowest pair of SP FP numbers, the CMPSS instruction returns an all “1” 32-bit mask or an all “0” 32-bit mask, using the comparison predicate specified by imm8; the values for the upper three pairs of SP FP numbers are not compared. Note that a subsequent computational instruction which uses this mask as an input operand will not generate a fault, since a mask of all “0’s” corresponds to a FP value of +0.0 and a mask of all “1’s” corresponds to a FP value of -qNaN. Some of the comparisons can be achieved only through software emulation. For these comparisons the programmer must swap the operands, copying registers when necessary to protect the data that will now be in the destination, and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in under the heading “Emulation”. The following table shows the different comparison types:

Predicate	Description[a]	Relation	Emulation	imm8 Encoding	Result if NaN Operand	qNaN Operand Signals Invalid
eq	equal	xmm1 == xmm2		000B	False	No
lt	less-than	xmm1 < xmm2		001B	False	Yes
le	less-than-or-equal	xmm1 <= xmm2		010B	False	Yes
	greater than	xmm1 > xmm2	swap, protect, lt		False	Yes
	greater-than-or-equal	xmm1 >= xmm2	swap protect, le		False	Yes
unord	unordered	xmm1 ? xmm2		011B	True	No
neq	not-equal	!(xmm1 == xmm2)		100B	True	No
nlt	not-less-than	!(xmm1 < xmm2)		101B	True	Yes
nle	not-less-than-or-equal	!(xmm1 <= xmm2)		110B	True	Yes
	not-greater-than	!(xmm1 > xmm2)	swap, protect, nlt		True	Yes
	not-greater-than-or-equal	!(xmm1 >= xmm2)	swap, protect, nle		True	Yes
ord	ordered	!(xmm1 ? xmm2)		111B	False	No

a. The greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations are not directly implemented in hardware.

FP Exceptions: None.

Numeric Exceptions: Invalid if sNaN operand, invalid if qNaN and predicate as listed in above table, denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Compilers and assemblers should implement the following 2-operand pseudo-ops in addition to the 3-operand CMPSS instruction:

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

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

Bits 7-4 of the immediate field are reserved. Different processors may handle them differently. Usage of these bits risks incompatibility with future processors.

COMISS: Scalar Ordered Single-FP Compare and set EFLAGS

Opcode	Instruction	Description
0F,2F,/r	COMISS xmm1, xmm2/m32	Compare lower SP FP number in XMM1 register with lower SP FP number in XMM2/Mem and set the status flags accordingly

Operation:

```
switch (xmm1[31-0] <> xmm2/m32[31-0]) {

    OF,SF,AF = 000;

    case UNORDERED:     ZF,PF,CF = 111;

    case GREATER_THAN:  ZF,PF,CF = 000;

    case LESS_THAN:     ZF,PF,CF = 001;

    case EQUAL:         ZF,PF,CF = 100;

}
```

Description: The COMISS instructions compare two SP FP numbers and sets the ZF,PF,CF bits in the EFLAGS register as described above. Although the data type is packed single-FP, only the lower SP numbers are compared. In addition, the OF, SF and AF bits in the EFLAGS register are zeroed out. The unordered predicate is returned if either source operand is a NaN (qNaN or sNaN).

FP Exceptions: None.

Numeric Exceptions: Invalid (if SNaN or QNaN operands), Denormal. Integer EFLAGS values will not be updated in the presence of unmasked numeric exceptions.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: COMISS differs from UCOMISS in that it signals an invalid numeric exception when a source operand is either a qNaN or sNaN; UCOMISS signals invalid only if a source operand is an sNaN.

The usage of Repeat (F2H, F3H) and Operand-Size (66H) prefixes with COMISS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with COMISS risks incompatibility with future processors.

CVTPI2PS: Packed Signed INT32 to Packed Single-FP Conversion

Opcode	Instruction	Description
0F,2A,/r	CVTPI2PS xmm, mm/m64	Convert two 32-bit signed integers from MM/Mem to two SP FP.

Operation:

```
xmm[31-0]   = (float) (mm/m64[31-0]);

xmm[63-32]  = (float) (mm/m64[63-32]);

xmm[95-64]  = xmm[95-64];

xmm[127-96] = xmm[127-96];
```

Description: The CVTPI2PS instruction converts signed 32-bit integers to SP FP numbers; when the conversion is inexact, rounding is done according to MXCSR.

FP Exceptions: None.

Numeric Exceptions: Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: This instruction behaves identically to original MMX instructions, in the presence of x87-FP instructions:

- Transition from x87-FP to MMX technology (TOS=0, FP valid bits set to all valid).
- MMX instructions write ones (1's) to the exponent part of the corresponding x87-FP register.

However, the use of a memory source operand with this instruction will not result in the above transition from x87-FP to MMX technology.

Prioritization for fault and assist behavior for CVTPI2PS is as follows:

Memory source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #SS or #GP, for limit violation
4. #PF, page fault
5. Streaming SIMD Extension numeric fault (i.e. precision)

Register source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #MF, pending x87-FP fault signalled
4. After returning from #MF, x87-FP->MMX technology transition
5. Streaming SIMD Extension numeric fault (i.e. precision)

CVTPS2PI: Packed Single-FP to Packed INT32 Conversion

Opcode	Instruction	Description
0F,2D,/r	CVTPS2PI mm, xmm/m64	Convert lower 2 SP FP from XMM/Mem to 2 32-bit signed integers in MM using rounding specified by MXCSR.

Operation:

```
mm[31-0]   = (int) (xmm/m64[31-0]);

mm[63-32]  = (int) (xmm/m64[63-32]);
```

Description: The CVTPS2PI instruction converts the lower 2 SP FP numbers in xmm/m64 to signed 32-bit integers in mm; when the conversion is inexact, the value rounded according to the MXCSR is returned. If the converted result(s) is/are larger than the maximum signed 32 bit value, the Integer Indefinite value (0x80000000) will be returned.

FP Exceptions: None.

Numeric Exceptions: Invalid, Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: This instruction behaves identically to original MMX instructions, in the presence of x87-FP instructions, including:

- Transition from x87-FP to MMX technology (TOS=0, FP valid bits set to all valid).
- MMX instructions write ones (1's) to the exponent part of the corresponding x87-FP register.

Prioritization for fault and assist behavior for CVTPS2PI is as follows:

Memory source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #MF, pending x87-FP fault signalled
4. After returning from #MF, x87-FP->MMX technology transition
5. #SS or #GP, for limit violation
6. #PF, page fault
7. Streaming SIMD Extension numeric fault (i.e. invalid, precision)

Register source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #MF, pending x87-FP fault signalled
4. After returning from #MF, x87-FP->MMX technology transition
5. Streaming SIMD Extension numeric fault (i.e. precision)

CVTSI2SS: Scalar signed INT32 to Single-FP Conversion

Opcode	Instruction	Description
F3,0F,2A, /r	CVTSI2SS xmm, r/m32	Convert one 32-bit signed integer from Integer Reg/Mem to one SP FP.

Operation:

```
xmm[31-0]   = (float) (r/m32);

xmm[63-32]  = xmm[63-32];

xmm[95-64]  = xmm[95-64];

xmm[127-96] = xmm[127-96];
```

Description: The CVTSI2SS instruction converts a signed 32-bit integer from memory or from a 32-bit integer register to a SP FP number; when the conversion is inexact, rounding is done according to the MXCSR.

FP Exceptions: None.

Numeric Exceptions: Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

CVTSS2SI: Scalar Single-FP to Signed INT32 Conversion

Opcode	Instruction	Description
F3,0F,2D, /r	CVTSS2SI r32, xmm/m32	Convert one SP FP from XMM/Mem to one 32 bit signed integer using rounding mode specified by MXCSR, and move the result to an integer register.

Operation:

```
r32 = (int) (xmm/m32[31-0]);
```

Description: The CVTSS2SI instruction converts a SP FP number to a signed 32-bit integer and returns it in the 32-bit integer register; when the conversion is inexact, the rounded value according to the MXCSR is returned. If the converted result is larger than the maximum signed 32 bit integer, the Integer Indefinite value (0x80000000) will be returned.

FP Exceptions: None.

Numeric Exceptions: Invalid, Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT = 0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

CVTTPS2PI: Packed Single-FP to Packed INT32 Conversion (truncate)

Opcode	Instruction	Description
0F,2C,/r	CVTTPS2PI mm, xmm/m64	Convert lower 2 SP FP from XMM/Mem to 2 32-bit signed integers in MM using truncate.

Operation:

```
mm[31-0]   = (int) (xmm/m64[31-0]);

mm[63-32]  = (int) (xmm/m64[63-32]);
```

Description: The CVTTPS2PI instruction converts the lower 2 SP FP numbers in xmm/m64 to 2 32-bit signed integers in mm; if the conversion is inexact, the truncated result is returned. If the converted result(s) is/are larger than the maximum signed 32 bit value, the Integer Indefinite value (0x80000000) will be returned.

FP Exceptions: None.

Numeric Exceptions: Invalid, Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: This instruction behaves identically to original MMX instructions, in the presence of x87-FP instructions, including:

- Transition from x87-FP to MMX technology (TOS=0, FP valid bits set to all valid).
- MMX instructions write ones (1's) to the exponent part of the corresponding x87-FP register.

Prioritization for fault and assist behavior for CVTTPS2PI is as follows:

Memory source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #MF, pending x87-FP fault signalled
4. After returning from #MF, x87-FP->MMX technology transition
5. #SS or #GP, for limit violation
6. #PF, page fault
7. Streaming SIMD Extension numeric fault (i.e. invalid, precision)

Register source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #MF, pending x87-FP fault signalled
4. After returning from #MF, x87-FP->MMX technology transition
5. Streaming SIMD Extension numeric fault (i.e. precision)

CVTTSS2SI: Scalar Single-FP to signed INT32 Conversion (truncate)

Opcode	Instruction	Description
F3,0F,2C, /r	CVTTSS2SI r32, xmm/m32	Convert lowest SP FP from XMM/Mem to one 32 bit signed integer using truncate, and move the result to an integer register.

Operation: `r32 = (int) (xmm/m32[31-0]);`

Description: The CVTTSS2SI instruction converts a SP FP number to a signed 32-bit integer and returns it in the 32-bit integer register; if the conversion is inexact, the truncated result is returned. If the converted result is larger than the maximum signed 32 bit value, the Integer Indefinite value (0x80000000) will be returned.

FP Exceptions: None.

Numeric Exceptions: Invalid, Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

DIVPS: Packed Single-FP Divide

Opcode	Instruction	Description
0F,5E,/r	DIVPS xmm1, xmm2/m128	Divide packed SP FP numbers in XMM1 by XMM2/Mem

Operation:

```
xmm1[31-0]   = xmm1[31-0]   / (xmm2/m128[31-0]);

xmm1[63-32]  = xmm1[63-32]  / (xmm2/m128[63-32]);

xmm1[95-64]  = xmm1[95-64]  / (xmm2/m128[95-64]);

xmm1[127-96] = xmm1[127-96] / (xmm2/m128[127-96]);
```

Description: The DIVPS instruction divides the packed SP FP numbers of both their operands.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Overflow, Underflow, Invalid, Divide by Zero, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

DIVSS: Scalar Single-FP Divide

Opcode	Instruction	Description
F3,0F,5E, /r	DIVSS xmm1, xmm2/m32	Divide lower SP FP numbers in XMM1 by XMM2/Mem

Operation:

```
xmm1[31-0]   = xmm1[31-0] / (xmm2/m32[31-0]);

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The DIVSS instructions divide the lowest SP FP numbers of both operands; the upper 3 fields are passed through from xmm1.

FP Exceptions: None.

Numeric Exceptions: Overflow, Underflow, Invalid, Divide by Zero, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

FXRSTOR: Restore FP and MMX™ state and Streaming SIMD Extension State

Opcode	Instruction	Description
0F,AE,/1	FXRSTOR m512byte	Load FP/MMX and Streaming SIMD Extension state from m512byte.

Operation: `FP and` MMX state `and` Streaming SIMD Extension `state = m512byte;`

Description: The FXRSTOR instruction reloads the FP and MMX technology state and Streaming SIMD Extension state (environment and registers) from the memory area defined by m512byte. This data should have been written by a previous FXSAVE.

The FP and MMX technology and Streaming SIMD Extension environment and registers consist of the following data structure (little-endian byte order as arranged in memory, with byte offset into row described by right column):

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	
Rsrvd	CS	IP	FOP	0
Reserved	MXCSR	Rsrvd	DS	16
Reserved	ST0/MM0			32
Reserved	ST1/MM1			48
Reserved	ST2/MM2			64
Reserved	ST3/MM3			80
Reserved	ST4/MM4			96
Reserved	ST5/MM5			112
Reserved	ST6/MM6			128
Reserved	ST7/MM7			144
XMM0				160
XMM1				176
XMM2				192
XMM3				208
XMM4				224
XMM5				240
XMM6				256
XMM7				272
Reserved				288
Reserved				304
Reserved				320
Reserved				336
Reserved				352
Reserved				368
Reserved				384
Reserved				400

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Rsrvd		CS		IP					FOP		FTW		FSW		FCW	0
Reserved																416
Reserved																432
Reserved																448
Reserved																464
Reserved																480
Reserved																496

Three fields in the floating-point save area contain reserved bits that are not indicated in the table:

- FOP: The lower 11-bits contain the opcode, upper 5-bits are reserved.
- IP & DP:32-bit mode: 32-bit IP-offset.
- 16-bit mode: lower 16-bits are IP-offset and upper 16-bits are reserved.

If the MXCSR state contains an unmasked exception with corresponding status flag also set, loading it will not result in a floating-point error condition being asserted; only the next occurrence of this unmasked exception will result in the error condition being asserted.

Some bits of MXCSR (bits 31-16 and bit 6) are defined as reserved and cleared; attempting to write a non-zero value to these bits will result in a general protection exception.

FXRSTOR does not flush pending x87-FP exceptions, unlike FRSTOR. To check and raise exceptions when loading a new operating environment, use FWAIT after FXRSTOR.

The Streaming SIMD Extension fields in the save image (XMM0-XMM7 and MXCSR) may not be loaded into the processor if the CR4.OSFXSR bit is not set. This CR4 bit must be set in order to enable execution of Streaming SIMD Extension instructions.

FP Exceptions: If #AC exception detection is disabled, a general protection exception is signalled if the address is not aligned on 16-byte boundary. Note that if #AC is enabled (and CPL is 3), signalling of #AC is not guaranteed and may vary with implementation; in all implementations where #AC is not signalled, a general protection fault will instead be signalled. In addition, the width of the alignment check when #AC is enabled may also vary with implementation; for instance, for a given implementation #AC might be signalled for a 2-byte misalignment, whereas #GP might be signalled for all other misalignments (4/8/16-byte). Invalid opcode exception if instruction is preceded by a LOCK override prefix. General protection fault if reserved bits of MXCSR are loaded with non-zero values

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #NM if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #NM if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Notes: State saved with FXSAVE and restored with FRSTOR (and vice versa) will result in incorrect restoration of state in the processor. The address size prefix will have the usual effect on address calculation but will have no effect on the format of the FXRSTOR image.

The use of Repeat (F2H, F3H) and Operand Size (66H) prefixes with FXRSTOR is reserved. Different processor implementations may handle this prefix differently. Use of this prefix with FXRSTOR risks incompatibility with future processors.

FXSAVE: Store FP and MMX™ State and Streaming SIMD Extension State

Opcode	Instruction	Description
0F,AE,/0	FXSAVE m512byte	Store FP and MMX state and Streaming SIMD Extension state to m512byte.

Operation:

```
m512byte = FP and MMX state and Streaming SIMD Extension state;
```

Description:

The FXSAVE instruction writes the current FP and MMX technology state and Streaming SIMD Extension state (environment and registers) to the specified destination defined by m512byte. It does this without checking for pending unmasked floating-point exceptions, similar to the operation of FNSAVE. Unlike the FSAVE/FNSAVE instructions, the processor retains the contents of the FP and MMX technology state and Streaming SIMD Extension state in the processor after the state has been saved. This instruction has been optimized to maximize floating-point save performance. The save data structure is as follows (little-endian byte order as arranged in memory, with byte offset into row described by right column):

15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0	
Rsrvd	CS	IP		FOP	FTW	FSW	FCW	0
Reserved		MXCSR		Rsrvd	DS	DP		16
Reserved		ST0/MM0						32
Reserved		ST1/MM1						48
Reserved		ST2/MM2						64
Reserved		ST3/MM3						80
Reserved		ST4/MM4						96
Reserved		ST5/MM5						112
Reserved		ST6/MM6						128
Reserved		ST7/MM7						144
XMM0								160
XMM1								176
XMM2								192
XMM3								208
XMM4								224
XMM5								240
XMM6								256
XMM7								272
Reserved								288
Reserved								304
Reserved								320
Reserved								336
Reserved								352
Reserved								368
Reserved								384
Reserved								400

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Rsrvd		CS		IP					FOP		FTW		FSW		FCW	0
Reserved																416
Reserved																432
Reserved																448
Reserved																464
Reserved																480
Reserved																496

Three fields in the floating-point save area contain reserved bits that are not indicated in the table:

- FOP: The lower 11-bits contain the opcode, upper 5-bits are reserved.
- IP & DP: 32-bit mode: 32-bit IP-offset.
- 16-bit mode: lower 16-bits are IP-offset and upper 16-bits are reserved.

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to use the FP and MMX technology and Streaming SIMD Extension units. It cannot be used by an application program to pass a “clean” FP state to a procedure, since it retains the current state. An application must explicitly execute an FINIT instruction after FXSAVE to provide for this functionality.

All of the x87-FP fields retain the same internal format as in FSAVE except for FTW.

Unlike FSAVE, FXSAVE saves only the FTW valid bits rather than the entire x87-FP FTW field. The FTW bits are saved in a non-TOS relative order, which means that FR0 is always saved first, followed by FR1, FR2 and so forth. As an example, if TOS=4 and only ST0, ST1 and ST2 are valid, FSAVE saves the FTW field in the following format:

ST3	ST2	ST1	ST0	ST7	ST6	ST5	ST4 (TOS=4)
FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0
11	xx	xx	xx	11	11	11	11

where xx is one of (00, 01, 10). (11) indicates an empty stack elements, and the 00, 01, and 10 indicate Valid, Zero, and Special, respectively. In this example, FXSAVE would save the following vector:

FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0
0	1	1	1	0	0	0	0

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit FP data (assuming the stored data was not the contents of MMX technology registers) using the following table:

Exponent all 1’s	Exponent all 0’s	Fraction all 0’s	J and M bits	FTW valid bit	x87 FTW	
0	0	0	0x	1	Special	10
0	0	0	1x	1	Valid	00
0	0	1	00	1	Special	10
0	0	1	10	1	Valid	00
0	1	0	0x	1	Special	10
0	1	0	1x	1	Special	10

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

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

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

If the FXSAVE instruction is immediately preceded by an FP instruction which does not use a memory operand, then the FXSAVE instruction does not write/update the DP field, in the FXSAVE image.

MXCSR holds the contents of the Streaming SIMD Extension Control/Status Register. See the LDMXCSR instruction for a full description of this field.

The fields XMM0-XMM7 contain the content of registers XMM0-XMM7 in exactly the same format as they exist in the registers.

The Streaming SIMD Extension fields in the save image (XMM0-XMM7 and MXCSR) may not be loaded into the processor if the CR4.OSFXSR bit is not set. This CR4 bit must be set in order to enable execution of Streaming SIMD Extension instructions.

The destination m512byte is assumed to be aligned on a 16-byte boundary. If m512byte is not aligned on a 16-byte boundary, FXSAVE generates a general protection exception.

FP Exceptions: If #AC exception detection is disabled, a general protection exception is signalled if the address is not aligned on 16-byte boundary. Note that if #AC is enabled (and CPL is 3), signalling of #AC is not guaranteed and may vary with implementation; in all implementations where #AC is not signalled, a general protection fault will instead be signalled. In addition, the width of the alignment check when #AC is enabled may also vary with implementation; for instance, for a given implementation #AC might be signalled for a 2-byte misalignment, whereas #GP might be signalled for all other misalignments (4/8/16-byte). Invalid opcode exception if instruction is preceded by a LOCK override prefix.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #NM if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #NM if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Notes: State saved with FXSAVE and restored with FRSTOR (and vice versa) will result in incorrect restoration of state in the processor. The address size prefix will have the usual effect on address calculation but will have no effect on the format of the FXSAVE image.

If there is a pending unmasked FP exception at the time FXSAVE is executed, the sequence of FXSAVE-FWAIT-FXRSTOR will result in incorrect state in the processor. The FWAIT instruction causes the processor to check and handle pending unmasked FP exceptions. Since the processor does not clear the FP state with FXSAVE (unlike FSAVE), the exception is handled but that fact is not reflected in the saved image. When the image is reloaded using FXRSTOR, the exception bits in FSW will be incorrectly reloaded.

The use of Repeat (F2H, F3H) and Operand Size (66H) prefixes with FXSAVE is reserved. Different processor implementations may handle this prefix differently. Use of these prefixes with FXSAVE risks incompatibility with future processors.

LDMXCSR: Load Streaming SIMD Extension Control/Status

Opcode	Instruction	Description
0F,AE,/2	LDMXCSR m32	Load Streaming SIMD Extension control/status word from m32.

Operation: `MXCSR = m32;`

Description: The MXCSR control/status register is used to enable masked/unmasked exception handling, to set rounding modes, to set flush-to-zero mode, and to view exception status flags. The following figure shows the format and encoding of the fields in MXCSR.

31-16	15				10					5						0
Reserved	FZ	RC	RC	PM	UM	OM	ZM	DM	IM	Rsvd	PE	UE	OE	ZE	DE	IE

Bits 5-0 indicate whether an Streaming SIMD Extension numerical exception has been detected. They are “sticky” flags, and can be cleared by using the LDMXCSR instruction to write zeroes to these fields. If a LDMXCSR instruction clears a mask bit and sets the corresponding exception flag bit, an exception will not be immediately generated. The exception will occur only upon the next Streaming SIMD Extension to cause this type of exception. Streaming SIMD Extension uses only one exception flag for each exception. There is no provision for individual exception reporting within a packed data type. In situations where multiple identical exceptions occur within the same instruction, the associated exception flag is updated and indicates that at least one of these conditions happened. These flags are cleared upon reset.

Bits 12-7 configure numerical exception masking; an exception type is masked if the corresponding bit is set and it is unmasked if the bit is clear. These enables are set upon reset, meaning that all numerical exceptions are masked.

Bits 14-13 encode the rounding-control, which provides for the common round-to-nearest mode, as well as directed rounding and true chop. Rounding control affects the arithmetic instructions and certain conversion instructions. The encoding for RC is as follows:

Rounding Mode	RC Field	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 (to minus infinity)	01B	Rounded result is close to but no greater than the infinitely precise result
Round up (toward positive infinity)	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 rounding-control is set to round to nearest upon reset.

Bit 15 (FZ) is used to turn on the Flush To Zero mode (bit is set). Turning on the Flush To Zero mode has the following effects during underflow situations:

- Zero results are returned with the sign of the true result.
- Precision and underflow exception flags are set.

The IEEE mandated masked response to underflow is to deliver the denormalized result (i.e. gradual underflow); consequently, the flush to zero mode is not compatible with IEEE Std. 754. It is provided primarily for performance reasons. At the cost of a slight precision loss, faster execution can be achieved for applications where underflows are common. Unmasking the underflow exception takes precedence over Flush To Zero mode; this means that an exception handler will be invoked for a Streaming SIMD Extension instruction that generates an underflow condition while this exception is unmasked, regardless of whether flush to zero is enabled.

The other bits of MXCSR (bits 31-16 and bit 6) are defined as reserved and cleared; attempting to write a non-zero value to these bits, using either the FXRSTOR or LDMXCSR instructions, will result in a general protection exception.

The linear address corresponds to the address of the least-significant byte of the referenced memory data.

FP Exceptions: General protection fault if reserved bits are loaded with non-zero values.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault. #AC for unaligned memory reference.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: The usage of Repeat (F2H, F3H) and Operand Size (66H) prefixes with LDMXCSR is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with LDMXCSR risks incompatibility with future processors.

MAXPS: Packed Single-FP Maximum

Opcode	Instruction	Description
0F,5F,/r	MAXPS xmm1, xmm2/m128	Return the maximum SP FP numbers between XMM2/Mem and XMM1.

Operation:

```
xmm1[31-0]        =   (xmm1[31-0] == NAN) ? xmm2[31-0] :
                      (xmm2[31-0] == NAN) ? xmm2[31-0] :
                      (xmm1[31-0] > xmm2/m128[31-0]) ? xmm1[31-0] ?
xmm2/m128[31-0];
xmm1[63-32]       =   (xmm1[63-32] == NAN) ? xmm2[63-32] :
                      (xmm2[63-32] == NAN) ? xmm2[63-32] :
                      (xmm1[63-32] > xmm2/m128[63-32]) ? xmm1[63-32] ?
xmm2/m128[63-32];
xmm1[95-64]       =   (xmm1[95-64] == NAN) ? xmm2[95-64] :
                      (xmm2[95-64] == NAN) ? xmm2[95-64] :
                      (xmm1[95-64] > xmm2/m128[95-64]) ? xmm1[95-64] ?
xmm2/m128[95-64];
xmm1[127-96]      =   (xmm1[127-96] == NAN) ? xmm2[127-96] :
                      (xmm2[127-96] == NAN) ? xmm2[127-96] :
                      (xmm1[127-96] > xmm2/m128[127-96]) ? xmm1[127-96] ?
xmm2/m128[127-96];
```

Description: The MAXPS instruction returns the maximum SP FP numbers from XMM1 and XMM2/Mem. If the values being compared are both zeros, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an sNaN, this sNaN is forwarded unchanged to the destination (i.e. a quieted version of the sNaN is not returned).

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Invalid (including qNaN source operand), Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 7-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. This approach for MAXPS allows compilers to use the MAXPS instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MAXSS: Scalar Single-FP Maximum

Opcode	Instruction	Description
F3,0F,5F,/r	MAXSS xmm1, xmm2/m32	Return the maximum SP FP number between the lower SP FP numbers from XMM2/Mem and XMM1.

Operation:

```
xmm1[31-0]    = (xmm1[31-0] == NAN) ? xmm2[31-0] :
                (xmm2[31-0] == NAN) ? xmm2[31-0] :
                (xmm1[31-0] > xmm2/m32[31-0]) ? xmm1[31-0] : xmm2/m32[31-0];
xmm1[63-32]   = xmm1[63-32];
xmm1[95-64]   = xmm1[95-64];
xmm1[127-96]  = xmm1[127-96];
```

Description: The MAXSS instruction returns the maximum SP FP number from the lower SP FP numbers of XMM1 and XMM2/Mem; the upper 3 fields are passed through from xmm1. If the values being compared are both zeros, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an sNaN, this sNaN is forwarded unchanged to the destination (i.e. a quieted version of the sNaN is not returned).

FP Exceptions: None

Numeric Exceptions: Invalid (including qNaN source operand), Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault

IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 7-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. The upper three operands are still bypassed from the src1 operand, as in all other scalar operations. This approach for MAXSS allows compilers to use the MAXSS instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MINPS: Packed Single-FP Minimum

Opcode	Instruction	Description
0F,5D,/r	MINPS xmm1, xmm2/m128	Return the minimum SP numbers between XMM2/Mem and XMM1.

Operation:

```
xmm1[31-0]        =   (xmm1[31-0] == NAN) ? xmm2[31-0] :
                      (xmm2[31-0] == NAN) ? xmm2[31-0] :
                      (xmm1[31-0] < xmm2/m128[31-0]) : xmm1[31-0] ?
xmm2/m128[31-0];
xmm1[63-32]       =   (xmm1[63-32] == NAN) ? xmm2[63-32] :
                      (xmm2[63-32] == NAN) ? xmm2[63-32] :
                      (xmm1[63-32] < xmm2/m128[63-32]) : xmm1[63-32] ?
xmm2/m128[63-32];
xmm1[95-64]       =   (xmm1[95-64] == NAN) ? xmm2[95-64] :
                      (xmm2[95-64] == NAN) ? xmm2[95-64] :
                      (xmm1[95-64] < xmm2/m128[95-64]) : xmm1[95-64] ?
xmm2/m128[95-64];
xmm1[127-96]      =   (xmm1[127-96] == NAN) ? xmm2[127-96] :
                      (xmm2[127-96] == NAN) ? xmm2[127-96] :
                      (xmm1[127-96] < xmm2/m128[127-96]) : xmm1[127-96] ?
xmm2/m128[127-96];
```

Description: The MINPS instruction returns the minimum SP FP numbers from XMM1 and XMM2/Mem. If the values being compared are both zeros, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an sNaN, this sNaN is forwarded unchanged to the destination (i.e. a quieted version of the sNaN is not returned).

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Invalid (including qNaN source operand), Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 7-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. This approach for MINPS allows compilers to use the MINPS instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MINSS: Scalar Single-FP Minimum

Opcode	Instruction	Description
F3,0F,5D, /r	MINSS xmm1, xmm2/m32	Return the minimum SP FP number between the lowest SP FP numbers from XMM2/Mem and XMM1.

Operation:

```
xmm1[31-0] = (xmm1[31-0] == NAN) ? xmm2[31-0] :
             (xmm2[31-0] == NAN) ? xmm2[31-0] :
             (xmm1[31-0] < xmm2/m32[31-0]) ? xmm1[31-0] : xmm2/m32[31-0];
xmm1[63-32]  = xmm1[63-32];
xmm1[95-64]  = xmm1[95-64];
xmm1[127-96] = xmm1[127-96];
```

Description: The MINSS instruction returns the minimum SP FP number from the lower SP FP numbers from XMM1 and XMM2/Mem; the upper 3 fields are passed through from xmm1.If the values being compared are both zeros, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an sNaN, this sNaN is forwarded unchanged to the destination (i.e. a quieted version of the sNaN is not returned).

FP Exceptions: None

Numeric Exceptions: Invalid (including qNaN source operand), Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory references.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault

IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 7-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. The upper three operands are still bypassed from the src1 operand, as in all other scalar operations. This approach for MINSS allows compilers to use the MINSS instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MOVAPS: Move Aligned Four Packed Single-FP

Opcode	Instruction	Description
0F,28,/r	MOVAPS xmm1, xmm2/m128	Move 128 bits representing 4 packed SP data from XMM2/Mem to XMM1 register.
0F,29,/r	MOVAPS xmm2/m128, xmm1	Move 128 bits representing 4 packed SP from XMM1 register to XMM2/Mem.

Operation:

```
if (destination == xmm1) {
    if (source == m128) {
        // load instruction
        xmm1[127-0] = m128;
    }
    else {
        // move instruction
        xmm1[127=0] = xmm2[127-0];
    }
}
else {
    if (destination == m128) {
        // store instruction
        m128 = xmm1[127-0];
    }
    else {
        // move instruction
        xmm2[127-0] = xmm1[127-0];
    }
}
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded or stored. When the register-register form of this operation is used, the content of the 128-bit source register is copied into 128-bit destination register.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: MOVAPS should be used when dealing with 16-byte aligned SP FP numbers. If the data is not known to be aligned, MOVUPS should be used instead of MOVAPS. The usage of this instruction should be limited to the cases where the aligned restriction is easy to meet. Processors that support Streaming SIMD Extension will provide optimal aligned performance for the MOVAPS instruction.

The usage of Repeat Prefixes (F2H, F3H) with MOVAPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVAPS risks incompatibility with future processors.

MOVHLPS: Move High to Low Packed Single-FP

Opcode	Instruction	Description
0F,12,/r	MOVHLPS xmm1, xmm2	Move 64 bits representing higher two SP operands from XMM2 to lower two fields of XMM1 register.

Operation:

```
// move instruction

xmm1[127-64] = xmm1[127-64];

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

Description: The upper 64-bits of the source register xmm2 are loaded into the lower 64-bits of the 128-bit register xmm1 and the upper 64-bits of xmm1 are left unchanged.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1

Comments: The usage of Repeat (F2H, F3H) and Operand Size (66H) prefixes with MOVHLPS is reserved. Different processor implementations may handle these prefixes differently. Usage of these prefixes with MOVHLPS risks incompatibility with future processors.

MOVHPS: Move High Packed Single-FP

Opcode	Instruction	Description
0F,16,/r	MOVHPS xmm, m64	Move 64 bits representing two SP operands from Mem to upper two fields of XMM register.
0F,17,/r	MOVHPS m64, xmm	Move 64 bits representing two SP operands from upper two fields of XMM register to Mem.

Operation:

```
if (destination == xmm) {
    // load instruction
    xmm[127-64]    = m64;
    xmm[31-0]   = xmm[31-0];
    xmm[63-32] = xmm[63-32];
}
else {
    // store instruction
    m64 = xmm[127-64];
}
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. When the load form of this operation is used, m64 is loaded into the upper 64-bits of the 128-bit register xmm and the lower 64-bits are left unchanged.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with MOVHPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVHPS risks incompatibility with future processors.

MOVLHPS: Move Low to High Packed Single-FP

Opcode	Instruction	Description
0F,16,/r	MOVLHPS xmm1, xmm2	Move 64 bits representing lower two SP operands from XMM2 to upper two fields of XMM1 register.

Operation:

```
// move instruction

xmm1[127-64] = xmm2[63-0];

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

Description: The lower 64-bits of the source register xmm2 are loaded into the upper 64-bits of the 128-bit register xmm1 and the lower 64-bits of xmm1 are left unchanged.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1

Comments:

Example: The usage of Repeat (F2H, F3H) and Operand Size (66H) prefixes with MOVLHPS is reserved. Different processor implementations may handle these prefixes differently. Usage of these prefixes with MOVLHPS risks incompatibility with future processors.

MOVLPS: Move Low Packed Single-FP

Opcode	Instruction	Description
0F,12,/r	MOVLPS xmm, m64	Move 64 bits representing two SP operands from Mem to lower two fields of XMM register.
0F,13,/r	MOVLPS m64, xmm	Move 64 bits representing two SP operands from lower two fields of XMM register to Mem.

Operation:

```
if (destination == xmm) {
    // load instruction
    xmm[63-0]   = m64;
    xmm[95-64]  = xmm[95-64];
    xmm[127-96] = xmm[127-96];
}
else {
    // store instruction
    m64 = xmm[63-0];
}
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. When the load form of this operation is used, m64 is loaded into the lower 64-bits of the 128-bit register xmm and the upper 64-bits are left unchanged.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with MOVLPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVLPS risks incompatibility with future processors.

MOVMSKPS: Move Mask to Integer

Opcode	Instruction	Description
0F,50,/r	MOVMSKPS r32, xmm	Move the single mask to r32.

Operation:

```
r32[3] = xmm[127]; r32[2] = xmm[95];

r32[1] = xmm[63];  r32[0] = xmm[31];

r32[7-4] = 0x0; r32[15-8] = 0x00;

r32[31-16] = 0x0000;
```

Description: The MOVMSKPS instruction returns to the integer register r32 a 4-bit mask formed of the most significant bits of each SP FP number of its operand.

FP Exceptions: None

Numeric Exceptions: None.

Protected Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set.; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with MOVMSKPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVMSKPS risks incompatibility with future processors.

MOVSS: Move Scalar Single-FP

Opcode	Instruction	Description
F3,0F,10,/r	MOVSS xmm1, xmm2/m32	Move 32 bits representing one scalar SP operand from XMM2/Mem to XMM1 register.
F3,0F,11,/r	MOVSS xmm2/m32, xmm1	Move 32 bits representing one scalar SP operand from XMM1 register to XMM2/Mem.

Operation:

```
if (destination == xmm1) {
    if (source == m32) {
        // load instruction
        xmm1[31-0]   = m32;
        xmm1[63-32]  = 0x00000000;
        xmm1[95-64]  = 0x00000000;
        xmm1[127-96] = 0x00000000;
    }
    else {
        // move instruction
        xmm1[31-0]   = xmm2[31-0];
        xmm1[63-32]  = xmm1[63-32];
        xmm1[95-64]  = xmm1[95-64];
        xmm1[127-96] = xmm1[127-96];
    }
}
else {
    if (destination == m32) {
        // store instruction
        m32 = xmm1[31-0];
    }
    else {
        // move instruction
        xmm2[31-0]   = xmm1[31-0]
        xmm2[63-32]  = xmm2[63-32];
```

```
            xmm2[95-64]   = xmm2[95-64];
            xmm2[127-96] = xmm2[127-96];
        }
    }
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 4 bytes of data at memory location m32 are loaded or stored. When the load form of this operation is used, the 32-bits from memory are copied into the lower 32 bits of the 128-bit register xmm, the 96 most significant bits being cleared.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

MOVUPS: Move Unaligned Four Packed Single-FP

Opcode	Instruction	Description
0F,10,/r	MOVUPS xmm1, xmm2/m128	Move 128 bits representing four SP data from XMM2/Mem to XMM1 register.
0F,11,/r	MOVUPS xmm2/m128, xmm1	Move 128 bits representing four SP data from XMM1 register to XMM2/Mem.

Operation:

```
if (destination == xmm1) {
    if (source == m128) {
        // load instruction
        xmm1[127-0] = m128;
    }
    else {
        // move instruction
        xmm1[127-0] = xmm2[127-0];
    }
}
else {
    if (destination == m128) {
        // store instruction
        m128 = xmm1[127-0];
    }
    else {
        // move instruction
        xmm2[127-0] = xmm1[127-0];
    }
}
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded to the 128-bit multimedia register xmm or stored from the 128-bit multimedia register xmm. When the register-register form of this operation is used, the content of the 128-bit source register is copied into 128-bit register xmm. No assumption is made about alignment.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #AC for unaligned memory reference if the current privilege level is 3; #NM if TS bit in CR0 is set.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: MOVUPS should be used with SP FP numbers when that data is known to be unaligned.The usage of this instruction should be limited to the cases where the aligned restriction is hard or impossible to meet. Streaming SIMD Extension implementations guarantee optimum unaligned support for MOVUPS. Efficient Streaming SIMD Extension applications should mainly rely on MOVAPS, not MOVUPS, when dealing with aligned data.

The usage of Repeat-NE Prefix (F2H) and Operand Size Prefix (66H) with MOVUPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVUPS risks incompatibility with future processors.

A linear address of the 128 bit data access, while executing in 16-bit mode, that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. Different processor implementations may/may not raise a GP fault in this case if the segment limit has been exceeded; additionally, the address that spans the end of the segment may/may not wrap around to the beginning of the segment.

MULPS: Packed Single-FP Multiply

Opcode	Instruction	Description
0F,59,/r	MULPS xmm1, xmm2/m128	Multiply packed SP FP numbers in XMM2/Mem to XMM1.

Operation:

```
xmm1[31-0]   = xmm1[31-0]   * xmm2/m128[31-0];

xmm1[63-32]  = xmm1[63-32]  * xmm2/m128[63-32];

xmm1[95-64]  = xmm1[95-64]  * xmm2/m128[95-64];

xmm1[127-96] = xmm1[127-96] * xmm2/m128[127-96];
```

Description: The MULPS instructions multiply the packed SP FP numbers of both their operands.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0).

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

MULSS: Scalar Single-FP Multiply

Opcode	Instruction	Description
F3,0F,59,/r	MULSS xmm1 xmm2/m32	Multiply the lowest SP FP number in XMM2/Mem to XMM1.

```
xmm1[31-0]   = xmm1[31-0] * xmm2/m32[31-0];

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The MULSS instructions multiply the lowest SP FP numbers of both their operands; the upper 3 fields are passed through from xmm1.

FP Exceptions: None

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0).

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

ORPS: Bit-wise Logical OR for Single-FP Data

Opcode	Instruction	Description
0F,56,/r	ORPS xmm1, xmm2/m128	OR 128 bits from XMM2/Mem to XMM1 register.

Operation: `xmm1[127-0] |= xmm2/m128[127-0];`

Description: The ORPS instructions return a bit-wise logical OR between xmm1 and xmm2/mem.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with ORPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with ORPS risks incompatibility with future processors.

RCPPS: Packed Single-FP Reciprocal

Opcode	Instruction	Description
0F,53,/r	RCPPS xmm1, xmm2/m128	Return a packed approximation of the reciprocal of XMM2/Mem.

Operation:

```
xmm1[31-0]   = approx (1.0/(xmm2/m128[31-0]));

xmm1[63-32]  = approx (1.0/(xmm2/m128[63-32]));

xmm1[95-64]  = approx (1.0/(xmm2/m128[95-64]));

xmm1[127-96] = approx (1.0/(xmm2/m128[127-96]));
```

Description: RCPPS returns an approximation of the reciprocal of the SP FP numbers from xmm2/m128. The maximum error for this approximation is:

$|Error| <= 1.5x2^{-12}$

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: RCPPS is not affected by the rounding control in MXCSR. Denormal inputs are treated as zeros (of the same sign) and underflow results are always flushed to zero, with the sign of the operand.

RCPSS: Scalar Single-FP Reciprocal

Opcode	Instruction	Description
F3,0F,5 3,/r	RCPSS xmm1, xmm2/m32	Return an approximation of the reciprocal of the lower SP FP number in XMM2/Mem.

Operation:

```
xmm1[31-0]   = approx (1.0/(xmm2/m32[31-0]));

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: RCPSS returns an approximation of the reciprocal of the lower SP FP number from xmm2/m32; the upper 3 fields are passed through from xmm1. The maximum error for this approximation is:

$|Error| <= 1.5x2^{-12}$

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #AC for unaligned memory reference if the current privilege level is 3; #NM if TS bit in CR0 is set.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: RCPSS is not affected by the rounding control in MXCSR. Denormal inputs are treated as zeros (of the same sign) and underflow results are always flushed to zero, with the sign of the operand.

RSQRTPS: Packed Single-FP Square Root Reciprocal

Opcode	Instruction	Description
0F,52,/r	RSQRTPS xmm1, xmm2/m128	Return a packed approximation of the square root of the reciprocal of XMM2/Mem.

Operation:

```
xmm1[31-0]   = approx (1.0/sqrt(xmm2/m128[31-0]));

xmm1[63-32]  = approx (1.0/sqrt(xmm2/m128[63-32]));

xmm1[95-64]  = approx (1.0/sqrt(xmm2/m128[95-64]));

xmm1[127-96] = approx (1.0/sqrt(xmm2/m128[127-96]));
```

Description: RSQRTPS returns an approximation of the reciprocal of the square root of the SP FP numbers from xmm2/m128. The maximum error for this approximation is:

$|Error| <= 1.5x2^{-12}$

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: RSQRTPS is not affected by the rounding control in MXCSR. Denormal inputs are treated as zeros (of the same sign) and underflow results are always flushed to zero, with the sign of the operand.

RSQRTSS: Scalar Single-FP Square Root Reciprocal

Opcode	Instruction	Description
F3,0F,52,/r	RSQRTSS xmm1, xmm2/m32	Return an approximation of the square root of the reciprocal of the lowest SP FP number in XMM2/Mem.

Operation:

```
xmm1[31-0]   = approx (1.0/sqrt(xmm2/m32[31-0]));

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: RSQRTSS returns an approximation of the reciprocal of the square root of the lowest SP FP number from xmm2/m32; the upper 3 fields are passed through from xmm1. The maximum error for this approximation is:

$|Error| <= 1.5x2^{-12}$

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments:

Example: RSQRTSS is not affected by the rounding control in MXCSR. Denormal inputs are treated as zeros (of the same sign) and underflow results are always flushed to zero, with the sign of the operand.

SHUFPS: Shuffle Single-FP

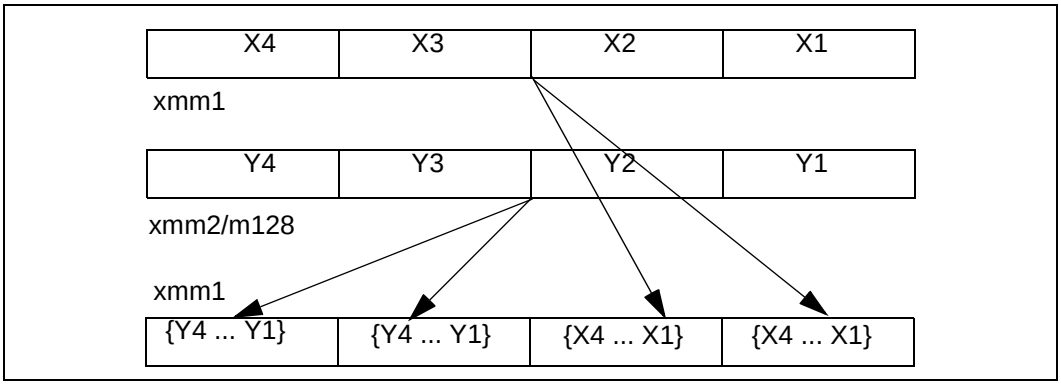
Opcode	Instruction	Description
0F,C6,/r, ib	SHUFPS xmm1, xmm2/m128, imm8	Shuffle Single.

Operation:

```
fp_select  = (imm8 >> 0) & 0x3;
xmm1[31-0]   = (fp_select == 0) ? xmm1[31-0]   :
               (fp_select == 1) ? xmm1[63-32]  :
               (fp_select == 2) ? xmm1[95-64]  :
                                  xmm1[127-96];
fp_select  = (imm8 >> 2) & 0x3;
xmm1[63-32]  = (fp_select == 0) ? xmm1[31-0]   :
               (fp_select == 1) ? xmm1[63-32]  :
               (fp_select == 2) ? xmm1[95-64]  :
                                  xmm1[127-96];
fp_select  = (imm8 >> 4) & 0x3;
xmm1[95-64]  = (fp_select == 0) ? xmm2/m128[31-0]   :
               (fp_select == 1) ? xmm2/m128[63-32]  :
               (fp_select == 2) ? xmm2/m128[95-64]  :
                                  xmm2/m128[127-96];
fp_select  = (imm8 >> 6) & 0x3;
xmm1[127-96] = (fp_select == 0) ? xmm2/m128[31-0]   :
               (fp_select == 1) ? xmm2/m128[63-32]  :
               (fp_select == 2) ? xmm2/m128[95-64]  :
                                  xmm2/m128[127-96];
```

Description: The SHUFPS instruction is able to shuffle any of the four SP FP numbers from xmm1 to the lower 2 destination fields; the upper 2 destination fields are generated from a shuffle of any of the four SP FP numbers from xmm2/m128. By using the same register for both sources, SHUFPS can return any combination of the four SP FP numbers from this register. Bits 0 and 1 of the immediate field are used to select which of the four input SP FP numbers will be put in the first SP FP number of the result; bits 3 and 2 of the immediate field are used to select which of the four input SP FP will be put in the second SP FP number of the result; etc.

Example:



FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with SHUFPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with SHUFPS risks incompatibility with future processors.

SQRTPS: Packed Single-FP Square Root

Opcode	Instruction	Description
0F,51,/r	SQRTPS xmm1, xmm2/m128	Square Root of the packed SP FP numbers in XMM2/Mem.

Operation:

```
xmm1[31-0]   = sqrt (xmm2/m128[31-0]);

xmm1[63-32]  = sqrt (xmm2/m128[63-32]);

xmm1[95-64]  = sqrt (xmm2/m128[95-64]);

xmm1[127-96] = sqrt (xmm2/m128[127-96]);
```

Description: The SQRTPS instruction returns the square root of the packed SP FP numbers from xmm2/m128.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

SQRTSS: Scalar Single-FP Square Root

Opcode	Instruction	Description
F3,0F,51,/r	SQRTSS xmm1, xmm2/m32	Square Root of the lower SP FP number in XMM2/Mem.

Operation:

```
xmm1[31-0]   = sqrt (xmm2/m32[31-0]);

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The SQRTSS instructions return the square root of the lowest SP FP numbers of their operand.

FP Exceptions: None

Numeric Exceptions: Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

STMXCSR: Store Streaming SIMD Extension Control/Status

Opcode	Instruction	Description
0F,AE,/3	STMXCSR m32	Store Streaming SIMD Extension control/status word to m32.

Operation: `m32 = MXCSR;`

Description: The MXCSR control/status register is used to enable masked/unmasked exception handling, to set rounding modes, to set flush-to-zero mode, and to view exception status flags. Refer to LDMXCSR for a description of the format of MXCSR. The linear address corresponds to the address of the least-significant byte of the referenced memory data. The reserved bits in the MXCSR are stored as zeroes.

FP Exceptions: None.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault. #AC for unaligned memory reference.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: The usage of Repeat (F2H, F3H) and Operand Size (66H) prefixes with STMXCSR is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with STMXCSR risks incompatibility with future processors.

SUBPS: Packed Single-FP Subtract

Opcode	Instruction	Description
0F,5C,/r	SUBPS xmm1 xmm2/m128	Subtract packed SP FP numbers in XMM2/Mem from XMM1.

Operation:

```
xmm1[31-0]   = xmm1[31-0]   - xmm2/m128[31-0];

xmm1[63-32]  = xmm1[63-32]  - xmm2/m128[63-32];

xmm1[95-64]  = xmm1[95-64]  - xmm2/m128[95-64];

xmm1[127-96] = xmm1[127-96] - xmm2/m128[127-96];
```

Description: The SUBPS instruction subtracts the packed SP FP numbers of both their operands.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault;.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

SUBSS: Scalar Single-FP Subtract

Opcode	Instruction	Description
F3,0F,5C, /r	SUBSS xmm1, xmm2/m32	Subtract the lower SP FP numbers in XMM2/Mem from XMM1.

Operation:

```
xmm1[31-0]   = xmm1[31-0] - xmm2/m32[31-0];

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The SUBSS instruction subtracts the lower SP FP numbers of both their operands.

FP Exceptions: None.

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

UCOMISS: Unordered Scalar Single-FP Compare and Set EFLAGS

Opcode	Instruction	Description
0F,2E,/r	UCOMISS xmm1, xmm2/m32	Compare lower SP FP number in XMM1 register with lower SP FP number in XMM2/Mem and set the status flags accordingly.

Operation:

```
switch (xmm1[31-0] <> xmm2/m32[31-0]) {

    OF,SF,AF = 000;

    case UNORDERED:     ZF,PF,CF = 111;

    case GREATER_THAN:  ZF,PF,CF = 000;

    case LESS_THAN:     ZF,PF,CF = 001;

    case EQUAL:         ZF,PF,CF = 100;

}
```

Description: The UCOMISS instructions compare the two lowest scalar SP FP numbers and sets the ZF,PF,CF bits in the EFLAGS register as described above. In addition, the OF, SF and AF bits in the EFLAGS register are zeroed out. The unordered predicate is returned if either source operand is a NaN (qNaN or sNaN).

FP Exceptions: None.

Numeric Exceptions: Invalid (if SNaN operands), Denormal. Integer EFLAGS values will not be updated in the presence of unmasked numeric exceptions.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked Streaming SIMD Extension numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: UCOMISS differs from COMISS in that it signals an invalid numeric exception when a source operand is an sNaN; COMISS signals invalid if a source operand is either a qNaN or an sNaN.

The usage of Repeat (F2H, F3H) and Operand-Size prefixes with UCOMISS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with UCOMISS risks incompatibility with future processors.

UNPCKHPS: Unpack High Packed Single-FP Data

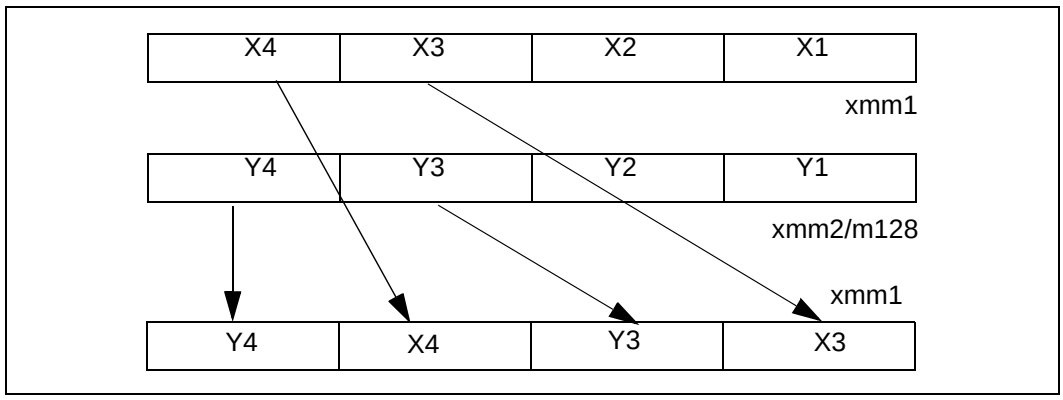
Opcode	Instruction	Description
0F,15,/r	UNPCKHPS xmm1, xmm2/m128	Interleaves SP FP numbers from the high halves of XMM1 and XMM2/Mem into XMM1 register.

Operation:

```
xmm1[31-0]    = xmm1[95-64];

xmm1[63-32]   = xmm2/m128[95-64];

xmm1[95-64]   = xmm1[127-96];

xmm1[127-96]  = xmm2/m128[127-96];
```

Description: The UNPCKHPS instruction performs an interleaved unpack of the high-order data elements of XMM1 and XMM2/Mem. It ignores the lower half of the sources.

Example:



FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: When unpacking from a memory operand, an implementation may decide to fetch only the appropriate 64 bits. Alignment to 16-byte boundary and normal segment checking will still be enforced.

The usage of Repeat Prefixes (F2H, F3H) with UNPCKHPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with UNPCKHPS risks incompatibility with future processors.

UNPCKLPS: Unpack Low Packed Single-FP Data

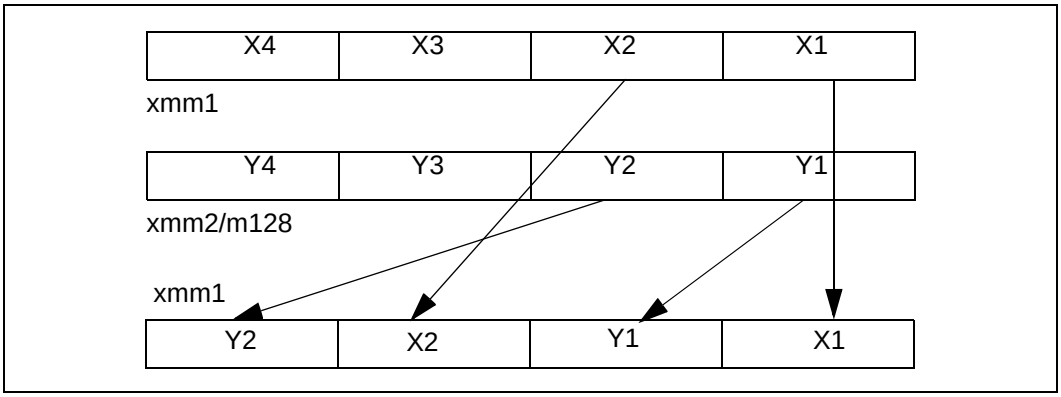
Opcode	Instruction	Description
0F,14,/r	UNPCKLPS xmm1, xmm2/m128	Interleaves SP FP numbers from the low halves of XMM1 and XMM2/Mem into XMM1 register.

Operation:

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

Description: The UNPCKLPS instruction performs an interleaved unpack of the low-order data elements of XMM1 and XMM2/Mem. It ignores the upper half part of the sources.

Example:



FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: When unpacking from a memory operand, an implementation may decide to fetch only the appropriate 64 bits. Alignment to 16-byte boundary and normal segment checking will still be enforced.

The usage of Repeat Prefixes (F2H, F3H) with UNPCKLPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with UNPCKLPS risks incompatibility with future processors.

XORPS: Bit-wise Logical Xor for Single-FP Data

Opcode	Instruction	Description
0F,57,/r	XORPS xmm1, xmm2/m128	XOR 128 bits from XMM2/Mem to XMM1 register.

Operation: `xmm[127-0] ^= xmm/m128[127-0];`

Description: The XORPS instruction returns a bit-wise logical XOR between XMM1 and XMM2/Mem.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments:

The usage of Repeat Prefixes (F2H, F3H) with XORPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with XORPS risks incompatibility with future processors.

7.13 SIMD Integer Instruction Set Extensions

Additional new SIMD Integer instructions have been added to accelerate the performance of 3D graphics, video decoding and encoding and other applications. These instructions operate on the MMX technology registers and on 64-bit memory operands.

PAVGB/PAVGW: Packed Average

Opcode	Instruction	Description
0F,E0, /r	PAVGB mm1,mm2/m64	Average with rounding packed unsigned bytes from MM2/Mem to packed bytes in MM1 register.
0F,E3, /r	PAVGW mm1, mm2/m64	Average with rounding packed unsigned words from MM2/Mem to packed words in MM1 register.

Operation:

```
if (instruction == PAVGB) {
    x[0]   = mm1[7-0]          y[0] = mm2/m64[7-0];
    x[1]   = mm1[15-8]         y[1] = mm2/m64[15-8];
    x[2]   = mm1[23-16]        y[2] = mm2/m64[23-16];
    x[3]   = mm1[31-24]        y[3] = mm2/m64[31-24];
    x[4]   = mm1[39-32]        y[4] = mm2/m64[39-32];
    x[5]   = mm1[47-40]        y[5] = mm2/m64[47-40];
    x[6]   = mm1[55-48]        y[6] = mm2/m64[55-48];
    x[7]   = mm1[63-56]        y[7] = mm2/m64[63-56];

    for (i = 0; i < 8; i++) {
        temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
        res[i] = (temp[i] +1) >> 1;
        }
    mm1[7-0]      =   res[0];
    ...
    mm1[63-56]    =   res[7];
}
else if (instruction == PAVGW){
    x[0]   = mm1[15-0]         y[0] = mm2/m64[15-0];
    x[1]   = mm1[31-16]        y[1] = mm2/m64[31-16];
    x[2]   = mm1[47-32]        y[2] = mm2/m64[47-32];
    x[3]   = mm1[63-48]        y[3] = mm2/m64[63-48];
```

```
    for (i = 0; i < 4; i++) {
        temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
        res[i] = (temp[i] +1) >> 1;
    }
    mm1[15-0]     =   res[0];
    ...
    mm1[63-48]    =   res[3];
}
```

Description: The PAVG instructions add the unsigned data elements of the source operand to the unsigned data elements of the destination register, along with a carry-in. The results of the add are then each independently right shifted by one bit position. The high order bits of each element are filled with the carry bits of the corresponding sum.

The destination operand is a MMX technology register. The source operand can either be a MMX technology register or a 64-bit memory operand.

The PAVGB instruction operates on packed unsigned bytes and the PAVGW instruction operates on packed unsigned words.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory references (if the current privilege level is 3).

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PEXTRW: Extract Word

Opcode	Instruction	Description
0F,C5, /r, ib	PEXTRW r32, mm, imm8	Extract the word pointed to by imm8 from MM and move it to a 32-bit integer register.

Operation:

```
sel = imm8 & 0x3;

mm_temp = (mm >> (sel * 16)) & 0xffff;

r[15-0] = mm_temp[15-0];

r[31-16] = 0x0000;
```

Description: The PEXTRW instruction moves the word in MM selected by the two least significant bits of imm8 to the lower half of a 32-bit integer register.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1

PINSRW: Insert Word

Opcode	Instruction	Description
0F,C4,/r,ib	PINSRW mm, r32/m16, imm8	Insert the word from the lower half of r32 or from Mem16 into the position in MM pointed to by imm8 without touching the other words.

Operation:

```
sel = imm8 & 0x3;

mask = (sel == 0)? 0x000000000000ffff :

       (sel == 1)? 0x00000000ffff0000 :

       (sel == 2)? 0x0000ffff00000000 :

                   0xffff000000000000;

mm = (mm & ~mask) | ((m16/r32[15-0] << (sel * 16)) & mask);
```

Description: The PINSRW instruction loads a word from the lower half of a 32-bit integer register (or from memory) and inserts it in the MM destination register at a position defined by the two least significant bits of the imm8 constant. The insertion is done in such a way that the three other words from the destination register are left untouched.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PMAXSW: Packed Signed Integer Word Maximum

Opcode	Instruction	Description
0F,EE, /r	PMAXSW mm1, mm2/m64	Return the maximum words between MM2/Mem and MM1.

Operation:

```
mm1[15-0]   = (mm1[15-0] > mm2/m64[15-0]) ? mm1[15-0] : mm2/m64[15-0];

mm1[31-16]  = (mm1[31-16] > mm2/m64[31-16]) ? mm1[31-16] : mm2/m64[31-16];

mm1[47-32]  = (mm1[47-32] > mm2/m64[47-32]) ? mm1[47-32] : mm2/m64[47-32];

mm1[63-48]  = (mm1[63-48] > mm2/m64[63-48]) ? mm1[63-48] : mm2/m64[63-48];
```

Description: The PMAXSW instruction returns the maximum between the four signed words in MM1 and MM2/Mem.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception..

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PMAXUB: Packed Unsigned Integer Byte Maximum

Opcode	Instruction	Description
0F,DE, /r	PMAXUB mm1, mm2/m64	Return the maximum bytes between MM2/Mem and MM1.

Operation:

```
mm1[7-0]   = (mm1[7-0] > mm2/m64[7-0]) ? mm1[7-0] : mm2/m64[7-0];

mm1[15-8]  = (mm1[15-8] > mm2/m64[15-8]) ? mm1[15-8] : mm2/m64[15-8];

mm1[23-16]  = (mm1[23-16] > mm2/m64[23-16]) ? mm1[23-16] : mm2/m64[23-16];

mm1[31-24]  = (mm1[31-24] > mm2/m64[31-24]) ? mm1[31-24] : mm2/m64[31-24];

mm1[39-32]  = (mm1[39-32] > mm2/m64[39-32]) ? mm1[39-32] : mm2/m64[39-32];

mm1[47-40]  = (mm1[47-40] > mm2/m64[47-40]) ? mm1[47-40] : mm2/m64[47-40];

mm1[55-48]  = (mm1[55-48] > mm2/m64[55-48]) ? mm1[55-48] : mm2/m64[55-48];

mm1[63-56]  = (mm1[63-56] > mm2/m64[63-56]) ? mm1[63-56] : mm2/m64[63-56];
```

Description: The PMAXUB instruction returns the maximum between the eight unsigned words in MM1 and MM2/Mem.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PMINSW: Packed Signed Integer Word Minimum

Opcode	Instruction	Description
0F,EA, /r	PMINSW mm1, mm2/m64	Return the minimum words between MM2/Mem and MM1.

Operation:

```
mm1[15-0]   = (mm1[15-0] < mm2/m64[15-0]) ? mm1[15-0] : mm2/m64[15-0];

mm1[31-16]  = (mm1[31-16] < mm2/m64[31-16]) ? mm1[31-16] : mm2/m64[31-16];

mm1[47-32]  = (mm1[47-32] < mm2/m64[47-32]) ? mm1[47-32] : mm2/m64[47-32];

mm1[63-48]  = (mm1[63-48] < mm2/m64[63-48]) ? mm1[63-48] : mm2/m64[63-48];
```

Description: The PMINSW instruction returns the minimum between the four signed words in MM1 and MM2/Mem.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception#AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PMINUB: Packed Unsigned Integer Byte Minimum

Opcode	Instruction	Description
0F,DA, /r	PMINUB mm1, mm2/m64	Return the minimum bytes between MM2/Mem and MM1.

Operation:

```
mm1[7-0]   = (mm1[7-0] < mm2/m64[7-0]) ? mm1[7-0] : mm2/m64[7-0];

mm1[15-8]  = (mm1[15-8] < mm2/m64[15-8]) ? mm1[15-8] : mm2/m64[15-8];

mm1[23-16]  = (mm1[23-16] < mm2/m64[23-16]) ? mm1[23-16] : mm2/m64[23-16];

mm1[31-24]  = (mm1[31-24] < mm2/m64[31-24]) ? mm1[31-24] : mm2/m64[31-24];

mm1[39-32]  = (mm1[39-32] < mm2/m64[39-32]) ? mm1[39-32] : mm2/m64[39-32];

mm1[47-40]  = (mm1[47-40] < mm2/m64[47-40]) ? mm1[47-40] : mm2/m64[47-40];

mm1[55-48]  = (mm1[55-48] < mm2/m64[55-48]) ? mm1[55-48] : mm2/m64[55-48];

mm1[63-56]  = (mm1[63-56] < mm2/m64[63-56]) ? mm1[63-56] : mm2/m64[63-56];
```

Description: The PMINUB instruction returns the minimum between the eight unsigned words in MM1 and MM2/Mem.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PMOVMSKB: Move Byte Mask To Integer

Opcode	Instruction	Description
0F,D7,/r	PMOVMSKB r32, mm	Move the byte mask of MM to r32.

Operation:

```
r32[7]  = mm[63];  r32[6]  = mm[55];

r32[5]  = mm[47];  r32[4]  = mm[39];

r32[3]  = mm[31];  r32[2]  = mm[23];

r32[1]  = mm[15];  r32[0]  = mm[7];

r32[31-8] = 0x000000;
```

Description: The PMOVMSKB instruction returns a 8-bit mask formed of the most significant bits of each byte of its source operand.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults Disabled FP Register Fault if PSR.dfl is 1

PMULHUW: Packed Multiply High Unsigned

Opcode	Instruction	Description
0F,E4,/r	PMULHUW mm1, mm2/m64	Multiply the packed unsigned words in MM1 register with the packed unsigned words in MM2/Mem, then store the high-order 16 bits of the results in MM1.

Operation:

```
mm1[15-0]    = (mm1[15-0]    * mm2/m64[15-0])[31-16];

mm1[31-16]   = (mm1[31-16]   * mm2/m64[31-16])[31-16];

mm1[47-32]   = (mm1[47-32]   * mm2/m64[47-32])[31-16];

mm1[63-48]   = (mm1[63-48]   * mm2/m64[63-48])[31-16];
```

Description: The PMULHUW instruction multiplies the four unsigned words in the destination operand with the four unsigned words in the source operand. The high-order 16 bits of the 32-bit intermediate results are written to the destination operand.

Numeric Exceptions: None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PSADBW: Packed Sum of Absolute Differences

Opcode	Instruction	Description
0F,F6, /r	PSADBW mm1,mm2/m64	Absolute difference of packed unsigned bytes from MM2 /Mem and MM1; these differences are then summed to produce a word result.

Operation:

```
temp1   = ABS(mm1[7-0]      - mm2/m64[7-0]);

temp2   = ABS(mm1[15-8]     - mm2/m64[15-8]);

temp3   = ABS(mm1[23-16]    - mm2/m64[23-16]);

temp4   = ABS(mm1[31-24]    - mm2/m64[31-24]);

temp5   = ABS(mm1[39-32]    - mm2/m64[39-32]);

temp6   = ABS(mm1[47-40]    - mm2/m64[47-40]);

temp7   = ABS(mm1[55-48]    - mm2/m64[55-48]);

temp8   = ABS(mm1[63-56]    - mm2/m64[63-56]);


mm1[15:0] = temp1 + temp2 + temp3 + temp4 + temp5 + temp6 + temp7 + temp8;

mm1[31:16] = 0x00000000;

mm1[47:32] = 0x00000000;

mm1[63:48] = 0x00000000;
```

Description: The PSADBW instruction computes the absolute value of the difference of unsigned bytes for mm1 and mm2/m64. These differences are then summed to produce a word result in the lower 16-bit field; the upper 3 words are cleared.

The destination operand is a MMX technology register. The source operand can either be a MMX technology register or a 64-bit memory operand.

Numeric Exceptions: None

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PSHUFW: Packed Shuffle Word

Opcode	Instruction	Description
0F,70,/r,ib	PSHUFW mm1, mm2/m64, imm8	Shuffle the words in MM2/Mem based on the encoding in imm8 and store in MM1.

Operation:

```
mm1[15-0]   = (mm2/m64 >> (imm8[1-0] * 16) )[15-0]

mm1[31-16]  = (mm2/m64 >> (imm8[3-2] * 16) )[15-0]

mm1[47-32]  = (mm2/m64 >> (imm8[5-4] * 16) )[15-0]

mm1[63-48]  = (mm2/m64 >> (imm8[7-6] * 16) )[15-0]
```

Description: The PSHUF instruction uses the imm8 operand to select which of the four words in MM2/Mem will be placed in each of the words in MM1. Bits 1 and 0 of imm8 encode the source for destination word 0 (MM1[15-0]), bits 3 and 2 encode for word 1, bits 5 and 4 encode for word 2, and bits 7 and 6 encode for word 3 (MM1[63-48]). Similarly, the two bit encoding represents which source word is to be used, e.g. an binary encoding of 10 indicates that source word 2 (MM2/Mem[47-32]) will be used.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

7.14 Cacheability Control Instructions

This section describes the cacheability control instructions which enable an application writer to minimize data access latency and cache pollution.

MASKMOVQ: Byte Mask Write

Opcode	Instruction	Description
0F,F7,/r	MASKMOVQ mm1, mm2	Move 64-bits representing integer data from MM1 register to memory location specified by the edi register, using the byte mask in MM2 register.

Operation:

```
if (mm2[7])   m64[edi]    = mm1[7-0];
if (mm2[15])  m64[edi+1]  = mm1[15-8];
if (mm2[23])  m64[edi+2]  = mm1[23-16];
if (mm2[31])  m64[edi+3]  = mm1[31-24];
if (mm2[39])  m64[edi+4]  = mm1[39-32];
if (mm2[47])  m64[edi+5]  = mm1[47-40];
if (mm2[55])  m64[edi+6]  = mm1[55-48];
if (mm2[63])  m64[edi+7]  = mm1[63-56];
```

Description: Data is stored from the mm1 register to the location specified by the di/edi register (using DS segment). The size of the store address depends on the address-size attribute. The most significant bit in each byte of the mask register mm2 is used to selectively write the data (0 = no write, 1 = write), on a per-byte basis. Behavior with a mask of all zeroes is as follows:

- No data will be written to memory. However, transition from FP to MMX technology state (if necessary) will occur, irrespective of the value of the mask.
- For memory references, a zero byte mask does not prevent addressing faults (i.e. #GP, #SS) from being signalled.
- Signalling of page faults (#PF) is implementation-specific.
- #UD, #NM, #MF, and #AC faults are signalled irrespective of the value of the mask.
- Signalling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (i.e. is reserved) and is implementation-specific. Dependency on the behavior of a specific implementation in this case is not recommended, and may lead to future incompatibility.

The Mod field of the ModR/M byte must be 11, or an Invalid Opcode Exception will result.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1

Comments: MASKMOVQ can be used to improve performance for algorithms which need to merge data on a byte granularity.MASKMOVQ should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store. Similar to the Streaming SIMD Extension non-temporal store instructions, MASKMOVQ minimizes pollution of the cache hierarchy. MASKMOVQ implicitly uses weakly-ordered, write-combining stores (WC). See Section 7.6.1.9 for further information about non-temporal stores.

As a consequence of the resulting weakly-ordered memory consistency model, a fencing operation such as SFENCE should be used if multiple processors may use different memory types to read/write the same memory location specified by edi.

This instruction behaves identically to MMX instructions, in the presence of x87-FP instructions: transition from x87-FP to MMX technology (TOS=0, FP valid bits set to all valid).

MASMOVQ ignores the value of CR4.OSFXSR. Since it does not affect the new Streaming SIMD Extension state, they will not generate an invalid exception if CR4.OSFXSR = 0.

MOVNTPS: Move Aligned Four Packed Single-FP Non-temporal

Opcode	Instruction	Description
0F,2B, /r	MOVNTPS m128, xmm	Move 128 bits representing four packed SP FP data from XMM register to Mem, minimizing pollution in the cache hierarchy.

Operation: `m128 = xmm;`

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. This store instruction minimizes cache pollution.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: MOVTNPS should be used when dealing with 16-byte aligned single-precision FP numbers. MOVNTPS minimizes pollution in the cache hierarchy. As a consequence of the resulting weakly-ordered memory consistency model, a fencing operation should be used if multiple processors may use different memory types to read/write the memory location. See Section 7.6.1.9 for further information about non-temporal stores.

The usage of Repeat Prefixes(F2H, F3H) with MOVNTPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVNTPS risks incompatibility with future processors.

MOVNTQ: Move 64 Bits Non-temporal

Opcode	Instruction	Description
0F,E7,/r	MOVNTQ m64, mm	Move 64 bits representing integer operands (8b, 16b, 32b) from MM register to memory, minimizing pollution within cache hierarchy.

Operation:

```
m64 = mm;
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. This store instruction minimizes cache pollution.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional IA-64 System Environment Exceptions

IA-64 Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
IA-64 Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: MOVNTQ minimizes pollution in the cache hierarchy. As a consequence of the resulting weakly-ordered memory consistency model, a fencing operation should be used if multiple processors may use different memory types to read/write the memory location. See Section 7.6.1.9 for further information about non-temporal stores.

MOVNTQ ignores the value of CR4.OSFXSR. Since it does not affect the new Streaming SIMD Extension state, they will not generate an invalid exception if CR4.OSFXSR = 0.

PREFETCH: Prefetch

Opcode	Instruction	Description
0F,18,/1	PREFETCHT0 m8	Move data specified by address closer to the processor using the t0 hint.
0F,18,/2	PREFETCHT1 m8	Move data specified by address closer to the processor using the t1 hint.
0F,18,/3	PREFETCHT2 m8	Move data specified by address closer to the processor using the t2 hint.
0F,18,/0	PREFETCHNTA m8	Move data specified by address closer to the processor using the nta hint.

Operation:

```
fetch (m8);
```

Description: If there are no excepting conditions, the prefetch instruction fetches the line containing the addresses byte to a location in the cache hierarchy specified by a locality hint. If the line is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. The bits 5:3 of the ModR/M byte specify locality hints as follows:

- Temporal data(t0) - prefetch data into all cache levels.
- Temporal with respect to first level cache (t1) – prefetch data in all cache levels except 0th cache level.
- Temporal with respect to second level cache (t2) – prefetch data in all cache levels, except 0th and 1st cache levels.
- Non-temporal with respect to all cache levels (nta) – prefetch data into non-temporal cache structure.

Locality hints do not affect the functional behavior of the program. They are implementation dependent, and can be overloaded or ignored by an implementation. The prefetch instruction does not cause any exceptions (except for code breakpoints), does not affect program behavior and may be ignored by the implementation. The amount of data prefetched is implementation dependent. It will however be a minimum of 32 bytes. Prefetches to uncacheable memory (UC or WC memory types) will be ignored. Additional ModRM encodings, besides those specified above, are defined to be reserved and the use of reserved encodings risks future incompatibility.

Numeric Exceptions: None

Protected Mode Exceptions: None

Real Address Mode Exceptions: None

Virtual 8086 Mode Exceptions: None

Additional IA-64 System Environment Exceptions: None

Comments: This instruction is merely a hint.If executed, this instruction moves data closer to the processor in anticipation of future use. The performance of these instructions in application code can be implementation specific. To achieve maximum speedup, code tuning might be necessary for each implementation. The non temporal hint also minimizes pollution of useful cache data.

PREFETCH instructions ignore the value of CR4.OSFXSR. Since they do not affect the new Streaming SIMD Extension state, they will not generate an invalid exception if CR4.OSFXSR = 0.

SFENCE: Store Fence

Opcode	Instruction	Description
0F AE /7	SFENCE	Guarantees that every store instruction that precedes in program order the store fence instruction is globally visible before any store instruction which follows the fence is globally visible.

Operation:

```
while (!(preceding_stores_globally_visible)) wait();
```

Description: Weakly ordered memory types can enable higher performance through such techniques as out-of-order issue, write-combining, and write-collapsing. Memory ordering issues can arise between a producer and a consumer of data and there are a number of common usage models which may be affected by weakly ordered stores: (1) library functions, which use weakly ordered memory to write results (2) compiler-generated code, which also benefit from writing weakly-ordered results, and (3) hand-written code. The degree to which a consumer of data knows that the data is weakly ordered can vary for these cases. As a result, the SFENCE instruction provides a performance-efficient way of ensuring ordering between routines that produce weakly-ordered results and routines that consume this data.

SFENCE uses the following ModRM encoding:

Mod (7:6) = 11B

Reg/Opcode (5:3) = 111B

R/M (2:0) = 000B

All other ModRM encodings are defined to be reserved, and use of these encodings risks incompatibility with future processors.

Numeric Exceptions: None

Protected Mode Exceptions: None

Real Address Mode Exceptions: None

Virtual 8086 Mode Exceptions: None

Additional IA-64 System Environment Exceptions: None

Comments: SFENCE ignores the value of CR4.OSFXSR. SFENCE will not generate an invalid exception if CR4.OSFXSR = 0