

# Intel® Itanium® Architecture Software Developer's Manual

Revision 2.3

Volume 3: Intel® Itanium® Instruction Set





# **Intel® Itanium® Architecture Software Developer's Manual**

**Volume 3: Intel® Itanium® Instruction Set Reference**

**Revision 2.3**

***May 2010***

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, 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® processors based on the Itanium architecture 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.

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

Copies of documents which have an 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://www.intel.com>.

Intel, Itanium, Pentium, VTune and MMX are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Copyright © 1999-2010, Intel Corporation

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

# Contents

---

<b>1</b>	<b>About this Manual . . . . .</b>	<b>3:1</b>
1.1	Overview of Volume 1: Application Architecture . . . . .	3:1
1.1.1	Part 1: Application Architecture Guide . . . . .	3:1
1.1.2	Part 2: Optimization Guide for the Intel® Itanium® Architecture . . . . .	3:1
1.2	Overview of Volume 2: System Architecture . . . . .	3:2
1.2.1	Part 1: System Architecture Guide . . . . .	3:2
1.2.2	Part 2: System Programmer's Guide . . . . .	3:3
1.2.3	Appendices . . . . .	3:4
1.3	Overview of Volume 3: Intel® Itanium® Instruction Set Reference . . . . .	3:4
1.4	Overview of Volume 4: IA-32 Instruction Set Reference . . . . .	3:4
1.5	Terminology . . . . .	3:5
1.6	Related Documents . . . . .	3:5
1.7	Revision History . . . . .	3:6
<b>2</b>	<b>Instruction Reference . . . . .</b>	<b>3:11</b>
2.1	Instruction Page Conventions . . . . .	3:11
2.2	Instruction Descriptions . . . . .	3:13
<b>3</b>	<b>Pseudo-Code Functions . . . . .</b>	<b>3:281</b>
<b>4</b>	<b>Instruction Formats . . . . .</b>	<b>3:293</b>
4.1	Format Summary . . . . .	3:294
4.2	A-Unit Instruction Encodings . . . . .	3:300
4.2.1	Integer ALU . . . . .	3:300
4.2.2	Integer Compare . . . . .	3:302
4.2.3	Multimedia . . . . .	3:306
4.3	I-Unit Instruction Encodings . . . . .	3:310
4.3.1	Multimedia and Variable Shifts . . . . .	3:310
4.3.2	Integer Shifts . . . . .	3:315
4.3.3	Test Bit . . . . .	3:316
4.3.4	Miscellaneous I-Unit Instructions . . . . .	3:318
4.3.5	GR/BR Moves . . . . .	3:320
4.3.6	GR/Predicate/IP Moves . . . . .	3:321
4.3.7	GR/AR Moves (I-Unit) . . . . .	3:321
4.3.8	Sign/Zero Extend/Compute Zero Index . . . . .	3:322
4.3.9	Test Feature . . . . .	3:323
4.4	M-Unit Instruction Encodings . . . . .	3:323
4.4.1	Loads and Stores . . . . .	3:323
4.4.2	Line Prefetch . . . . .	3:337
4.4.3	Semaphores . . . . .	3:338
4.4.4	Set/Get FR . . . . .	3:339
4.4.5	Speculation and Advanced Load Checks . . . . .	3:340
4.4.6	Cache/Synchronization/RSE/ALAT . . . . .	3:341
4.4.7	GR/AR Moves (M-Unit) . . . . .	3:342
4.4.8	GR/CR Moves . . . . .	3:343
4.4.9	Miscellaneous M-Unit Instructions . . . . .	3:344
4.4.10	System/Memory Management . . . . .	3:345
4.4.11	Nop/Hint (M-Unit) . . . . .	3:349
4.5	B-Unit Instruction Encodings . . . . .	3:349
4.5.1	Branches . . . . .	3:350
4.5.2	Branch Predict/Nop/Hint . . . . .	3:353
4.5.3	Miscellaneous B-Unit Instructions . . . . .	3:355
4.6	F-Unit Instruction Encodings . . . . .	3:356
4.6.1	Arithmetic . . . . .	3:358

4.6.2	Parallel Floating-point Select . . . . .	3:359
4.6.3	Compare and Classify . . . . .	3:359
4.6.4	Approximation . . . . .	3:361
4.6.5	Minimum/Maximum and Parallel Compare . . . . .	3:362
4.6.6	Merge and Logical . . . . .	3:363
4.6.7	Conversion . . . . .	3:363
4.6.8	Status Field Manipulation . . . . .	3:364
4.6.9	Miscellaneous F-Unit Instructions . . . . .	3:365
4.7	X-Unit Instruction Encodings . . . . .	3:365
4.7.1	Miscellaneous X-Unit Instructions . . . . .	3:365
4.7.2	Move Long Immediate <sub>64</sub> . . . . .	3:366
4.7.3	Long Branches . . . . .	3:367
4.7.4	Nop/Hint (X-Unit) . . . . .	3:368
4.8	Immediate Formation . . . . .	3:368
<b>5</b>	<b>Resource and Dependency Semantics . . . . .</b>	<b>3:371</b>
5.1	Reading and Writing Resources . . . . .	3:371
5.2	Dependencies and Serialization . . . . .	3:371
5.3	Resource and Dependency Table Format Notes . . . . .	3:372
5.3.1	Special Case Instruction Rules . . . . .	3:374
5.3.2	RAW Dependency Table . . . . .	3:374
5.3.3	WAW Dependency Table . . . . .	3:383
5.3.4	WAR Dependency Table . . . . .	3:387
5.3.5	Listing of Rules Referenced in Dependency Tables . . . . .	3:387
5.4	Support Tables . . . . .	3:389
<b>Index.</b>	<b>3:397</b>	

## Figures

2-1	Add Pointer . . . . .	3:15
2-2	Stack Frame . . . . .	3:16
2-3	Operation of br.ctop and br.cexit . . . . .	3:23
2-4	Operation of br.wtop and br.wexit . . . . .	3:24
2-5	Deposit Example (merge_form) . . . . .	3:51
2-6	Deposit Example (zero_form) . . . . .	3:51
2-7	Extract Example . . . . .	3:54
2-8	Floating-point Merge Negative Sign Operation . . . . .	3:80
2-9	Floating-point Merge Sign Operation . . . . .	3:80
2-10	Floating-point Merge Sign and Exponent Operation . . . . .	3:80
2-11	Floating-point Mix Left . . . . .	3:83
2-12	Floating-point Mix Right . . . . .	3:83
2-13	Floating-point Mix Left-Right . . . . .	3:83
2-14	Floating-point Pack . . . . .	3:96
2-15	Floating-point Parallel Merge Negative Sign Operation . . . . .	3:111
2-16	Floating-point Parallel Merge Sign Operation . . . . .	3:111
2-17	Floating-point Parallel Merge Sign and Exponent Operation . . . . .	3:112
2-18	Floating-point Swap . . . . .	3:137
2-19	Floating-point Swap Negate Left . . . . .	3:137
2-20	Floating-point Swap Negate Right . . . . .	3:138
2-21	Floating-point Sign Extend Left . . . . .	3:139
2-22	Floating-point Sign Extend Right . . . . .	3:139
2-23	Function of getf.exp . . . . .	3:143
2-24	Function of getf.sig . . . . .	3:143

2-25	Mix Examples . . . . .	3:170
2-26	Mux1 Operation (8-bit elements) . . . . .	3:190
2-27	Mux2 Examples (16-bit elements) . . . . .	3:191
2-28	Pack Operation . . . . .	3:195
2-29	Parallel Add Examples . . . . .	3:197
2-30	Parallel Average Example . . . . .	3:201
2-31	Parallel Average with Round Away from Zero Example . . . . .	3:202
2-32	Parallel Average Subtract Example . . . . .	3:204
2-33	Parallel Compare Examples . . . . .	3:206
2-34	Parallel Maximum Examples . . . . .	3:209
2-35	Parallel Minimum Examples . . . . .	3:211
2-36	Parallel Multiply Operation . . . . .	3:213
2-37	Parallel Multiply and Shift Right Operation . . . . .	3:214
2-38	Parallel Sum of Absolute Difference Example . . . . .	3:220
2-39	Parallel Shift Left Examples . . . . .	3:222
2-40	Parallel Subtract Examples . . . . .	3:227
2-41	Function of setf.exp . . . . .	3:242
2-42	Function of setf.sig . . . . .	3:242
2-43	Shift Left and Add Pointer . . . . .	3:246
2-44	Shift Right Pair . . . . .	3:248
2-45	Unpack Operation . . . . .	3:271
4-1	Bundle Format . . . . .	3:293

## Tables

2-1	Instruction Page Description . . . . .	3:11
2-2	Instruction Page Font Conventions . . . . .	3:11
2-3	Register File Notation . . . . .	3:12
2-4	C Syntax Differences . . . . .	3:12
2-5	Pervasive Conditions Not Included in Instruction Description Code . . . . .	3:13
2-6	Branch Types . . . . .	3:20
2-7	Branch Whether Hint . . . . .	3:25
2-8	Sequential Prefetch Hint . . . . .	3:25
2-9	Branch Cache Deallocation Hint . . . . .	3:25
2-10	Long Branch Types . . . . .	3:30
2-11	IP-relative Branch Predict Whether Hint . . . . .	3:32
2-12	Indirect Branch Predict Whether Hint . . . . .	3:32
2-13	Importance Hint . . . . .	3:32
2-14	ALAT Clear Completer . . . . .	3:35
2-15	Comparison Types . . . . .	3:39
2-16	64-bit Comparison Relations for Normal and unc Comparisons . . . . .	3:40
2-17	64-bit Comparison Relations for Parallel Comparisons . . . . .	3:40
2-18	Immediate Range for 32-bit Comparisons . . . . .	3:43
2-19	Memory Compare and Exchange Size . . . . .	3:46
2-20	Compare and Exchange Semaphore Types . . . . .	3:46
2-21	Result Ranges for czx . . . . .	3:49
2-22	Specified pc Mnemonic Values . . . . .	3:56
2-23	sf Mnemonic Values . . . . .	3:56
2-24	Floating-point Class Relations . . . . .	3:64

2-25	Floating-point Classes . . . . .	3:64
2-26	Floating-point Comparison Types . . . . .	3:67
2-27	Floating-point Comparison Relations . . . . .	3:67
2-28	Fetch and Add Semaphore Types . . . . .	3:74
2-29	Floating-point Parallel Comparison Results . . . . .	3:101
2-30	Floating-point Parallel Comparison Relations . . . . .	3:101
2-31	Hint Immediates . . . . .	3:145
2-32	<i>sz</i> Completers . . . . .	3:151
2-33	Load Types . . . . .	3:151
2-34	Load Hints . . . . .	3:152
2-35	<i>fsz</i> Completers . . . . .	3:157
2-36	FP Load Types . . . . .	3:157
2-37	<i>lftype</i> Mnemonic Values . . . . .	3:164
2-38	<i>lfhint</i> Mnemonic Values . . . . .	3:165
2-39	Move to BR Whether Hints . . . . .	3:174
2-40	Indirect Register File Mnemonics . . . . .	3:180
2-41	Mux Permutations for 8-bit Elements . . . . .	3:190
2-42	Pack Saturation Limits . . . . .	3:195
2-43	Parallel Add Saturation Completers . . . . .	3:197
2-44	Parallel Add Saturation Limits . . . . .	3:197
2-45	Pcmp Relations . . . . .	3:206
2-46	Parallel Multiply and Shift Right Shift Options . . . . .	3:214
2-47	Faults for regular_form and fault_form Probe Instructions . . . . .	3:218
2-48	Parallel Subtract Saturation Completers . . . . .	3:227
2-49	Parallel Subtract Saturation Limits . . . . .	3:227
2-50	Store Types . . . . .	3:251
2-51	Store Hints . . . . .	3:252
2-52	<i>xsz</i> Mnemonic Values . . . . .	3:258
2-53	Test Bit Relations for Normal and unc tbits . . . . .	3:261
2-54	Test Bit Relations for Parallel tbits . . . . .	3:261
2-55	Test Feature Relations for Normal and unc tf . . . . .	3:263
2-56	Test Feature Relations for Parallel tf . . . . .	3:263
2-57	Test Feature Features Assignment . . . . .	3:263
2-58	Test NaT Relations for Normal and unc tnats . . . . .	3:266
2-59	Test NaT Relations for Parallel tnats . . . . .	3:266
2-60	Memory Exchange Size . . . . .	3:274
3-1	Pseudo-code Functions . . . . .	3:281
4-1	Relationship between Instruction Type and Execution Unit Type . . . . .	3:293
4-2	Template Field Encoding and Instruction Slot Mapping . . . . .	3:294
4-3	Major Opcode Assignments . . . . .	3:295
4-4	Instruction Format Summary . . . . .	3:296
4-5	Instruction Field Color Key . . . . .	3:298
4-6	Instruction Field Names . . . . .	3:298
4-7	Special Instruction Notations . . . . .	3:299
4-8	Integer ALU 2-bit+1-bit Opcode Extensions . . . . .	3:300
4-9	Integer ALU 4-bit+2-bit Opcode Extensions . . . . .	3:301
4-10	Integer Compare Opcode Extensions . . . . .	3:303
4-11	Integer Compare Immediate Opcode Extensions . . . . .	3:303
4-12	Multimedia ALU 2-bit+1-bit Opcode Extensions . . . . .	3:306
4-13	Multimedia ALU Size 1 4-bit+2-bit Opcode Extensions . . . . .	3:307
4-14	Multimedia ALU Size 2 4-bit+2-bit Opcode Extensions . . . . .	3:307

4-15	Multimedia ALU Size 4 4-bit+2-bit Opcode Extensions . . . . .	3:308
4-16	Multimedia and Variable Shift 1-bit Opcode Extensions. . . . .	3:310
4-17	Multimedia Opcode 7 Size 1 2-bit Opcode Extensions . . . . .	3:310
4-18	Multimedia Opcode 7 Size 2 2-bit Opcode Extensions . . . . .	3:311
4-19	Multimedia Opcode 7 Size 4 2-bit Opcode Extensions . . . . .	3:311
4-20	Variable Shift Opcode 7 2-bit Opcode Extensions. . . . .	3:312
4-21	Integer Shift/Test Bit/Test NaT 2-bit Opcode Extensions . . . . .	3:315
4-22	Deposit Opcode Extensions . . . . .	3:315
4-23	Test Bit Opcode Extensions . . . . .	3:317
4-24	Misc I-Unit 3-bit Opcode Extensions. . . . .	3:318
4-25	Misc I-Unit 6-bit Opcode Extensions. . . . .	3:319
4-26	Misc I-Unit 1-bit Opcode Extensions. . . . .	3:319
4-27	Move to BR Whether Hint Completer . . . . .	3:320
4-28	Integer Load/Store/Semaphore/Get FR 1-bit Opcode Extensions. . . . .	3:323
4-29	Floating-point Load/Store/Load Pair/Set FR 1-bit Opcode Extensions . . . . .	3:323
4-30	Integer Load/Store Opcode Extensions . . . . .	3:324
4-31	Integer Load +Reg Opcode Extensions . . . . .	3:324
4-32	Integer Load/Store +Imm Opcode Extensions. . . . .	3:325
4-33	Semaphore/Get FR/16-Byte Opcode Extensions . . . . .	3:325
4-34	Floating-point Load/Store/Lfetch Opcode Extensions . . . . .	3:326
4-35	Floating-point Load/Lfetch +Reg Opcode Extensions . . . . .	3:326
4-36	Floating-point Load/Store/Lfetch +Imm Opcode Extensions . . . . .	3:327
4-37	Floating-point Load Pair/Set FR Opcode Extensions . . . . .	3:327
4-38	Floating-point Load Pair +Imm Opcode Extensions . . . . .	3:328
4-39	Load Hint Completer. . . . .	3:328
4-40	Store Hint Completer . . . . .	3:328
4-41	Line Prefetch Hint Completer . . . . .	3:337
4-42	Opcde 0 System/Memory Management 3-bit Opcode Extensions. . . . .	3:345
4-43	Opcde 0 System/Memory Management 4-bit+2-bit Opcode Extensions. . . . .	3:345
4-44	Opcde 1 System/Memory Management 3-bit Opcde Extensions. . . . .	3:346
4-45	Opcde 1 System/Memory Management 6-bit Opcde Extensions. . . . .	3:346
4-46	Misc M-Unit 1-bit Opcode Extensions . . . . .	3:349
4-47	IP-Relative Branch Types . . . . .	3:350
4-48	Indirect/Miscellaneous Branch Opcode Extensions . . . . .	3:350
4-49	Indirect Branch Types . . . . .	3:351
4-50	Indirect Return Branch Types . . . . .	3:351
4-51	Sequential Prefetch Hint Completer . . . . .	3:351
4-52	Branch Whether Hint Completer. . . . .	3:352
4-53	Indirect Call Whether Hint Completer . . . . .	3:352
4-54	Branch Cache Deallocation Hint Completer. . . . .	3:352
4-55	Indirect Predict/Nop/Hint Opcode Extensions . . . . .	3:354
4-56	Branch Importance Hint Completer . . . . .	3:354
4-57	IP-Relative Predict Whether Hint Completer . . . . .	3:354
4-58	Indirect Predict Whether Hint Completer . . . . .	3:355
4-59	Miscellaneous Floating-point 1-bit Opcode Extensions . . . . .	3:356
4-60	Opcde 0 Miscellaneous Floating-point 6-bit Opcde Extensions . . . . .	3:357
4-61	Opcde 1 Miscellaneous Floating-point 6-bit Opcde Extensions . . . . .	3:357
4-62	Reciprocal Approximation 1-bit Opcde Extensions. . . . .	3:358
4-63	Floating-point Status Field Completer . . . . .	3:358
4-64	Floating-point Arithmetic 1-bit Opcde Extensions . . . . .	3:358
4-65	Fixed-point Multiply Add and Select Opcde Extensions . . . . .	3:358

4-66	Floating-point Compare Opcode Extensions . . . . .	3:360
4-67	Floating-point Class 1-bit Opcode Extensions. . . . .	3:360
4-68	Misc F-Unit 1-bit Opcode Extensions . . . . .	3:365
4-69	Misc X-Unit 3-bit Opcode Extensions . . . . .	3:366
4-70	Misc X-Unit 6-bit Opcode Extensions . . . . .	3:366
4-71	Move Long 1-bit Opcode Extensions . . . . .	3:367
4-72	Long Branch Types . . . . .	3:367
4-73	Misc X-Unit 1-bit Opcode Extensions . . . . .	3:368
4-74	Immediate Formation . . . . .	3:368
5-1	Semantics of Dependency Codes . . . . .	3:373
5-2	RAW Dependencies Organized by Resource . . . . .	3:375
5-3	WAW Dependencies Organized by Resource. . . . .	3:383
5-4	WAR Dependencies Organized by Resource . . . . .	3:387
5-5	Instruction Classes . . . . .	3:389

§

The Intel® Itanium® 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 Itanium 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 Itanium architecture is IA-32 instruction set compatibility.

The *Intel® Itanium® 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 the features of the Itanium architecture to help them optimize code.

## 1.1 Overview of Volume 1: Application Architecture

This volume defines the Itanium 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: Application Architecture Guide

[Chapter 1, “About this Manual”](#) provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

[Chapter 2, “Introduction to the Intel® Itanium® Architecture”](#) provides an overview of the architecture.

[Chapter 3, “Execution Environment”](#) describes the Itanium register set used by applications and the memory organization models.

[Chapter 4, “Application Programming Model”](#) gives an overview of the behavior of Itanium application instructions (grouped into related functions).

[Chapter 5, “Floating-point Programming Model”](#) describes the Itanium floating-point architecture (including integer multiply).

[Chapter 6, “IA-32 Application Execution Model in an Intel® Itanium® System Environment”](#) describes the operation of IA-32 instructions within the Itanium System Environment from the perspective of an application programmer.

### 1.1.2 Part 2: Optimization Guide for the Intel® Itanium® Architecture

[Chapter 1, “About the Optimization Guide”](#) gives an overview of the optimization guide.

[Chapter 2, "Introduction to Programming for the Intel® Itanium® Architecture"](#) provides an overview of the application programming environment for the Itanium architecture.

[Chapter 3, "Memory Reference"](#) discusses features and optimizations related to control and data speculation.

[Chapter 4, "Predication, Control Flow, and Instruction Stream"](#) describes optimization features related to predication, control flow, and branch hints.

[Chapter 5, "Software Pipelining and Loop Support"](#) provides a detailed discussion on optimizing loops through use of software pipelining.

[Chapter 6, "Floating-point Applications"](#) discusses current performance limitations in floating-point applications and features that address these limitations.

## 1.2

## Overview of Volume 2: System Architecture

This volume defines the Itanium 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: System Architecture Guide

[Chapter 1, "About this Manual"](#) provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer's Manual*.

[Chapter 2, "Intel® Itanium® System Environment"](#) introduces the environment designed to support execution of Itanium architecture-based operating systems running IA-32 or Itanium architecture-based applications.

[Chapter 3, "System State and Programming Model"](#) describes the Itanium architectural state which is visible only to an operating system.

[Chapter 4, "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, "Interruptions"](#) describes all interruptions that can be generated by a processor based on the Itanium architecture.

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

[Chapter 7, "Debugging and Performance Monitoring"](#) is an overview of the performance monitoring and debugging resources that are available in the Itanium architecture.

[Chapter 8, "Interrupt Vector Descriptions"](#) lists all 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 Itanium System Environment.

[Chapter 10, "Itanium® Architecture-based Operating System Interaction Model with IA-32 Applications"](#) defines the operation of IA-32 instructions within the Itanium System Environment from the perspective of an Itanium architecture-based operating system.

[Chapter 11, "Processor Abstraction Layer"](#) describes the firmware layer which abstracts processor implementation-dependent features.

## 1.2.2 Part 2: System Programmer's Guide

[Chapter 1, "About the System Programmer's Guide"](#) gives an introduction to the second section of the system architecture guide.

[Chapter 2, "MP Coherence and Synchronization"](#) describes multiprocessing synchronization primitives and the Itanium memory ordering model.

[Chapter 3, "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 4, "Context Management"](#) describes how operating systems need to preserve Itanium register contents and state. This chapter also describes 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 5, "Memory Management"](#) introduces various memory management strategies.

[Chapter 6, "Runtime Support for Control and Data Speculation"](#) describes the operating system support that is required for control and data speculation.

[Chapter 7, "Instruction Emulation and Other Fault Handlers"](#) describes a variety of instruction emulation handlers that Itanium architecture-based operating systems are expected to support.

[Chapter 8, "Floating-point System Software"](#) discusses how processors based on the Itanium architecture handle floating-point numeric exceptions and how the software stack provides complete IEEE-754 compliance.

[Chapter 9, "IA-32 Application Support"](#) describes the support an Itanium architecture-based operating system needs to provide to host IA-32 applications.

[Chapter 10, "External Interrupt Architecture"](#) describes the external interrupt architecture with a focus on how external asynchronous interrupt handling can be controlled by software.

[Chapter 11, "I/O Architecture"](#) describes the I/O architecture with a focus on platform issues and support for the existing IA-32 I/O port space.

[Chapter 12, "Performance Monitoring Support"](#) describes the performance monitor architecture with a focus on what kind of support is needed from Itanium architecture-based operating systems.

[Chapter 13, "Firmware Overview"](#) introduces the firmware model, and how various firmware layers (PAL, SAL, UEFI, ACPI) work together to enable processor and system initialization, and operating system boot.

### **1.2.3 Appendices**

[Appendix A, "Code Examples"](#) provides OS boot flow sample code.

## **1.3 Overview of Volume 3: Intel® Itanium® Instruction Set Reference**

This volume is a comprehensive reference to the Itanium instruction set, including instruction format/encoding.

[Chapter 1, "About this Manual"](#) provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer's Manual*.

[Chapter 2, "Instruction Reference"](#) provides a detailed description of all Itanium instructions, organized in alphabetical order by assembly language mnemonic.

[Chapter 3, "Pseudo-Code Functions"](#) provides a table of pseudo-code functions which are used to define the behavior of the Itanium instructions.

[Chapter 4, "Instruction Formats"](#) describes the encoding and instruction format instructions.

[Chapter 5, "Resource and Dependency Semantics"](#) summarizes the dependency rules that are applicable when generating code for processors based on the Itanium architecture.

## **1.4 Overview of Volume 4: IA-32 Instruction Set Reference**

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

[Chapter 1, "About this Manual"](#) provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer's Manual*.

[Chapter 2, "Base IA-32 Instruction Reference"](#) provides a detailed description of all base IA-32 instructions, organized in alphabetical order by assembly language mnemonic.

[Chapter 3, "IA-32 Intel® MMX™ Technology Instruction Reference"](#) provides a detailed description of all IA-32 Intel® MMX™ technology instructions designed to increase performance of multimedia intensive applications. Organized in alphabetical order by assembly language mnemonic.

[Chapter 4, "IA-32 SSE Instruction Reference"](#) provides a detailed description of all IA-32 SSE instructions designed to increase performance of multimedia intensive applications, and is organized in alphabetical order by assembly language mnemonic.

## 1.5 Terminology

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

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

**Itanium 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® 64 and IA-32 Architectures Software Developer's Manual](#).

**Itanium System Environment** – The operating system environment that supports the execution of both IA-32 and Itanium architecture-based code.

**Itanium® Architecture-based Firmware** – The Processor Abstraction Layer (PAL) and System Abstraction Layer (SAL).

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

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

## 1.6 Related Documents

The following documents can be downloaded at the Intel's Developer Site at <http://developer.intel.com>:

- **Dual-Core Update to the Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization** – Document number 308065 provides model-specific information about the dual-core Itanium processors.
- **Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization** – This document (Document number 251110) describes model-specific architectural features incorporated into the Intel® Itanium® 2 processor, the second processor based on the Itanium architecture.
- **Intel® Itanium® Processor Reference Manual for Software Development** – This document (Document number 245320) describes model-specific architectural features incorporated into the Intel® Itanium® processor, the first processor based on the Itanium architecture.

- **Intel® 64 and IA-32 Architectures Software Developer’s Manual** – This set of manuals describes the Intel 32-bit architecture. They are available from the Intel Literature Department by calling 1-800-548-4725 and requesting Document Numbers 243190, 243191 and 243192.
- **Intel® Itanium® Software Conventions and Runtime Architecture Guide** – This document (Document number 245358) defines general information necessary to compile, link, and execute a program on an Itanium architecture-based operating system.
- **Intel® Itanium® Processor Family System Abstraction Layer Specification** – This document (Document number 245359) specifies requirements to develop platform firmware for Itanium architecture-based systems.

The following document can be downloaded at the Unified EFI Forum website at <http://www.uefi.org>:

- **Unified Extensible Firmware Interface Specification** – This document defines a new model for the interface between operating systems and platform firmware.

## 1.7 Revision History

Date of Revision	Revision Number	Description
March 2010	2.3	<p>Added information about illegal virtualization optimization combinations and IIPA requirements.</p> <p>Added Resource Utilization Counter and PAL_VP_INFO.</p> <p>PAL_VP_INIT and VPD.vpr changes.</p> <p>New PAL_VPS_RESUME_HANDLER parameter to indicate RSE Current Frame Load Enable setting at the target instruction.</p> <p>PAL_VP_INIT_ENV implementation-specific configuration option.</p> <p>Minimum Virtual address increased to 54 bits.</p> <p>New PAL_MC_ERROR_INFO health indicator.</p> <p>New PAL_MC_ERROR_INJECT implementation-specific bit fields.</p> <p>MOV-to_SR.L reserved field checking.</p> <p>Added virtual machine disable.</p> <p>Added variable frequency mode additions to ACPI P-state description.</p> <p>Removed <i>pal_proc_vector</i> argument from PAL_VP_SAVE and PAL_VP_RESTORE.</p> <p>Added PAL_PROC_SET_FEATURES data speculation disable.</p> <p>Added Interruption Instruction Bundle registers.</p> <p>Min-state save area size change.</p> <p>PAL_MC_DYNAMIC_STATE changes.</p> <p>PAL_PROC_SET_FEATURES data poisoning promotion changes.</p> <p>ACPI P-state clarifications.</p> <p>Synchronization requirements for virtualization opcode optimization.</p> <p>New priority hint and multi-threading hint recommendations.</p>

Date of Revision	Revision Number	Description
August 2005	2.2	<p>Allow register fields in CR.LID register to be read-only and CR.LID checking on interruption messages by processors optional. See Vol 2, Part I, Ch 5 "Interruptions" and Section 11.2.2 PALE_RESET Exit State for details.</p> <p>Relaxed reserved and ignored fields checkings in IA-32 application registers in Vol 1 Ch 6 and Vol 2, Part I, Ch 10.</p> <p>Introduced visibility constraints between stores and local purges to ensure TLB consistency for UP VHPT update and local purge scenarios. See Vol 2, Part I, Ch 4 and description of <code>ptc.1</code> instruction in Vol 3 for details.</p> <p>Architecture extensions for processor Power/Performance states (P-states). See Vol 2 PAL Chapter for details.</p> <p>Introduced Unimplemented Instruction Address fault.</p> <p>Relaxed ordering constraints for VHPT walks. See Vol 2, Part I, Ch 4 and 5 for details.</p> <p>Architecture extensions for processor virtualization.</p> <p>All instructions which must be last in an instruction group results in undefined behavior when this rule is violated.</p> <p>Added architectural sequence that guarantees increasing ITC and PMD values on successive reads.</p> <p>Addition of <code>PAL_BRAND_INFO</code>, <code>PAL_GET_HW_POLICY</code>, <code>PAL_MC_ERROR_INJECT</code>, <code>PAL_MEMORY_BUFFER</code>, <code>PAL_SET_HW_POLICY</code> and <code>PAL_SHUTDOWN</code> procedures.</p> <p>Allows IPI-redirection feature to be optional.</p> <p>Undefined behavior for 1-byte accesses to the non-architected regions in the IPI block.</p> <p>Modified insertion behavior for TR overlaps. See Vol 2, Part I, Ch 4 for details.</p> <p>"Bus parking" feature is now optional for <code>PAL_BUS_GET_FEATURES</code>.</p> <p>Introduced low-power synchronization primitive using <code>hint</code> instruction.</p> <p>FR32-127 is now preserved in PAL calling convention.</p> <p>New return value from <code>PAL_VM_SUMMARY</code> procedure to indicate the number of multiple concurrent outstanding TLB purges.</p> <p>Performance Monitor Data (PMD) registers are no longer sign-extended.</p> <p>New memory attribute transition sequence for memory on-line delete. See Vol 2, Part I, Ch 4 for details.</p> <p>Added 'shared error' (se) bit to the Processor State Parameter (PSP) in <code>PAL_MC_ERROR_INFO</code> procedure.</p> <p>Clarified PMU interrupts as edge-triggered.</p> <p>Modified 'proc_number' parameter in <code>PAL_LOGICAL_TO_PHYSICAL</code> procedure.</p> <p>Modified <code>pal_copy_info</code> alignment requirements.</p> <p>New bit in <code>PAL_PROC_GET_FEATURES</code> for variable P-state performance.</p> <p>Clarified descriptions for <code>check_target_register</code> and <code>check_target_register_sof</code>.</p> <p>Various fixes in dependency tables in Vol 3 Ch 5.</p> <p>Clarified effect of sending IPIs to non-existent processor in Vol 2, Part I, Ch 5.</p> <p>Clarified instruction serialization requirements for interruptions in Vol 2, Part II, Ch 3.</p> <p>Updated performance monitor context switch routine in Vol 2, Part I, Ch 7.</p>

Date of Revision	Revision Number	Description
August 2002	2.1	<p>Added Predicate Behavior of <code>alloc</code> Instruction Clarification (Section 4.1.2, Part I, Volume 1; Section 2.2, Part I, Volume 3).</p> <p>Added New <code>fc.i</code> Instruction (Section 4.4.6.1, and 4.4.6.2, Part I, Volume 1; Section 4.3.3, 4.4.1, 4.4.5, 4.4.6, 4.4.7, 5.5.2, and 7.1.2, Part I, Volume 2; Section 2.5, 2.5.1, 2.5.2, 2.5.3, and 4.5.2.1, Part II, Volume 2; Section 2.2, 3, 4.1, 4.4.6.5, and 4.4.10.10, Part I, Volume 3).</p> <p>Added Interval Time Counter (ITC) Fault Clarification (Section 3.3.2, Part I, Volume 2).</p> <p>Added Interruption Control Registers Clarification (Section 3.3.5, Part I, Volume 2).</p> <p>Added Spontaneous NaT Generation on Speculative Load (<code>ld.s</code>) (Section 5.5.5 and 11.9, Part I, Volume 2; Section 2.2 and 3, Part I, Volume 3).</p> <p>Added Performance Counter Standardization (Sections 7.2.3 and 11.6, Part I, Volume 2).</p> <p>Added Freeze Bit Functionality in Context Switching and Interrupt Generation Clarification (Sections 7.2.1, 7.2.2, 7.2.4.1, and 7.2.4.2, Part I, Volume 2)</p> <p>Added IA_32_Exception (Debug) IIPA Description Change (Section 9.2, Part I, Volume 2).</p> <p>Added capability for Allowing Multiple PAL_A_SPEC and PAL_B Entries in the Firmware Interface Table (Section 11.1.6, Part I, Volume 2).</p> <p>Added BR1 to Min-state Save Area (Sections 11.3.2.3 and 11.3.3, Part I, Volume 2).</p> <p>Added Fault Handling Semantics for <code>lfetch.fault</code> Instruction (Section 2.2, Part I, Volume 3).</p>
December 2001	2.0	<p>Volume 1:</p> <p>Faults in <code>ld.c</code> that hits ALAT clarification (Section 4.4.5.3.1).</p> <p>IA-32 related changes (Section 6.2.5.4, Section 6.2.3, Section 6.2.4, Section 6.2.5.3).</p> <p>Load instructions change (Section 4.4.1).</p>

Date of Revision	Revision Number	Description
		<p>Volume 2:</p> <ul style="list-style-type: none"> <li>Class pr-writers-int clarification (Table A-5).</li> <li>PAL_MC_DRAIN clarification (Section 4.4.6.1).</li> <li>VHPT walk and forward progress change (Section 4.1.1.2).</li> <li>IA-32 IBR/DBR match clarification (Section 7.1.1).</li> <li>ISR figure changes (pp. 8-5, 8-26, 8-33 and 8-36).</li> <li>PAL_CACHE_FLUSH return argument change – added new status return argument (Section 11.8.3).</li> <li>PAL self-test Control and PAL_A procedure requirement change – added new arguments, figures, requirements (Section 11.2).</li> <li>PAL_CACHE_FLUSH clarifications (Chapter 11).</li> <li>Non-speculative reference clarification (Section 4.4.6).</li> <li>RID and Preferred Page Size usage clarification (Section 4.1).</li> <li>VHPT read atomicity clarification (Section 4.1).</li> <li>IIP and WC flush clarification (Section 4.4.5).</li> <li>Revised RSE and PMC typographical errors (Section 6.4).</li> <li>Revised DV table (Section A.4).</li> <li>Memory attribute transitions – added new requirements (Section 4.4).</li> <li>MCA for WC/UC aliasing change (Section 4.4.1).</li> <li>Bus lock deprecation – changed behavior of DCR 'lc' bit (Section 3.3.4.1, Section 10.6.8, Section 11.8.3).</li> <li>PAL_PROC_GET/SET_FEATURES changes – extend calls to allow implementation-specific feature control (Section 11.8.3).</li> <li>Split PAL_A architecture changes (Section 11.1.6).</li> <li>Simple barrier synchronization clarification (Section 13.4.2).</li> <li>Limited speculation clarification – added hardware-generated speculative references (Section 4.4.6).</li> <li>PAL memory accesses and restrictions clarification (Section 11.9).</li> <li>PSP validity on INITs from PAL_MC_ERROR_INFO clarification (Section 11.8.3).</li> <li>Speculation attributes clarification (Section 4.4.6).</li> <li>PAL_A FIT entry, PAL_VM_TR_READ, PSP, PAL_VERSION clarifications (Sections 11.8.3 and 11.3.2.1).</li> <li>TLB searching clarifications (Section 4.1).</li> <li>IA-32 related changes (Section 10.3, Section 10.3.2, Section 10.3.2, Section 10.3.3.1, Section 10.10.1).</li> <li>IPSR.ri and ISR.ei changes (Table 3-2, Section 3.3.5.1, Section 3.3.5.2, Section 5.5, Section 8.3, and Section 2.2).</li> </ul>
		<p>Volume 3:</p> <ul style="list-style-type: none"> <li>IA-32 CPUID clarification (p. 5-71).</li> <li>Revised figures for extract, deposit, and alloc instructions (Section 2.2).</li> <li>RCPPS, RCPSS, RSQRTPS, and RSQRTSS clarification (Section 7.12).</li> <li>IA-32 related changes (Section 5.3).</li> <li>tak, tpa change (Section 2.2).</li> </ul>
July 2000	1.1	<p>Volume 1:</p> <ul style="list-style-type: none"> <li>Processor Serial Number feature removed (Chapter 3).</li> <li>Clarification on exceptions to instruction dependency (Section 3.4.3).</li> </ul>

Date of Revision	Revision Number	Description
		<p>Volume 2:</p> <p>Clarifications regarding “reserved” fields in ITIR (Chapter 3). Instruction and Data translation must be enabled for executing IA-32 instructions (Chapters 3,4 and 10).</p> <p>FCR/FDR mappings, and clarification to the value of PSR.ri after an RFI (Chapters 3 and 4).</p> <p>Clarification regarding ordering data dependency.</p> <p>Out-of-order IPI delivery is now allowed (Chapters 4 and 5).</p> <p>Content of EFLAG field changed in IIM (p. 9-24).</p> <p>PAL_CHECK and PAL_INIT calls – exit state changes (Chapter 11).</p> <p>PAL_CHECK processor state parameter changes (Chapter 11).</p> <p>PAL_BUS_GET/SET_FEATURES calls – added two new bits (Chapter 11).</p> <p>PAL_MC_ERROR_INFO call – Changes made to enhance and simplify the call to provide more information regarding machine check (Chapter 11).</p> <p>PAL_ENTER_IA_32_Env call changes – entry parameter represents the entry order; SAL needs to initialize all the IA-32 registers properly before making this call (Chapter 11).</p> <p>PAL_CACHE_FLUSH – added a new cache_type argument (Chapter 11).</p> <p>PAL_SHUTDOWN – removed from list of PAL calls (Chapter 11).</p> <p>Clarified memory ordering changes (Chapter 13).</p> <p>Clarification in dependence violation table (Appendix A).</p>
		<p>Volume 3:</p> <p>fmix instruction page figures corrected (Chapter 2).</p> <p>Clarification of “reserved” fields in ITIR (Chapters 2 and 3).</p> <p>Modified conditions for alloc/loadrs/flushrs instruction placement in bundle/instruction group (Chapters 2 and 4).</p> <p>IA-32 JMPE instruction page typo fix (p. 5-238).</p> <p>Processor Serial Number feature removed (Chapter 5).</p>
January 2000	1.0	Initial release of document.

S

This chapter describes the function of each Itanium 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
<i>italic</i>	(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
<i>code_italic</i>	(Operation section) Assembly language field name corresponding to a <i>italic</i> 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.

The Operation section contains code that specifies only the execution semantics of each instruction and does not include any behavior relating to instruction fetch (e.g., interrupts and faults caused during fetch). The Interruptions section does not list any faults that may be caused by instruction fetch or by mandatory RSE loads. The code to raise certain pervasive faults and actions is not included in the code in the Operation section. These faults and actions are listed in [Table 2-5](#). The Single step trap applies to all instructions and is not listed in the Interruptions section.

**Table 2-5. Pervasive Conditions Not Included in Instruction Description Code**

Condition	Action
Read of a register outside the current frame.	An undefined value is returned (no fault).
Access to a banked general register (GR 16 through GR 31).	The GR bank specified by PSR.bn is accessed.
PSR.ss is set.	A Single Step trap is raised.

## 2.2 Instruction Descriptions

The remainder of this chapter provides a description of each of the Itanium instructions.

## add — Add

<b>Format:</b>	(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 on the size of the immediate operand and the value of  $r_3$ .

```
Operation: if (PR[qp]) {
    check_target_register( $r_1$ );

    if (register_form)                                // register form
        tmp_src = GR[ $r_2$ ];
    else if (imm14_form)                            // 14-bit immediate form
        tmp_src = sign_ext( $imm_{14}$ , 14);
    else                                              // 22-bit immediate form
        tmp_src = sign_ext( $imm_{22}$ , 22);

    tmp_nat = (register_form ? GR[ $r_2$ ].nat : 0);

    if (plus1_form)
        GR[ $r_1$ ] = tmp_src + GR[ $r_3$ ] + 1;
    else
        GR[ $r_1$ ] = tmp_src + GR[ $r_3$ ];

    GR[ $r_1$ ].nat = tmp_nat || GR[ $r_3$ ].nat;
}
```

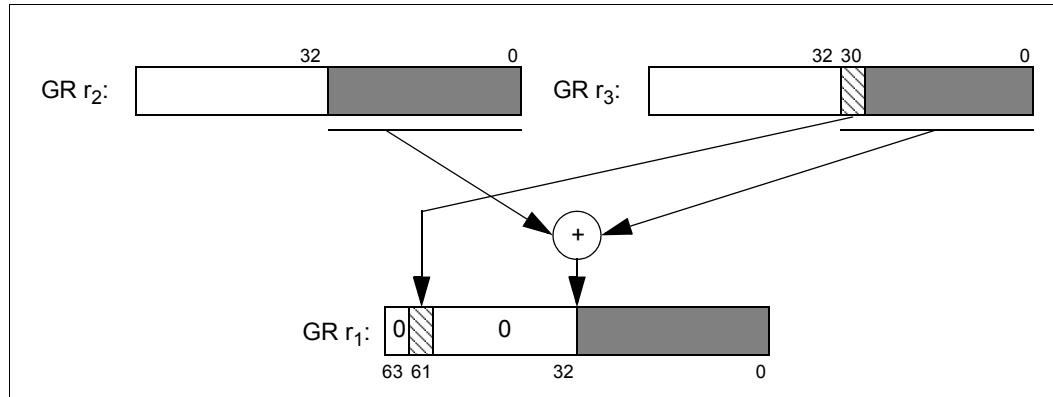
**Interruptions:** Illegal Operation fault

## addp4 – 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( $r_1$ );

    tmp_src = (register_form ? GR[ $r_2$ ] : sign_ext( $imm_{14}$ , 14));
    tmp_nat = (register_form ? GR[ $r_2$ ].nat : 0);

    tmp_res = tmp_src + GR[ $r_3$ ];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[ $r_3$ ]{31:30};
    GR[ $r_1$ ] = tmp_res;
    GR[ $r_1$ ].nat = tmp_nat || GR[ $r_3$ ].nat;
}
```

**Interruptions:** Illegal Operation fault

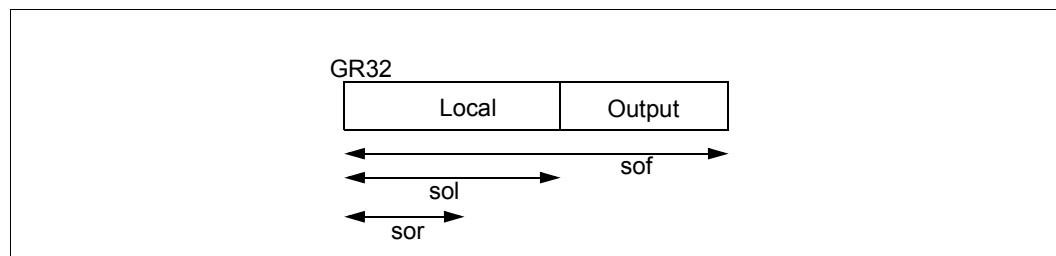
## alloc — Allocate Stack Frame

**Format:** (qp) alloc  $r_1 = \text{ar.pfs}, i, l, o, r$

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. 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, or specifying a qualifying predicate other than PR 0, will cause an Illegal Operation fault.

This instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0; 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 || qp != 0)
    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;

```

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

## **and – 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

## **andcm – 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

## br — Branch

<b>Format:</b>	(qp) br.btype.bwh.ph.dh target <sub>25</sub>	ip_relative_form	B1
	(qp) br.btype.bwh.ph.dh b <sub>1</sub> = target <sub>25</sub>	call_form, ip_relative_form	B3
	br.btype.bwh.ph.dh target <sub>25</sub>	counted_form, ip_relative_form	B2
	br.ph.dh target <sub>25</sub>	pseudo-op	
	(qp) br.btype.bwh.ph.dh b <sub>2</sub>	indirect_form	B4
	(qp) br.btype.bwh.ph.dh b <sub>1</sub> = b <sub>2</sub>	call_form, indirect_form	B5
	br.ph.dh b <sub>2</sub>	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<sub>25</sub>* operand, in assembly, specifies a label to branch to. This is encoded in the branch instruction as a signed immediate displacement (*imm<sub>21</sub>*) between the target bundle and the bundle containing this instruction (*imm<sub>21</sub> = target<sub>25</sub> - IP >> 4*). For indirect branches, the target address is taken from BR *b<sub>2</sub>*.

**Table 2-6. 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.ip 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 set PSR properly before branching to the IA-32 instruction set; otherwise processor operation is undefined. See [Table 3-2, "Processor Status Register Fields" on page 2:24](#) for details.

Software must issue a mf instruction before the branch if memory ordering is required between IA-32 processor consistent and Itanium unordered memory references. The processor does not ensure Itanium-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 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 Itanium 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 (see also [Section 4.5.1, "Modulo-scheduled Loop Support" on page 1:75](#)). 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.

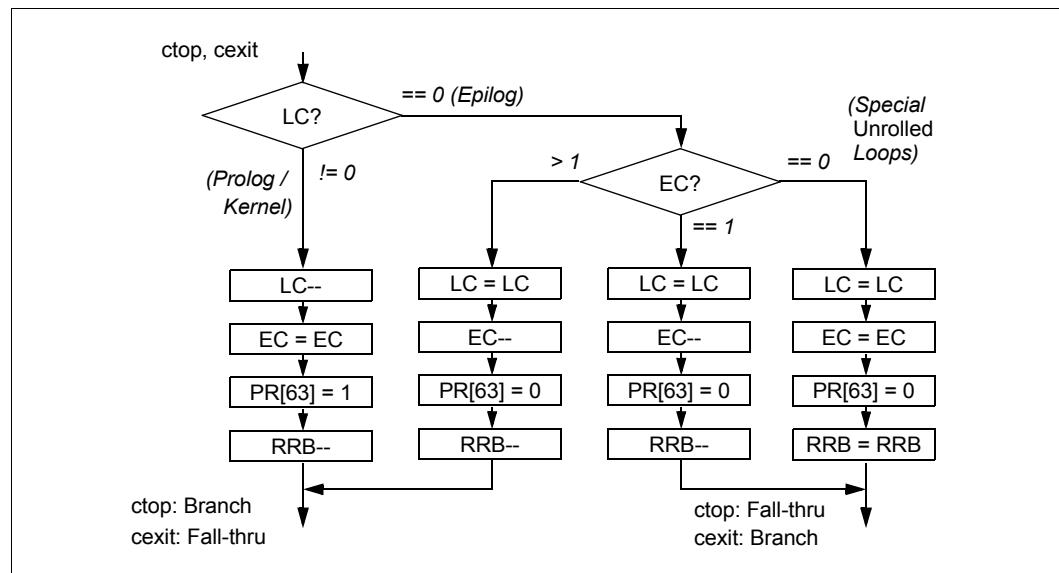
For these modulo-scheduled loop types, the calculation of whether the branch is taken or not depends on the kernel branch condition (LC for counted types, and the qualifying predicate for while types) and on the epilog condition (whether EC is greater than one or not).

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, see [Section 4.5.1, "Modulo-scheduled Loop Support" on page 1:75](#)).

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](#).

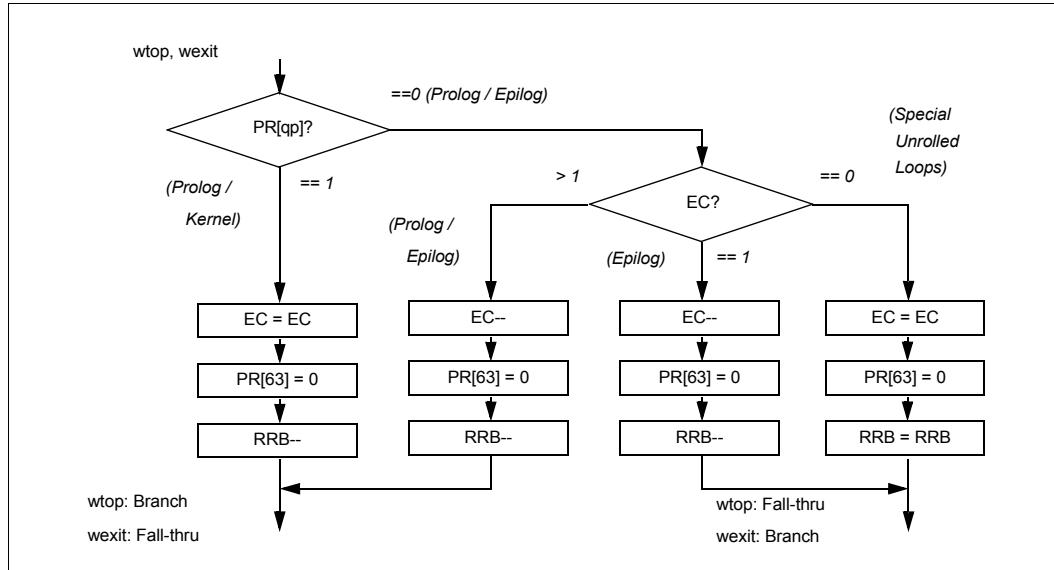
**Figure 2-3. Operation of br.ctop and br.cexit**



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



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.

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 2-7](#). Sequential Prefetch hints are shown in [Table 2-8](#). Branch Cache Deallocation hints are shown in [Table 2-9](#). See [Section 4.5.2, "Branch Prediction Hints" on page 1:78](#).

**Table 2-7. Branch Whether Hint**

<i>bwh</i> Completer	Branch Whether Hint
spnt	Static Not-Taken
sptk	Static Taken
dptn	Dynamic Not-Taken
dptk	Dynamic Taken

**Table 2-8. Sequential Prefetch Hint**

<i>ph</i> Completer	Sequential Prefetch Hint
few or none	Few lines
many	Many lines

**Table 2-9. Branch Cache Deallocation Hint**

<i>dh</i> 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 Itanium 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, "RSE Operation" on page 2:137
        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 < 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 (PSR.ic == 0 || PSR.dt == 0 || PSR.mc == 1 || PSR.it == 0)
    undefined_behavior();
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[EC];
        AR[EC]--;
        PR[63] = 0;
        rotate_regs();
    } else {
        AR[LC] = AR[EC];
        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 (!impl_uia_fault_supported() &&
    ((PSR.it && unimplemented_virtual_address(tmp_IP, PSR.vm))
     || (!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 — Break

<b>Format:</b>	(qp) break <i>imm</i> <sub>21</sub>	pseudo-op
	(qp) break.i <i>imm</i> <sub>21</sub>	i_unit_form I19
	(qp) break.b <i>imm</i> <sub>21</sub>	b_unit_form B9
	(qp) break.m <i>imm</i> <sub>21</sub>	m_unit_form M37
	(qp) break.f <i>imm</i> <sub>21</sub>	f_unit_form F15
	(qp) break.x <i>imm</i> <sub>62</sub>	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*<sub>21</sub> is zero-extended and placed in the Interruption Immediate control register (IIM).

For the b\_unit\_form, *imm*<sub>21</sub> 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*<sub>62</sub> 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*<sub>62</sub>.

A break.i instruction may be encoded in an MLT-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

## bri — Branch Long

<b>Format:</b>	(qp) bri.btype.bwh.ph.dh target <sub>64</sub>	X3
	(qp) bri.btype.bwh.ph.dh b <sub>1</sub> = target <sub>64</sub> bri.ph.dh target <sub>64</sub>	X4 call_form 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<sub>64</sub>* operand, in assembly, specifies a label to branch to. This is encoded in the long branch instruction as an immediate displacement (*imm<sub>60</sub>*) between the target bundle and the bundle containing this instruction (*imm<sub>60</sub> = target<sub>64</sub> - IP >> 4*). The L slot of the bundle contains 39 bits of *imm<sub>60</sub>*.

**Table 2-10. 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<sub>1</sub>*.

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 3:24](#) 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 2-7 on page 3:25](#), Sequential Prefetch hints are shown in [Table 2-8 on page 3:25](#), and Branch Cache Deallocation hints are shown in [Table 2-9 on page 3:25](#). See [Section 4.5.2, "Branch Prediction Hints" on page 1:78](#).

This instruction is not implemented on the 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 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 [Section 3.1.11, “Processor Identification Registers” on page 1:34](#).

```
Operation: tmp_IP = IP + (imm60 << 4); // determine branch target
if (!followed_by_stop())
    undefined_behavior();
if (!instruction_implemented(BRL))
    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 (!impl_uia_fault_supported() &&
        ((PSR.it && unimplemented_virtual_address(tmp_IP, PSR.vm)) ||
         (!PSR.it && unimplemented_physical_address(tmp_IP))))
        unimplemented_instruction_address_trap(0, tmp_IP);
    if (PSR.tb)
        taken_branch_trap();
}
```

<b>Interruptions:</b>	Illegal Operation fault	Taken Branch trap
	Unimplemented Instruction Address trap	

## brp — Branch Predict

<b>Format:</b>	brp.ipwh.ih target <sub>25</sub> , tag <sub>13</sub>	ip_relative_form	B6
	brp.indwh.ih b <sub>2</sub> , tag <sub>13</sub>	indirect_form	B7
	brp.ret.indwh.ih b <sub>2</sub> , tag <sub>13</sub>	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<sub>13</sub>* 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<sub>9</sub>*) between the bundle containing the presaged branch and the bundle containing this instruction (*timm<sub>9</sub> = tag<sub>13</sub> - IP >> 4*).

The *target<sub>25</sub>* 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<sub>21</sub>*) between the target bundle and the bundle containing this instruction (*imm<sub>21</sub> = target<sub>25</sub> - 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<sub>2</sub>*.

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 [Section 4.5.2, “Branch Prediction Hints” on page 1:78](#).

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

**Table 2-11. IP-relative Branch Predict Whether Hint**

<i>ipwh</i> 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 2-12. Indirect Branch Predict Whether Hint**

<i>indwh</i> 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 2-13. Importance Hint**

<i>ih</i> 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

## **bsw – Bank Switch**

<b>Format:</b>	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, operation is undefined. 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 can only be executed at the most privileged level, and when PSR.vm is 0.

This instruction cannot be predicated.

```
Operation:    if (!followed_by_stop())
                     undefined_behavior();

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

                  if (PSR.vm == 1)
                     virtualization_fault();

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

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

## chk — Speculation Check

<b>Format:</b>	(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 ( $imm_{21} = target_{25} - IP >> 4$ ).

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 2-14](#)).

**Table 2-14. ALAT Clear Completer**

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

Note that if the *c/r* 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 (!impl_uia_fault_supported() &&
                ((PSR.it && unimplemented_virtual_address(IP, PSR.vm))
                 || (!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

## clrrrb — 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; otherwise, operation is undefined.

This instruction cannot be predicated.

**Operation:**

```

if (!followed_by_stop())
    undefined_behavior();

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

```

**Interruptions:** None

## clz — Count Leading Zeros

**Format:** (qp) clz  $r_1 = r_3$

**Description:** The number of leading zeros in GR  $r_3$  is placed in GR  $r_1$ .

An Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See [Section 3.1.11, “Processor Identification Registers” on page 1:34](#) for details. This capability may also be determined using the test feature (tf) instruction using the @clz operand.

```
Operation: if (PR[qp])
    if (!instruction_implemented(CLZ))
        illegal_operation_fault();
    check_target_register( $r_1$ );

    tmp_val = 0;

    do {
        if (GR[ $r_3$ ]{63 - tmp_val} != 0) break;
    } while (tmp_val++ < 63);

    GR[ $r_1$ ] = tmp_val;
    GR[ $r_1$ ].nat = GR[ $r_3$ ].nat;
}
```

**Interruptions:** Illegal Operation fault

## cmp — Compare

Format:	(qp) cmp.crel.ctype p <sub>1</sub> , p <sub>2</sub> = r <sub>2</sub> , r <sub>3</sub>	register_form	A6
	(qp) cmp.crel.ctype p <sub>1</sub> , p <sub>2</sub> = imm <sub>8</sub> , r <sub>3</sub>	imm8_form	A8
	(qp) cmp.crel.ctype p <sub>1</sub> , p <sub>2</sub> = r0, r <sub>3</sub>	parallel_inequality_form	A7
	(qp) cmp.crel.ctype p <sub>1</sub> , p <sub>2</sub> = r <sub>3</sub> , 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*<sub>1</sub> and *p*<sub>2</sub>. 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 2-15](#). A blank entry indicates the predicate target is left unchanged.

**Table 2-15. 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 <sub>1</sub> ]	PR[p <sub>2</sub> ]	PR[p <sub>1</sub> ]	PR[p <sub>2</sub> ]	PR[p <sub>1</sub> ]	PR[p <sub>2</sub> ]	PR[p <sub>1</sub> ]	PR[p <sub>2</sub> ]
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		
orcsm	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*<sub>2</sub>; in the imm8\_form the first operand is taken from the sign-extended *imm*<sub>8</sub> 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*<sub>1</sub> and *p*<sub>2</sub> 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 2-16](#) (for normal and unc type compares), and [Table 2-17](#) (for parallel type compares).

**Table 2-16. 64-bit Comparison Relations for Normal and unc Compares**

<i>crel</i>	Compare Relation ( <i>a rel b</i> )	Register Form is a pseudo-op of	Immediate Form is a pseudo-op of	Immediate Range
eq	<i>a == b</i>			-128 .. 127
ne	<i>a != b</i>	eq	<i>p<sub>1</sub> ↔ p<sub>2</sub></i>	-128 .. 127
lt	<i>a &lt; b</i>	signed		-128 .. 127
le	<i>a &lt;= b</i>	lt	<i>a ↔ b      p<sub>1</sub> ↔ p<sub>2</sub></i>	-127 .. 128
gt	<i>a &gt; b</i>	lt	<i>a ↔ b</i>	-127 .. 128
ge	<i>a &gt;= b</i>	lt	<i>p<sub>1</sub> ↔ p<sub>2</sub></i>	-128 .. 127
ltu	<i>a &lt; b</i>	unsigned		0 .. 127, $2^{64}-128 .. 2^{64}-1$
leu	<i>a &lt;= b</i>	ltu	<i>a ↔ b      p<sub>1</sub> ↔ p<sub>2</sub></i>	1 .. 128, $2^{64}-127 .. 2^{64}$
gtu	<i>a &gt; b</i>	ltu	<i>a ↔ b</i>	1 .. 128, $2^{64}-127 .. 2^{64}$
geu	<i>a &gt;= b</i>	ltu	<i>p<sub>1</sub> ↔ p<sub>2</sub></i>	0 .. 127, $2^{64}-128 .. 2^{64}-1$

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*<sub>2</sub>) 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.

**Table 2-17. 64-bit Comparison Relations for Parallel Compares**

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

```

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)

```

**cmp**

```
    illegal_operation_fault();  
    PR[p1] = 0;  
    PR[p2] = 0;  
}
```

```
}
```

**Interruptions:** Illegal Operation fault

## cmp4 — Compare 4 Bytes

<b>Format:</b>	(qp) cmp4.crel ctype p <sub>1</sub> , p <sub>2</sub> = r <sub>2</sub> , r <sub>3</sub>	register_form	A6
	(qp) cmp4.crel ctype p <sub>1</sub> , p <sub>2</sub> = imm <sub>8</sub> , r <sub>3</sub>	imm8_form	A8
	(qp) cmp4.crel ctype p <sub>1</sub> , p <sub>2</sub> = r0, r <sub>3</sub>	parallel_inequality_form	A7
	(qp) cmp4.crel ctype p <sub>1</sub> , p <sub>2</sub> = r <sub>3</sub> , 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*<sub>1</sub> and *p*<sub>2</sub>. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and [Table 2-15 on page 3:39](#).

In the register\_form the first operand is GR *r*<sub>2</sub>; in the imm8\_form the first operand is taken from the sign-extended *imm*<sub>8</sub> 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 2-17 on page 3:40](#).

If the two predicate register destinations are the same (*p*<sub>1</sub> and *p*<sub>2</sub> 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 2-16](#) and [Table 2-17 on page 3:40](#). The range for immediates is given below.

**Table 2-18. Immediate Range for 32-bit Comparisons**

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 <sup>32</sup> -128 .. 2 <sup>32</sup> -1
leu	a <= b	1 .. 128, 2 <sup>32</sup> -127 .. 2 <sup>32</sup>	
gtu	a > b	1 .. 128, 2 <sup>32</sup> -127 .. 2 <sup>32</sup>	
geu	a >= b	0 .. 127, 2 <sup>32</sup> -128 .. 2 <sup>32</sup> -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':
default:
    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

## cmpxchg — Compare and Exchange

**Format:** (qp) cmpxchgsz.sem.*ldhint*  $r_1 = [r_3], r_2$ , ar.ccv M16  
 (qp) cmp8xchg16.sem.*ldhint*  $r_1 = [r_3], r_2$ , ar.csd, ar.ccv sixteen\_byte\_form M16

**Description:** A value consisting of *sz* bytes (8 bytes for `cmp8xchg16`) 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  $r_3$ . For `cmp8xchg16`, if the two are equal, then 8-bytes from GR  $r_2$  are stored at the specified address ignoring bit 3 (GR  $r_3$  &  $\sim 0x8$ ), and 8 bytes from the Compare and Store Data application register (AR[CSD]) are stored at that address + 8 ((GR  $r_3$  &  $\sim 0x8$ ) + 8). 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 2-19](#). The *sem* completer specifies the type of semaphore operation. These operations are described in [Table 2-20](#). See [Section 4.4.7, “Sequentiality Attribute and Ordering” on page 2:82](#) for details on memory ordering.

**Table 2-19. Memory Compare and Exchange Size**

<b>sz Completer</b>	<b>Bytes Accessed</b>
1	1
2	2
4	4
8	8

**Table 2-20. Compare and Exchange Semaphore Types**

<b>sem Completer</b>	<b>Ordering Semantics</b>	<b>Semaphore Operation</b>
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). For the `cmp8xchg16` instruction, the address specified must be 8-byte aligned.

The memory read and write are guaranteed to be atomic. For the `cmp8xchg16` instruction, the 8-byte memory read and the 16-byte memory 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 2-34 on page 3:152](#). Locality hints do not

affect program functionality and may be ignored by the implementation. See [Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69](#) for details.

For `cmp8xchg16`, Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See [Section 3.1.11, "Processor Identification Registers" on page 1:34](#) for details.

```
Operation: if (PR[qp]) {
    size = sixteen_byte_form ? 16 : sz;

    if (sixteen_byte_form && !instruction_implemented(CMP8XCHG16))
        illegal_operation_fault();
    check_target_register(r1);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(SEMAPHORE);

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

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

    if (sixteen_byte_form) {
        if (sem == 'acq')
            val = mem_xchg16_cond(AR[CCV], GR[r2], AR[CSD], paddr, UM.be,
                                  mattr, ACQUIRE, ldhint);
        else // 'rel'
            val = mem_xchg16_cond(AR[CCV], GR[r2], AR[CSD], paddr, UM.be,
                                  mattr, RELEASE, ldhint);
    } else {
        if (sem == 'acq')
            val = mem_xchg_cond(AR[CCV], GR[r2], paddr, size, UM.be, mattr,
                                ACQUIRE, ldhint);
        else // 'rel'
            val = mem_xchg_cond(AR[CCV], GR[r2], paddr, size, UM.be, mattr,
                                RELEASE, ldhint);
        val = zero_ext(val, size * 8);
    }

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

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

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

## cover — Cover Stack Frame

**Format:** cover

**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, operation is undefined.

This instruction cannot be predicated.

**Operation:**

```
if (!followed_by_stop())
    undefined_behavior();

if (PSR.cpl == 0 && PSR.vm == 1)
    virtualization_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:** Virtualization fault

## czx — Compute Zero Index

<b>Format:</b>	(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 2-21 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 2-21. 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( $r_1$ );

    if (one_byte_form) {
        if (left_form) { // scan from most significant down
            if (((GR[ $r_3$ ] & 0xffff000000000000) == 0) GR[ $r_1$ ] = 0;
            else if (((GR[ $r_3$ ] & 0x00ff000000000000) == 0) GR[ $r_1$ ] = 1;
            else if (((GR[ $r_3$ ] & 0x0000ff0000000000) == 0) GR[ $r_1$ ] = 2;
            else if (((GR[ $r_3$ ] & 0x000000ff00000000) == 0) GR[ $r_1$ ] = 3;
            else if (((GR[ $r_3$ ] & 0x00000000ff000000) == 0) GR[ $r_1$ ] = 4;
            else if (((GR[ $r_3$ ] & 0x0000000000ff0000) == 0) GR[ $r_1$ ] = 5;
            else if (((GR[ $r_3$ ] & 0x000000000000ff00) == 0) GR[ $r_1$ ] = 6;
            else if (((GR[ $r_3$ ] & 0x00000000000000ff) == 0) GR[ $r_1$ ] = 7;
            else GR[ $r_1$ ] = 8;
        } else { // right_form      scan from least significant up
            if (((GR[ $r_3$ ] & 0x000000000000ffff) == 0) GR[ $r_1$ ] = 0;
            else if (((GR[ $r_3$ ] & 0x000000000000ff00) == 0) GR[ $r_1$ ] = 1;
            else if (((GR[ $r_3$ ] & 0x0000000000000ff0) == 0) GR[ $r_1$ ] = 2;
            else if (((GR[ $r_3$ ] & 0x00000000000000ff) == 0) GR[ $r_1$ ] = 3;
            else if (((GR[ $r_3$ ] & 0x00000ff00000000) == 0) GR[ $r_1$ ] = 4;
            else if (((GR[ $r_3$ ] & 0x0000ff0000000000) == 0) GR[ $r_1$ ] = 5;
            else if (((GR[ $r_3$ ] & 0x00ff000000000000) == 0) GR[ $r_1$ ] = 6;
            else if (((GR[ $r_3$ ] & 0xff00000000000000) == 0) GR[ $r_1$ ] = 7;
            else GR[ $r_1$ ] = 8;
        }
    } else { // two_byte_form
        if (left_form) { // scan from most significant down
            if (((GR[ $r_3$ ] & 0xffff000000000000) == 0) GR[ $r_1$ ] = 0;
            else if (((GR[ $r_3$ ] & 0x0000ffff00000000) == 0) GR[ $r_1$ ] = 1;
            else if (((GR[ $r_3$ ] & 0x000000ff00000000) == 0) GR[ $r_1$ ] = 2;
            else if (((GR[ $r_3$ ] & 0x00000000ffff0000) == 0) GR[ $r_1$ ] = 3;
            else GR[ $r_1$ ] = 4;
        } else { // right_form      scan from least significant up
            if (((GR[ $r_3$ ] & 0x000000000000ffff) == 0) GR[ $r_1$ ] = 0;
            else if (((GR[ $r_3$ ] & 0x000000000000ff00) == 0) GR[ $r_1$ ] = 1;
        }
    }
}

```

```
        else if ((GR[r3] & 0x0000ffff00000000) == 0) GR[r1] = 2;
        else if ((GR[r3] & 0xfffff000000000000) == 0) GR[r1] = 3;
        else GR[r1] = 4;
    }
}
GR[r1].nat = GR[r3].nat;
}
```

**Interruptions:** Illegal Operation fault

## dep — Deposit

<b>Format:</b>	(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

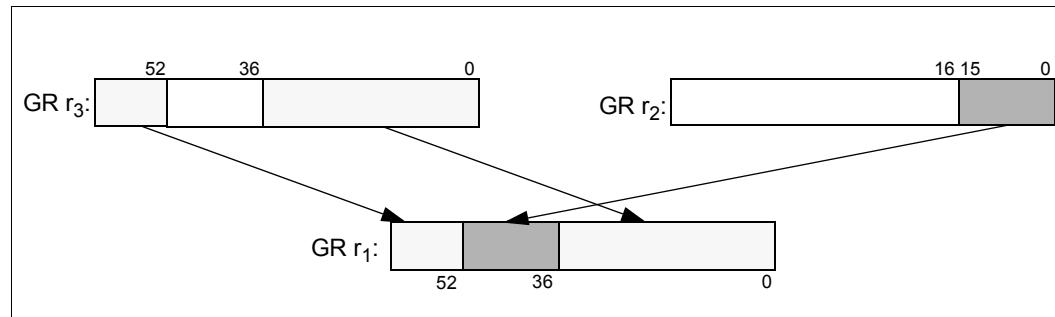
**Description:** In the merge\_form, a right justified bit field taken from the first source operand is deposited into the value in GR  $r_3$  at an arbitrary bit position and the result is placed in GR  $r_1$ . In the register\_form the first source operand is GR  $r_2$ ; and in the imm\_form it is 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.

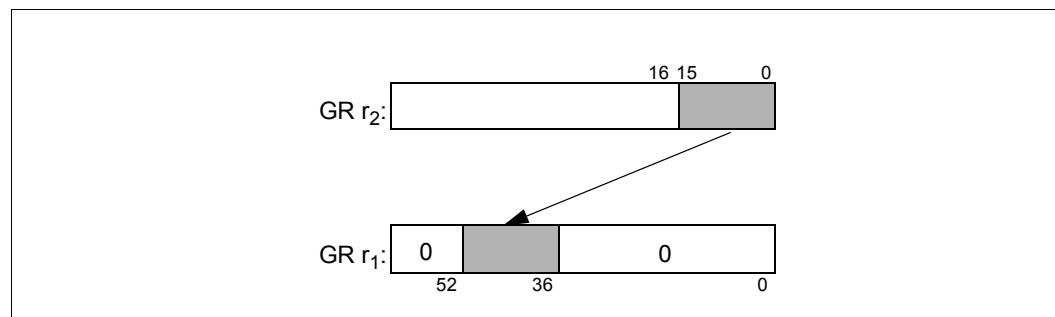
The operation of  $\text{dep } r_1 = r_2, r_3, 36, 16$  is illustrated in [Figure 2-5](#).

**Figure 2-5. Deposit Example (merge\_form)**



The operation of  $\text{dep.z } r_1 = r_2, 36, 16$  is illustrated in [Figure 2-6](#).

**Figure 2-6. Deposit Example (zero\_form)**



**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 > 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

## epc — 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 < PSR.cpl)
    illegal_operation_fault();

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

**Interruptions:** Illegal Operation fault

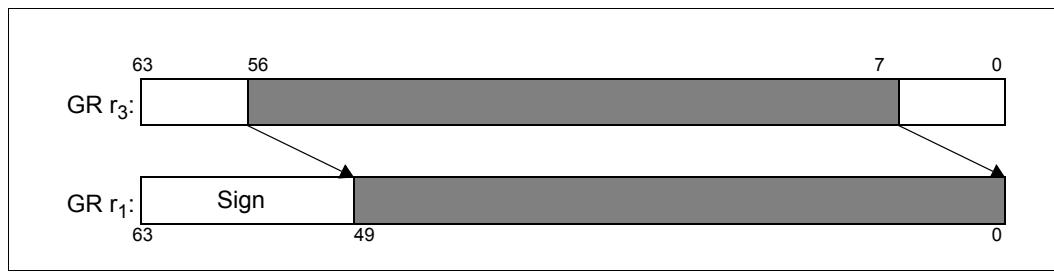
## extr — Extract

**Format:**  $(qp) \text{ extr } r_1 = r_3, pos_6, len_6$       signed\_form      I11  
 $(qp) \text{ extr.u } r_1 = r_3, pos_6, len_6$       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 r1 = r3, 7, 50` is illustrated in [Figure 2-7](#).

**Figure 2-7. Extract Example**



```

Operation: if (PR[qp]) {
    check_target_register(r1);

    tmp_len = len6;

    if (pos6 + tmp_len > 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

## **fabs — Floating-point Absolute Value**

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

pseudo-op of: (*qp*) fmerge.s  $f_1 = f_0, f_3$

**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 “fmerge – Floating-point Merge” on page 3:80.

## **fadd — Floating-point Add**

- Format:** (qp) fadd.pc.sf  $f_1 = f_3, f_2$       pseudo-op of: (qp) fma.pc.sf  $f_1 = f_3, f_1, f_2$
- 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 2-22](#). The mnemonic values for sf are given in [Table 2-23](#). 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 1:90](#).

**Table 2-22. Specified pc Mnemonic Values**

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

**Table 2-23. sf Mnemonic Values**

<b>sf Mnemonic</b>	<b>Status Field Accessed</b>
.s0 or <i>none</i>	sf0
.s1	sf1
.s2	sf2
.s3	sf3

- Operation:** See "fma — Floating-point Multiply Add" on page 3:77.

## famax — 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 2-23 on page 3:56](#).

```
Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, 0$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ])) {
        FR[ $f_1$ ] = NATVAL;
    } else {
        fminmax_exception_fault_check( $f_2, f_3, sf, &tmp_fp_env$ );
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_right = fp_reg_read(FR[ $f_2$ ]);
        tmp_left = fp_reg_read(FR[ $f_3$ ]);
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        FR[ $f_1$ ] = tmp_bool_res ? FR[ $f_2$ ] : FR[ $f_3$ ];

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr( $f_1$ );
}
```

**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

## famin — 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 2-23 on page 3:56](#).

```
Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, 0$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ])) {
        FR[ $f_1$ ] = NATVAL;
    } else {
        fminmax_exception_fault_check( $f_2, f_3, sf, &tmp_fp_env$ );
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_left = fp_reg_read(FR[ $f_2$ ]);
        tmp_right = fp_reg_read(FR[ $f_3$ ]);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        FR[ $f_1$ ] = tmp_bool_res ? FR[ $f_2$ ] : FR[ $f_3$ ];

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr( $f_1$ );
}
```

**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

## fand — 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

## **fandcm — 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  $f_3$  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

## fc — Flush Cache

<b>Format:</b>	(qp) fc <i>r</i> <sub>3</sub>	invalidate_line_form	M28
	(qp) fc.i <i>r</i> <sub>3</sub>	instruction_cache_coherent_form	M28

**Description:** In the invalidate\_line form, the cache line associated with the address specified by the value of GR *r*<sub>3</sub> 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.

In the instruction\_cache\_coherent form, the cache line specified by GR *r*<sub>3</sub> is flushed in an implementation-specific manner that ensures that the instruction caches are coherent with the data caches. The fc.i instruction is not required to invalidate the targeted cache line nor write the targeted cache line back to memory if it is inconsistent with memory, but may do so if this is required to make the instruction caches coherent with the data caches. The fc.i instruction is broadcast throughout the coherence domain if necessary to make all instruction caches coherent. 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 and fc.i perform no access rights or protection key checks. At other privilege levels, fc and fc.i perform access rights checks as if they were 1-byte reads, but do not perform any protection key checks (regardless of PSR.pk).

The memory attribute of the page containing the affected line has no effect on the behavior of these instructions. The fc 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.

These instructions follow data dependency ordering rules; they are ordered only with respect to previous load, store or semaphore instructions to the same line. fc and fc.i have data dependencies in the sense that any prior stores by this processor will be included in the flush operation. Subsequent memory operations to the same line need not wait for prior fc or fc.i completion before being globally visible. fc and fc.i are unordered operations, and are not affected by a memory fence (mf) instruction. These instructions are 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);

    if (invalidate_line_form)
        mem_flush(tmp_paddr);
    else // instruction_cache_coherent_form
        make_icache_coherent(tmp_paddr);
}

```

<b>Interruptions:</b>	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
-----------------------	--	--

## fchkf — Floating-point Check Flags

**Format:** (qp) fchkf.sf *target*<sub>25</sub>

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*<sub>25</sub> is taken.

The *target*<sub>25</sub> operand, specifies a label to branch to. This is encoded in the instruction as a signed immediate displacement (*imm*<sub>21</sub>) between the target bundle and the bundle containing this instruction (*imm*<sub>21</sub> = *target*<sub>25</sub> - 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*<sub>21</sub> 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 2-23 on page 3:56](#).

```
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 (!impl_uia_fault_supported() &&
                ((PSR.it && unimplemented_virtual_address(IP, PSR.vm))
                 || (!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

<b>Interruptions:</b>	Speculative Operation fault	Taken Branch trap
	Unimplemented Instruction Address trap	

## fclass – Floating-point Class

**Format:** (qp) fclass.fcrel.fctype p<sub>1</sub>, p<sub>2</sub> = f<sub>2</sub>, fclass<sub>9</sub>

F5

**Description:** The contents of FR f<sub>2</sub> are classified according to the fclass<sub>9</sub> completer as shown in [Table 2-25](#). This produces a boolean result based on whether the contents of FR f<sub>2</sub> agrees with the floating-point number format specified by fclass<sub>9</sub>, as specified by the fcrel completer. This result is written to the two predicate register destinations, p<sub>1</sub> and p<sub>2</sub>. 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 2-26 on page 3:67](#). The assembly syntax allows the specification of membership or non-membership and the assembler swaps the target predicates to achieve the desired effect.

**Table 2-24. Floating-point Class Relations**

fcrel	Test Relation
m	FR f <sub>2</sub> agrees with the pattern specified by fclass <sub>9</sub> (is a member)
nm	FR f <sub>2</sub> does not agree with the pattern specified by fclass <sub>9</sub> (is not a member)

A number agrees with the pattern specified by fclass<sub>9</sub> if:

- the number is NaTVal and fclass<sub>9</sub> {8} is 1, or
- the number is a quiet NaN and fclass<sub>9</sub> {7} is 1, or
- the number is a signaling NaN and fclass<sub>9</sub> {6} is 1, or
- the sign of the number agrees with the sign specified by one of the two low-order bits of fclass<sub>9</sub>, and the type of the number (disregarding the sign) agrees with the number-type specified by the next four bits of fclass<sub>9</sub>, as shown in [Table 2-25](#).

**Note:** An fclass<sub>9</sub> of 0x1FF is equivalent to testing for any supported operand.

The class names used in [Table 2-25](#) are defined in [Table 5-2, “Floating-point Register Encodings” on page 1:86](#).

**Table 2-25. Floating-point Classes**

fclass <sub>9</sub>	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 four 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

## **fclrf — 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 2-23 on page 3:56](#).

**Operation:**

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

**FP Exceptions:** None

**Interruptions:** None

## fcmp – Floating-point Compare

**Format:** (qp) fcmp.freI.fctype.sf p<sub>1</sub>, p<sub>2</sub> = f<sub>2</sub>, f<sub>3</sub>

F4

**Description:** The two source operands are compared for one of twelve relations specified by *freI*. 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<sub>1</sub> and p<sub>2</sub>. 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 2-26. Floating-point Comparison Types**

<i>fctype</i>	PR[qp]==0		PR[qp]==1					
			Result==0, No Source NaTVals		Result==1, No Source NaTVals		One or More Source NaTVals	
	PR[p <sub>1</sub> ]	PR[p <sub>2</sub> ]	PR[p <sub>1</sub> ]	PR[p <sub>2</sub> ]	PR[p <sub>1</sub> ]	PR[p <sub>2</sub> ]	PR[p <sub>1</sub> ]	PR[p <sub>2</sub> ]
<i>none</i>	0	0	0	1	1	0	0	0
unc	0	0	0	1	1	0	0	0

The mnemonic values for sf are given in [Table 2-23 on page 3:56](#).

The relations are defined for each of the comparison types in [Table 2-27](#). 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 2-27. Floating-point Comparison Relations**

<i>freI</i>	<i>freI</i> Completer Unabbreviated	Relation	Pseudo-op of		Quiet NaN as Operand Signals Invalid
eq	equal	f <sub>2</sub> == f <sub>3</sub>			No
lt	less than	f <sub>2</sub> < f <sub>3</sub>			Yes
le	less than or equal	f <sub>2</sub> <= f <sub>3</sub>			Yes
gt	greater than	f <sub>2</sub> > f <sub>3</sub>	lt	f <sub>2</sub> ↔ f <sub>3</sub>	Yes
ge	greater than or equal	f <sub>2</sub> >= f <sub>3</sub>	le	f <sub>2</sub> ↔ f <sub>3</sub>	Yes
unord	unordered	f <sub>2</sub> ? f <sub>3</sub>			No
neq	not equal	!(f <sub>2</sub> == f <sub>3</sub> )	eq	p <sub>1</sub> ↔ p <sub>2</sub>	No
nlt	not less than	!(f <sub>2</sub> < f <sub>3</sub> )	lt	p <sub>1</sub> ↔ p <sub>2</sub>	Yes
nle	not less than or equal	!(f <sub>2</sub> <= f <sub>3</sub> )	le	p <sub>1</sub> ↔ p <sub>2</sub>	Yes
ngt	not greater than	!(f <sub>2</sub> > f <sub>3</sub> )	lt	f <sub>2</sub> ↔ f <sub>3</sub> p <sub>1</sub> ↔ p <sub>2</sub>	Yes
nge	not greater than or equal	!(f <sub>2</sub> >= f <sub>3</sub> )	le	f <sub>2</sub> ↔ f <sub>3</sub> p <sub>1</sub> ↔ p <sub>2</sub>	Yes
ord	ordered	!(f <sub>2</sub> ? f <sub>3</sub> )	unord	p <sub>1</sub> ↔ p <sub>2</sub>	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

Disabled Floating-point Register fault

Floating-point Exception fault

## fcvt.fx – Convert Floating-point to Integer

<b>Format:</b>	(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 2-23 on page 3:56](#).

```

Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1$ ,  $f_2$ , 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ])) {
        FR[ $f_1$ ] = NATVAL;
        fp_update_psr( $f_1$ );
    } else {
        tmp_default_result = fcvt_exception_fault_check( $f_2$ , 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[ $f_1$ ].significand = INTEGER_INDEFINITE;
            FR[ $f_1$ ].exponent = FP_INTEGER_EXP;
            FR[ $f_1$ ].sign = FP_SIGN_POSITIVE;
        } else {
            tmp_res = fp_ieee_rnd_to_int(fp_reg_read(FR[ $f_2$ ]), &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[ $f_1$ ].significand = tmp_res.significand;
            FR[ $f_1$ ].exponent = FP_INTEGER_EXP;
            FR[ $f_1$ ].sign = FP_SIGN_POSITIVE;
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr( $f_1$ );
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}

```

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

## fcvt.xf – 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

## fcvt.xuf — Convert Unsigned Integer to Floating-point

**Format:** (qp) fcvt.xuf.pc.sf  $f_1 = f_3$       pseudo-op of: (qp) fma.pc.sf  $f_1 = f_3, f_1, f_0$

**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 2-22 on page 3:56](#). The mnemonic values for *sf* are given in [Table 2-23 on page 3:56](#). 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 1:90](#).

**Operation:** See “[fma — Floating-point Multiply Add](#)” on page 3:77.

## fetchadd — Fetch and Add Immediate

**Format:** (qp) fetchadd4.sem.lhint  $r_1 = [r_3], inc_3$       four\_byte\_form M17  
           (qp) fetchadd8.sem.lhint  $r_1 = [r_3], inc_3$       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 2-28](#). See [Section 4.4.7, “Sequentiality Attribute and Ordering” on page 2:82](#) for details on memory ordering.

**Table 2-28. 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. Details on memory attributes are described in [Section 4.4, “Memory Attributes” on page 2:75](#).

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 [Section 4.4.9, “Effects of Memory Attributes on Memory Reference Instructions” on page 2:86](#).

The value of the *lhint* completer specifies the locality of the memory access. The values of the *lhint* completer are given in [Table 2-34 on page 3:152](#). Locality hints do not affect program functionality and may be ignored by the implementation. See [Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69](#) for details.

```

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;
                }

```

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

## flushrs — 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 [Chapter 6, "Register Stack Engine" in 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 and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0; 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();
}
```

<b>Interruptions:</b>	Unimplemented Data Address fault	Data Key Miss fault
	VHPT Data fault	Data Key Permission fault
	Data Nested TLB fault	Data Access Rights fault
	Data TLB fault	Data Dirty Bit fault
	Alternate Data TLB fault	Data Access Bit fault
	Data Page Not Present fault	Data Debug fault
	Data NaT Page Consumption fault	

## fma — 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 “[fmpy — Floating-point Multiply](#)” on page [3:85](#).

The mnemonic values for the opcode’s pc are given in [Table 2-22 on page 3:56](#). The mnemonic values for sf are given in [Table 2-23 on page 3:56](#). 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 1:90](#).

```

Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, f_4$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ]) ||
        fp_is_natval(FR[ $f_4$ ])) {
        FR[ $f_1$ ] = NATVAL;
        fp_update_psr( $f_1$ );
    } else {
        tmp_default_result = fma_exception_fault_check( $f_2, f_3, f_4,$ 
                                                       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[ $f_1$ ] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[ $f_3$ ]), fp_reg_read(FR[ $f_4$ ]));
```

```
            if ( $f_2 \neq 0$ )
                tmp_res = fp_add(tmp_res, fp_reg_read(FR[ $f_2$ ]), tmp_fp_env);
            FR[ $f_1$ ] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }
    }

    fp_update_fpsr(sf, tmp_fp_env);
    fp_update_psr( $f_1$ );
    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

<b>Interruptions:</b>	Illegal Operation fault Disabled Floating-point Register fault	Floating-point Exception fault Floating-point Exception trap
-----------------------	---	---

## fmax – 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 2-23 on page 3:56](#).

```
Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, 0$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ])) {
        FR[ $f_1$ ] = NATVAL;
    } else {
        fminmax_exception_fault_check( $f_2, f_3, sf, &tmp_fp_env$ );
        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[ $f_3$ ]),
                                    fp_reg_read(FR[ $f_2$ ]));  

        FR[ $f_1$ ] = (tmp_bool_res ? FR[ $f_2$ ] : FR[ $f_3$ ]];

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr( $f_1$ );
}
```

**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

## fmerge — Floating-point Merge

<b>Format:</b>	(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  $f_1$ .

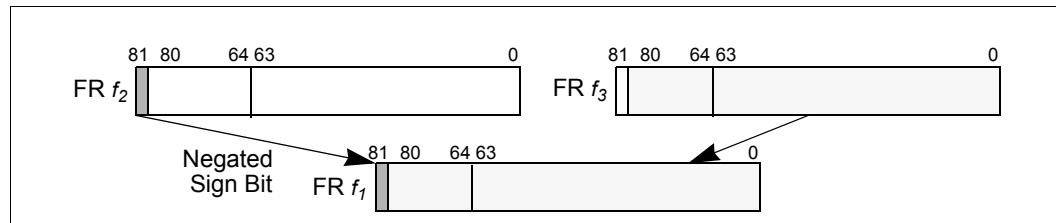
For the neg\_sign\_form, the sign of FR  $f_2$  is negated and concatenated with the exponent and the significand of FR  $f_3$ . This form can be used to negate a floating-point number by using the same register for FR  $f_2$  and FR  $f_3$ .

For the sign\_form, the sign of FR  $f_2$  is concatenated with the exponent and the significand of FR  $f_3$ .

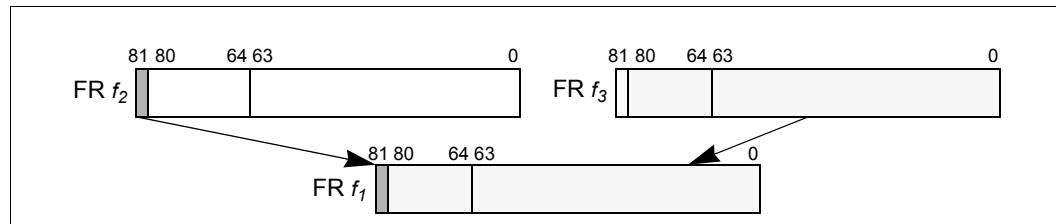
For the sign\_exp\_form, the sign and exponent of FR  $f_2$  is concatenated with the significand of FR  $f_3$ .

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.

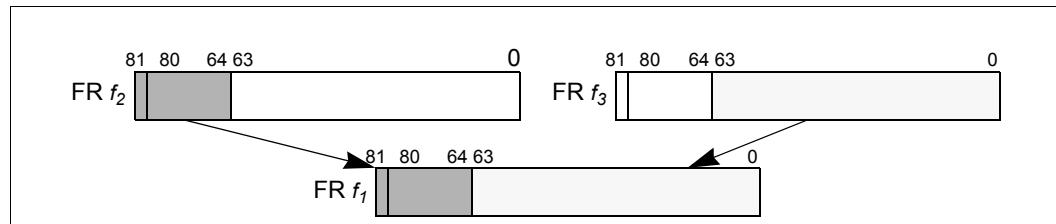
**Figure 2-8. Floating-point Merge Negative Sign Operation**



**Figure 2-9. Floating-point Merge Sign Operation**



**Figure 2-10. Floating-point Merge Sign and Exponent Operation**



```

Operation:    if (PR[qp]) {
                    fp_check_target_register( $f_1$ );
                    if (tmp_isrcode = fp_reg_disabled( $f_1$ ,  $f_2$ ,  $f_3$ , 0))
                        disabled_fp_register_fault(tmp_isrcode, 0);

                    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ]))) {
                        FR[ $f_1$ ] = NATVAL;
                    } else {
                        FR[ $f_1$ ].significand = FR[ $f_3$ ].significand;
                        if (neg_sign_form) {
                            FR[ $f_1$ ].exponent = FR[ $f_3$ ].exponent;
                            FR[ $f_1$ ].sign = !FR[ $f_2$ ].sign;
                        } else if (sign_form) {
                            FR[ $f_1$ ].exponent = FR[ $f_3$ ].exponent;
                            FR[ $f_1$ ].sign = FR[ $f_2$ ].sign;
                        } else {
                            FR[ $f_1$ ].exponent = FR[ $f_2$ ].exponent;
                            FR[ $f_1$ ].sign = FR[ $f_2$ ].sign;
                        }
                    }
                    fp_update_psr( $f_1$ );
                }
            
```

**FP Exceptions:** None

**Interruptions:** Illegal Operation fault

Disabled Floating-point Register fault

## **fmin – 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 2-23 on page 3:56](#).

```
Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, 0$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ])) {
        FR[ $f_1$ ] = NATVAL;
    } else {
        fminmax_exception_fault_check( $f_2, f_3, sf, \&tmp_fp_env$ );
        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[ $f_2$ ]),
                                    fp_reg_read(FR[ $f_3$ ])));
        FR[ $f_1$ ] = tmp_bool_res ? FR[ $f_2$ ] : FR[ $f_3$ ];

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr( $f_1$ );
}
```

**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

## fmix – Floating-point Mix

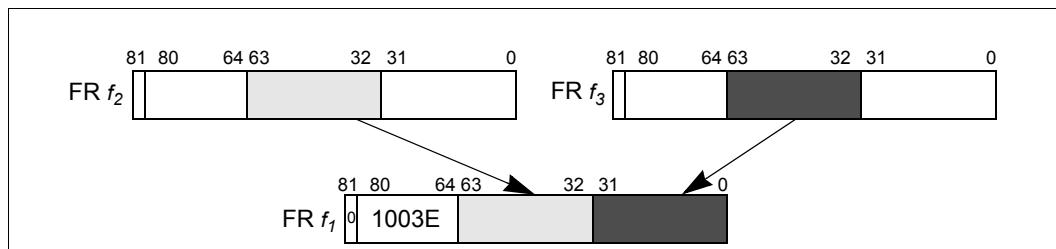
<b>Format:</b>	(qp) fmix.l $f_1 = f_2, f_3$	mix_l_form	F9
	(qp) fmix.r $f_1 = f_2, f_3$	mix_r_form	F9
	(qp) fmix.lr $f_1 = f_2, f_3$	mix_lr_form	F9

**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  $f_3$ .

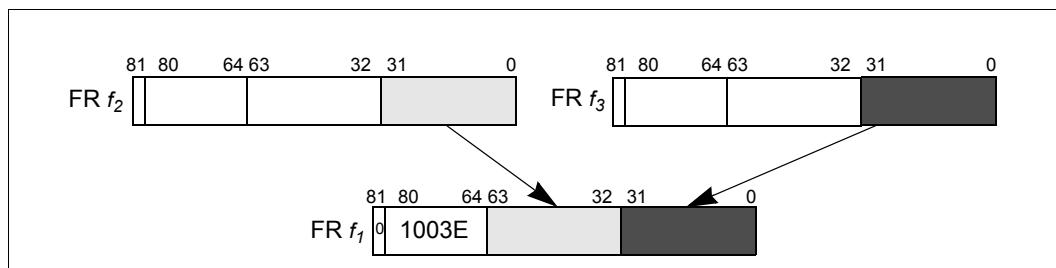
For all forms, the exponent field of FR  $f_1$  is set to the biased exponent for  $2.0^{63}$  (0x1003E) and the sign field of FR  $f_1$  is set to positive (0).

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.

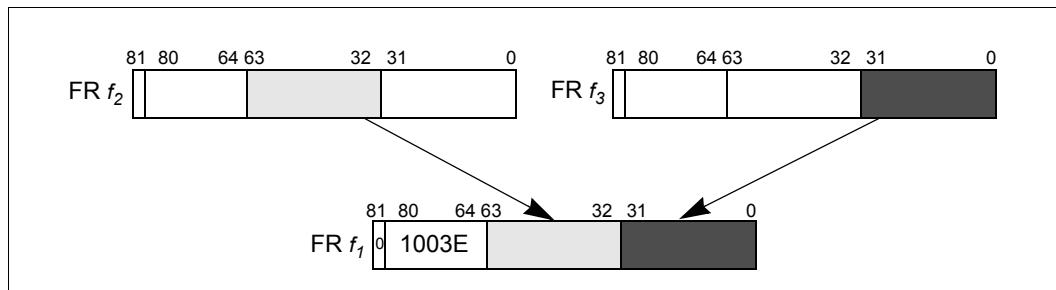
**Figure 2-11. Floating-point Mix Left**



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



**Figure 2-13. Floating-point Mix Left-Right**



```

Operation:   if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1$ ,  $f_2$ ,  $f_3$ , 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natural(FR[ $f_2$ ]) || fp_is_natural(FR[ $f_3$ ])) {
        FR[ $f_1$ ] = NATVAL;
    } else {
        if (mix_l_form) {
            tmp_res_hi = FR[ $f_2$ ].significand{63:32};
            tmp_res_lo = FR[ $f_3$ ].significand{63:32};
        } else if (mix_r_form) {
            tmp_res_hi = FR[ $f_2$ ].significand{31:0};
            tmp_res_lo = FR[ $f_3$ ].significand{31:0};
        } else { // mix_lr_form
            tmp_res_hi = FR[ $f_2$ ].significand{63:32};
            tmp_res_lo = FR[ $f_3$ ].significand{31:0};
        }
        FR[ $f_1$ ].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[ $f_1$ ].exponent = FP_INTEGER_EXP;
        FR[ $f_1$ ].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr( $f_1$ );
}

```

**FP Exceptions:** None

**Interruptions:** Illegal Operation fault

Disabled Floating-point Register fault

## fmpy — Floating-point Multiply

**Format:** (qp) fmpy.pc.sf  $f_1 = f_3, f_4$

pseudo-op of: (qp) fma.pc.sf  $f_1 = f_3, f_4, \text{f}0$

**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 2-22 on page 3:56](#). The mnemonic values for  $sf$  are given in [Table 2-23 on page 3:56](#). 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 1:90](#).

**Operation:**

See “[fma — Floating-point Multiply Add](#)” on page 3:77.

## fms — 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 “[fmpy — Floating-point Multiply](#)” on page [3:85](#).

The mnemonic values for the opcode’s pc are given in [Table 2-22 on page 3:56](#). The mnemonic values for sf are given in [Table 2-23 on page 3:56](#). 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 1:90](#).

```

Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, f_4$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ]) ||
        fp_is_natval(FR[ $f_4$ ])) {
        FR[ $f_1$ ] = NATVAL;
        fp_update_psr( $f_1$ );
    } else {
        tmp_default_result = fms_fnma_exception_fault_check( $f_2, f_3, f_4,$ 
                                                            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[ $f_1$ ] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[ $f_3$ ]), fp_reg_read(FR[ $f_4$ ]));
```

```
            tmp_fr2 = fp_reg_read(FR[ $f_2$ ]);
            tmp_fr2.sign = !tmp_fr2.sign;
            if ( $f_2 \neq 0$ )
                tmp_res = fp_add(tmp_res, tmp_fr2, tmp_fp_env);
            FR[ $f_1$ ] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }
    }

    fp_update_fpsr(sf, tmp_fp_env);
    fp_update_psr( $f_1$ );
    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

<b>Interruptions:</b>	Illegal Operation fault	Floating-point Exception fault
	Disabled Floating-point Register fault	Floating-point Exception trap

## **fneg – Floating-point Negate**

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

**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 “fmerge – Floating-point Merge” on page 3:80.

## **fnegabs – Floating-point Negate Absolute Value**

**Format:**  $(qp)$  fmerge.ns  $f_1 = f_3$  pseudo-op of:  $(qp)$  fmerge.ns  $f_1 = \text{f0}, f_3$

pseudo-op of: (*qp*) fmerge.ns  $f_1 = f_0, f_3$

**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 “fmerge – Floating-point Merge” on page 3:80.

## fnma — 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 “[fnmpy — Floating-point Negative Multiply](#)” on page 3:92.

The mnemonic values for the opcode’s pc are given in [Table 2-22 on page 3:56](#). The mnemonic values for sf are given in [Table 2-23 on page 3:56](#). 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 1:90](#).

```

Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, f_4$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ]) ||
        fp_is_natval(FR[ $f_4$ ])) {
        FR[ $f_1$ ] = NATVAL;
        fp_update_psr( $f_1$ );
    } else {
        tmp_default_result = fms_fnma_exception_fault_check( $f_2, f_3, f_4,$ 
                                                            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[ $f_1$ ] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[ $f_3$ ]), fp_reg_read(FR[ $f_4$ ]));
```

```
            tmp_res.sign = !tmp_res.sign;
            if ( $f_2 \neq 0$ )
                tmp_res = fp_add(tmp_res, fp_reg_read(FR[ $f_2$ ]), tmp_fp_env);
            FR[ $f_1$ ] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }
    }

    fp_update_fpsr(sf, tmp_fp_env);
    fp_update_psr( $f_1$ );
    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

<b>Interruptions:</b>	Illegal Operation fault	Floating-point Exception fault
	Disabled Floating-point Register fault	Floating-point Exception trap

## **fnmpy – 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 2-22 on page 3:56](#). The mnemonic values for sf are given in [Table 2-23 on page 3:56](#). 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 1:90](#).

**Operation:** See “[fnma – Floating-point Negative Multiply Add](#)” on page 3:90.

## fnorm — Floating-point Normalize

**Format:** (qp) fnorm.pc.sf  $f_1 = f_3$

pseudo-op of: (qp) fma.pc.sf  $f_1 = f_3, f_1, f_0$

**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 2-22 on page 3:56](#). The mnemonic values for *sf* are given in [Table 2-23 on page 3:56](#). 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 1:90](#).

**Operation:**

See "fma — Floating-point Multiply Add" on page 3:77.

## for — 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

## fpabs — Floating-point Parallel Absolute Value

**Format:** (qp) fpabs  $f_1 = f_3$

pseudo-op of: (qp) fpmerge.s  $f_1 = f_0, f_3$

**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 “[fpmerge — Floating-point Parallel Merge](#)” on page 3:111.

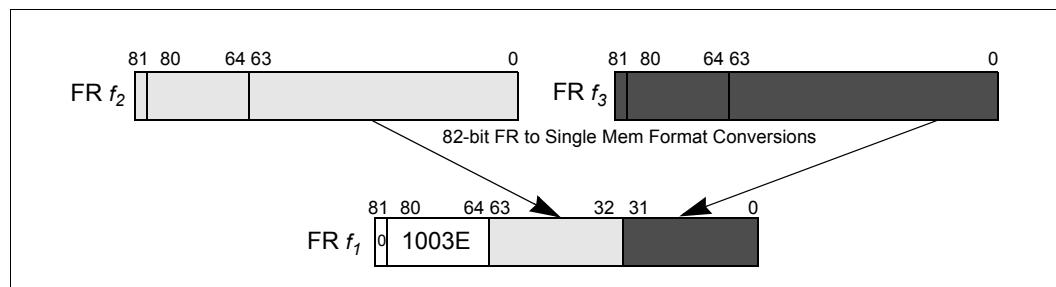
## fpack — Floating-point Pack

**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 2-14. 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

## fpamax — 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 2-23 on page 3:56](#).

```
Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, 0$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ])) {
        FR[ $f_1$ ] = NATVAL;
    } else {
        fpminmax_exception_fault_check( $f_2, f_3, sf, \&tmp_fp_env$ );
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_right = fp_reg_read_hi( $f_2$ );
        tmp_fr3 = tmp_left = fp_reg_read_hi( $f_3$ );
        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( $f_2$ );
        tmp_fr3 = tmp_left = fp_reg_read_lo( $f_3$ );
        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[ $f_1$ ].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[ $f_1$ ].exponent = FP_INTEGER_EXP;
        FR[ $f_1$ ].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr( $f_1$ );
}
```

**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

## fpamin — 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 2-23 on page 3:56](#).

```
Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, 0$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ])))
        FR[ $f_1$ ] = NATVAL;
    } else {
        fpminmax_exception_fault_check( $f_2, f_3, sf, \&tmp_fp_env$ );
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_left = fp_reg_read_hi( $f_2$ );
        tmp_fr3 = tmp_right = fp_reg_read_hi( $f_3$ );
        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( $f_2$ );
        tmp_fr3 = tmp_right = fp_reg_read_lo( $f_3$ );
        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[ $f_1$ ].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[ $f_1$ ].exponent = FP_INTEGER_EXP;
        FR[ $f_1$ ].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr( $f_1$ );
}
```

**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

## fpcmp — 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 2-29. 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 2-23 on page 3:56](#).

The relations are defined for each of the comparison types in [Table 2-29](#). 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 2-30. Floating-point Parallel Comparison Relations**

<i>frel</i>	<i>frel</i> 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	$f_2 \leq f_3$			Yes
gt	greater than	$f_2 > f_3$	lt	$f_2 \leftrightarrow f_3$	Yes
ge	greater than or equal	$f_2 \geq f_3$	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	$!(f_2 \leq f_3)$			Yes
ngt	not greater than	$!(f_2 > f_3)$	nlt	$f_2 \leftrightarrow f_3$	Yes
nge	not greater than or equal	$!(f_2 \geq f_3)$	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

## **fpcvt.fx — Convert Parallel Floating-point to Integer**

<b>Format:</b>	(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 2-23 on page 3:56](#).

```

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)

<b>Interruptions:</b>	Illegal Operation fault Disabled Floating-point Register fault	Floating-point Exception fault Floating-point Exception trap
-----------------------	---	---

## fpma — 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 “[fpmpy — Floating-point Parallel Multiply](#)” on page 3:115.) 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 2-23 on page 3:56](#).

The encodings and interpretation for the status field’s rc are given in [Table 5-6 on page 1:90](#).

```

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)

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

## fpmax – 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 fpcomp.lt operation.

The mnemonic values for sf are given in [Table 2-23 on page 3:56](#).

```
Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, 0$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ])) {
        FR[ $f_1$ ] = NATVAL;
    } else {
        fpminmax_exception_fault_check( $f_2, f_3, sf, &tmp_fp_env$ );
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_right = fp_reg_read_hi( $f_2$ );
        tmp_fr3 = tmp_left = fp_reg_read_hi( $f_3$ );
        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( $f_2$ );
        tmp_fr3 = tmp_left = fp_reg_read_lo( $f_3$ );
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2 : tmp_fr3);

        FR[ $f_1$ ].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[ $f_1$ ].exponent = FP_INTEGER_EXP;
        FR[ $f_1$ ].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr( $f_1$ );
}
```

**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

## fpmerge – Floating-point Parallel Merge

<b>Format:</b>	(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  $f_3$ .

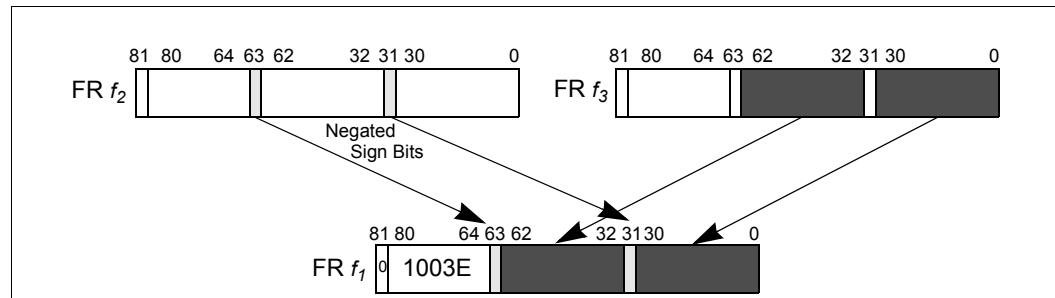
For the sign\_form, the signs of the pair of single precision values in the significand field of FR  $f_2$  are concatenated with the exponents and the significands of the pair of single precision values in the significand field of FR  $f_3$  and stored in FR  $f_1$ .

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  $f_1$ .

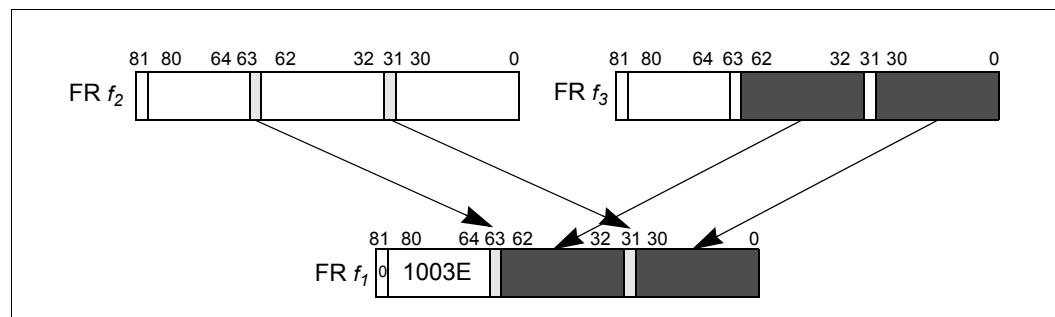
For all forms, the exponent field of FR  $f_1$  is set to the biased exponent for  $2.0^{63}$  (0x1003E) and the sign field of FR  $f_1$  is set to positive (0).

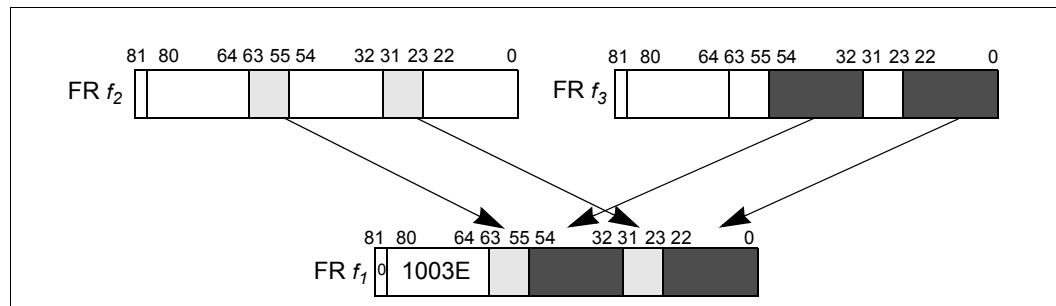
For all forms, if either FR  $f_2$  or FR  $f_3$  is a NaTVal, FR  $f_1$  is set to NaTVal instead of the computed result.

**Figure 2-15. Floating-point Parallel Merge Negative Sign Operation**



**Figure 2-16. Floating-point Parallel Merge Sign Operation**



**Figure 2-17. Floating-point Parallel 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

## fpmin — 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 fpcomp.lt operation.

The mnemonic values for sf are given in [Table 2-23 on page 3:56](#).

```
Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, 0$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ])) {
        FR[ $f_1$ ] = NATVAL;
    } else {
        fpminmax_exception_fault_check( $f_2, f_3, sf, &tmp_fp_env$ );
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_left = fp_reg_read_hi( $f_2$ );
        tmp_fr3 = tmp_right = fp_reg_read_hi( $f_3$ );
        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( $f_2$ );
        tmp_fr3 = tmp_right = fp_reg_read_lo( $f_3$ );
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[ $f_1$ ].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[ $f_1$ ].exponent = FP_INTEGER_EXP;
        FR[ $f_1$ ].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr( $f_1$ );
}
```

**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

## fpm<sub>py</sub> – Floating-point Parallel Multiply

**Format:** (qp) fpm<sub>py</sub>.sf  $f_1 = f_3, f_4$       pseudo-op of: (qp) fpma.sf  $f_1 = f_3, f_4, f_0$

**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 2-23 on page 3:56](#).

The encodings and interpretation for the status field's rc are given in [Table 5-6 on page 1:90](#).

**Operation:** See “fpma – Floating-point Parallel Multiply Add” on page 3:107.

## fpms — 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 2-23 on page 3:56](#).

The encodings and interpretation for the status field's rc are given in [Table 5-6 on page 1:90](#).

```

Operation: if (PR[qp]) {
    fp_check_target_register( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1, f_2, f_3, f_4$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ]) ||
        fp_is_natval(FR[ $f_4$ ])) {
        FR[ $f_1$ ] = NATVAL;
        fp_update_psr( $f_1$ );
    } else {
        tmp_default_result_pair = fpms_fpnma_exception_fault_check( $f_2, f_3,$ 
                                                                 $f_4, sf, &tmp_fp_env$ );
        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( $f_3$ ), fp_reg_read_hi( $f_4$ ));
            if ( $f_2 \neq 0$ ) {
                tmp_sub = fp_reg_read_hi( $f_2$ );
                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( $f_3$ ), fp_reg_read_lo( $f_4$ ));
    if ( $f_2 \neq 0$ ) {
        tmp_sub = fp_reg_read_lo( $f_2$ );
        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)

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

## **fpneg — Floating-point Parallel Negate**

<b>Format:</b>	(qp) fpneg $f_1 = f_3$	pseudo-op of: (qp) fpmerge.ns $f_1 = f_3, f_3$
<b>Description:</b>	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.	
<b>Operation:</b>	See “ <a href="#">fpmerge — Floating-point Parallel Merge</a> ” on page 3:111.	

**fpnegabs — Floating-point Parallel Negate Absolute Value**

**Format:** (qp) fpnegabs  $f_1 = f_3$       pseudo-op of: (qp) fpmerge.ns  $f_1 = f_0, f_3$

**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 “[fpmerge — Floating-point Parallel Merge](#)” on page 3:111.

## fpnma — 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 2-23 on page 3:56](#).

The encodings and interpretation for the status field's rc are given in [Table 5-6 on page 1:90](#).

```

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)  
 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

## **fpnmpy — Floating-point Parallel Negative Multiply**

<b>Format:</b>	(qp) fpnmpy.sf $f_1 = f_3, f_4$	pseudo-op of: (qp) fpnma.sf $f_1 = f_3, f_4, f_0$
<b>Description:</b> 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 <a href="#">Table 2-23 on page 3:56</a> . The encodings and interpretation for the status field's rc are given in <a href="#">Table 5-6 on page 1:90</a> .		
<b>Operation:</b>	See “fpnma — Floating-point Parallel Negative Multiply Add” on <a href="#">page 3:120</a> .	

## fprcpa — 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 2-23 on page 3:56](#).

```
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  
Disabled Floating-point Register fault      Floating-point Exception fault

## fprsqrt — Floating-point Parallel Reciprocal Square Root Approximation

**Format:** (qp) fprsqrt.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 frsqrt 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 2-23 on page 3:56](#).

**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 = fprsqrt_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

Disabled Floating-point Register fault

Floating-point Exception fault

## frcpa — Floating-point Reciprocal Approximation

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

**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 2-23 on page 3:56](#).

```
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:** Illegal Operation fault

Disabled Floating-point Register fault

Floating-point Exception fault

## frsqrta — 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 2-23 on page 3:56](#).

**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_natural(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  
Disabled Floating-point Register fault      Floating-point Exception fault

## **fselect — 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

## fsetc — Floating-point Set Controls

**Format:** (qp) fsetc.sf *amask*<sub>7</sub>, *omask*<sub>7</sub>

F12

**Description:** The status field's control bits are initialized to the value obtained by logically AND-ing the sf0.controls and *amask*<sub>7</sub> immediate field and logically OR-ing the *omask*<sub>7</sub> immediate field.

The mnemonic values for *sf* are given in [Table 2-23 on page 3:56](#).

```
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

## **fsub — Floating-point Subtract**

- Format:** (qp) fsub.pc.sf  $f_1 = f_3, f_2$       pseudo-op of: (qp) fms.pc.sf  $f_1 = f_3, f_1, f_2$
- 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 2-22 on page 3:56](#). The mnemonic values for *sf* are given in [Table 2-23 on page 3:56](#). 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 1:90](#).
- Operation:** See “[fms — Floating-point Multiply Subtract](#)” on page 3:86.

## fswap — Floating-point Swap

<b>Format:</b>	(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.

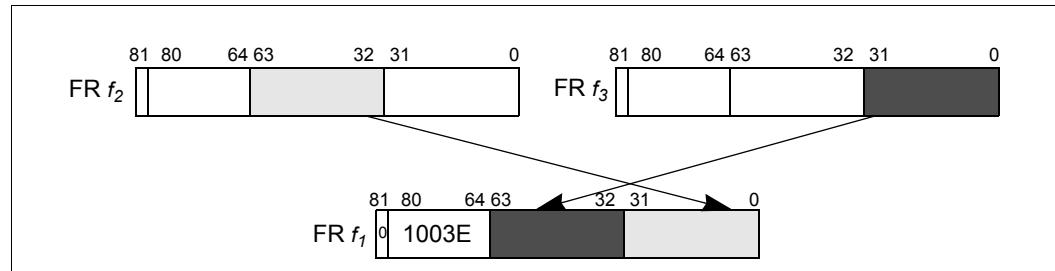
For the swap\_nl\_form, the left single precision value in FR  $f_2$  is concatenated with the right single precision value in FR  $f_3$ . The concatenated pair is then swapped, and the left single precision value is negated.

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.

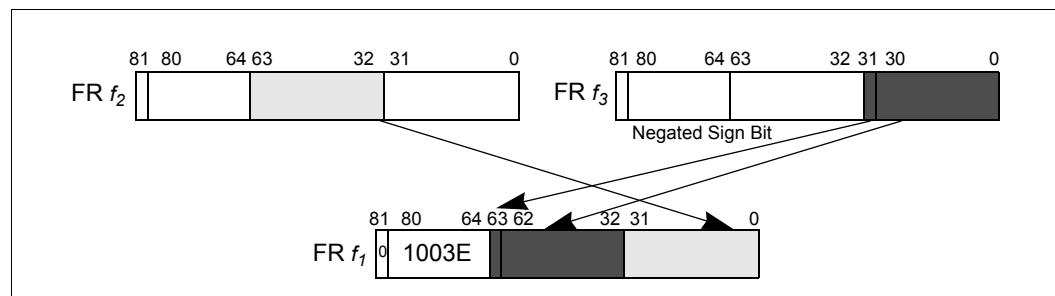
For all forms, the exponent field of FR  $f_1$  is set to the biased exponent for  $2.0^{63}$  (0x1003E) and the sign field of FR  $f_1$  is set to positive (0).

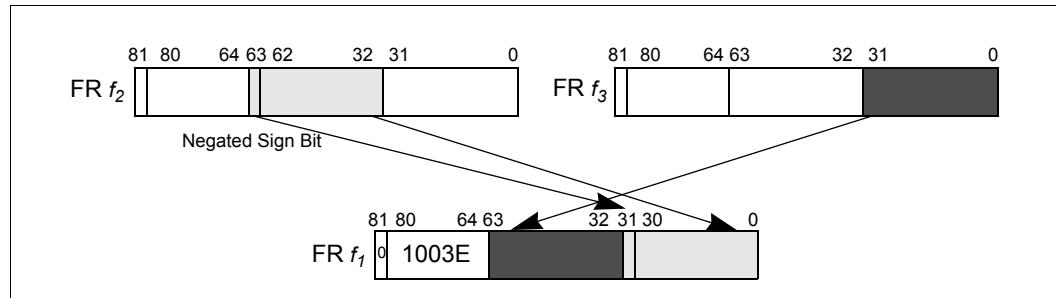
For all forms, if either FR  $f_2$  or FR  $f_3$  is a NaTVal, FR  $f_1$  is set to NaTVal instead of the computed result.

**Figure 2-18. Floating-point Swap**



**Figure 2-19. Floating-point Swap Negate Left**



**Figure 2-20. Floating-point Swap Negate 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

## **fsxt – Floating-point Sign Extend**

**Format:**            (qp) fsxt.l  $f_1 = f_2, f_3$   
                       (qp) fsxt.r  $f_1 = f_2, f_3$

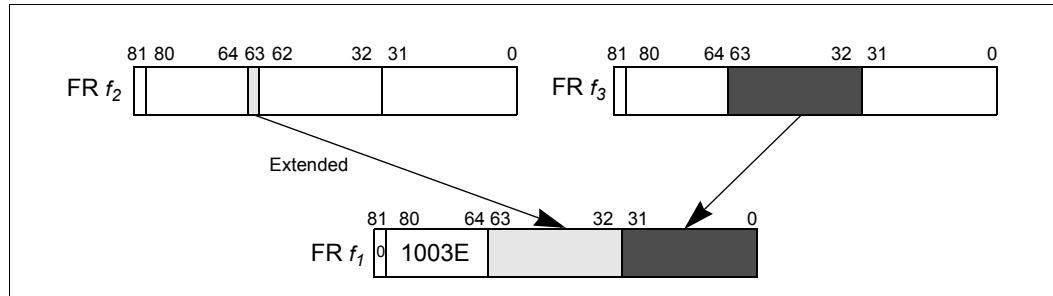
sxt\_l\_form F9  
sxt\_r\_form F9

**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  $f_3$ .

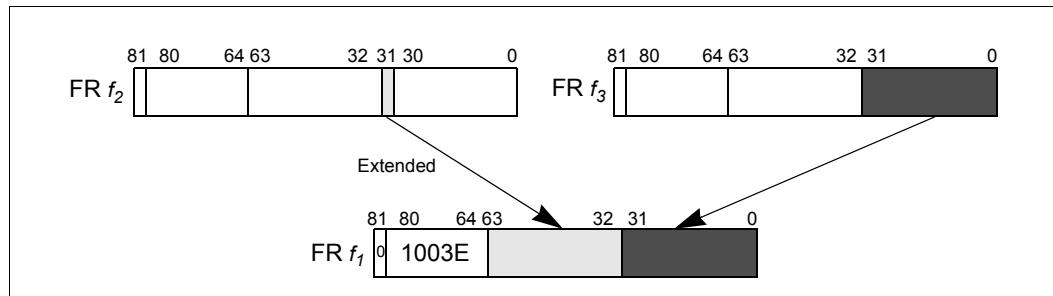
For all forms, the exponent field of FR  $f_1$  is set to the biased exponent for  $2.0^{63}$  (0x1003E) and the sign field of FR  $f_1$  is set to positive (0).

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 2-21.** Floating-point Sign Extend Left



**Figure 2-22. 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_natural(FR[f2]) || fp_is_natural(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

## fwb — 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 4-15 on page 2:83](#) 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

## fxor — 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

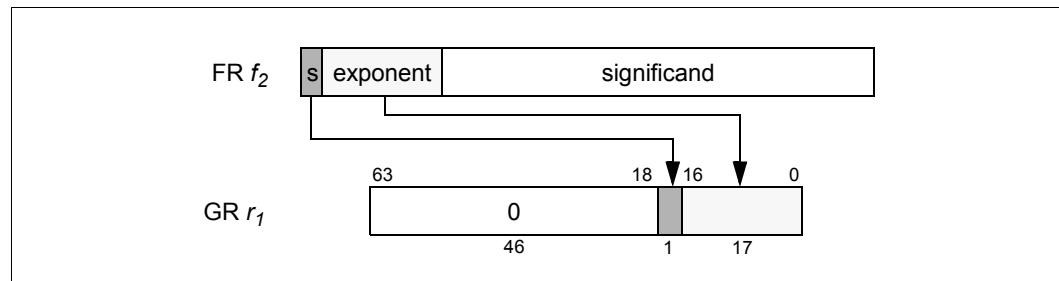
## getf — Get Floating-point Value or Exponent or Significand

<b>Format:</b>	(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  $r_1$ , as shown in [Figure 5-7](#) and [Figure 5-8 on page 1:95](#), respectively. In the single\_form, the most-significant 32 bits of GR  $r_1$  are set to 0.

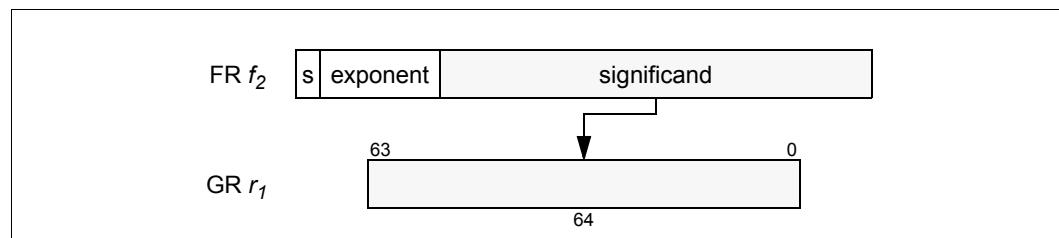
In the exponent\_form, the exponent field of FR  $f_2$  is copied to bits 16:0 of GR  $r_1$  and the sign bit of the value in FR  $f_2$  is copied to bit 17 of GR  $r_1$ . The most-significant 46-bits of GR  $r_1$  are set to zero.

**Figure 2-23. 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 2-24. 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_natural(FR[f2]))
        GR[r1].nat = 1;
    else
        GR[r1].nat = 0;
}

```

**Interruptions:** Illegal Operation fault

Disabled Floating-point Register fault

## hint — Performance Hint

<b>Format:</b>	(qp) hint <i>imm</i> <sub>21</sub>	pseudo-op
	(qp) hint.i <i>imm</i> <sub>21</sub>	i_unit_form I18
	(qp) hint.b <i>imm</i> <sub>21</sub>	b_unit_form B9
	(qp) hint.m <i>imm</i> <sub>21</sub>	m_unit_form M48
	(qp) hint.f <i>imm</i> <sub>21</sub>	f_unit_form F16
	(qp) hint.x <i>imm</i> <sub>62</sub>	x_unit_form X5

**Description:** Provides a performance hint to the processor about the program being executed. It has no effect on architectural machine state, and operates as a `nop` instruction except for its performance effects.

The immediate, *imm*<sub>21</sub> or *imm*<sub>62</sub>, specifies the hint. For the x\_unit\_form, the L slot of the bundle contains the upper 41 bits of *imm*<sub>62</sub>.

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.

**Table 2-31. Hint Immediates**

<i>imm</i> <sub>21</sub> or <i>imm</i> <sub>62</sub>	Mnemonic	Hint
0x0	@pause	Indicates to the processor that the currently executing stream is waiting, spinning, or performing low priority tasks. This hint can be used by the processor to allocate more resources or time to another executing stream on the same processor. For the case where the currently executing stream is spinning or otherwise waiting for a particular address in memory to change, an advanced load to that address should be done before executing a <code>hint @pause;</code> this hint can be used by the processor to resume normal allocation of resources or time to the currently executing stream at the point when some other stream stores to that address.
0x1	@priority	Indicates to the processor that the currently executing stream is performing a high priority task. This hint can be used by the processor to allocate more resources or time to this stream. Implementations will ensure that such increased allocation is only temporary, and that repeated use of this hint will not impair longer-term fairness of allocation.
0x02-0x3f		These values are available for future architected extensions and will execute as a <code>nop</code> on all current processors. Use of these values may cause unexpected performance issues on future processors and should not be used.
other		Implementation specific. Performs an implementation-specific hint action. Consult processor model-specific documentation for details.

**Operation:**

```

if (PR[qp]) {
    if (x_unit_form)
        hint = imm62;
    else // i_unit_form || b_unit_form || b_unit_form || f_unit_form
        hint = imm21;

    if (is_supported_hint(hint))
        execute_hint(hint);
}

```

**Interruptions:** None

## **invala – Invalidate ALAT**

<b>Format:</b>	(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

## itc — 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 2:52](#).

This instruction can only be executed at the most privileged level, and when PSR.ic and PSR.vm are both 0.

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 (!impl_check_mov_ifa() &&
                        unimplemented_virtual_address(CR[IFA], PSR.vm))
                        unimplemented_data_address_fault(0);
                    if (PSR.vm == 1)
                        virtualization_fault();

                    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);
                    }
                }
}

```

<b>Interruptions:</b>	Machine Check abort Illegal Operation fault Privileged Operation fault Register NaT Consumption fault	Reserved Register/Field fault Unimplemented Data Address fault Virtualization fault
-----------------------	--	---

<b>Serialization:</b>	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.
-----------------------	--

## **itr – Insert Translation Register**

**Format:**  $(qp) \text{ itr.i } \text{itr}[r_3] = r_2$  instruction\_form M42  
 $(qp) \text{ itr.d } \text{dtr}[r_3] = r_2$  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, “Purge Behavior of TLB Inserts and Purges” on page 2-52](#), 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 necessarily purged from the processor's TLBs and may remain as a TC entry. To ensure that the previous TR translation is purged, software must use explicit `ptr` instructions before inserting the new TR entry.

This instruction can only be executed at the most privileged level, and when PSR.ic and PSR.vm are both 0.

```

Operation: if (PR[gp]) {
    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 (!impl_check_mov_ifa() &&
        unimplemented_virtual_address(CR[IFA], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    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);
    }
}

```

<b>Interruptions:</b>	Machine Check abort Illegal Operation fault Privileged Operation fault Register NaT Consumption fault	Reserved Register/Field fault Unimplemented Data Address fault Virtualization fault
<b>Serialization:</b>	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.	
<b>Notes:</b>	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.	

## Id — Load

<b>Format:</b>	(qp) <code>lds<sub>sz</sub>.ldtype.ldhint r<sub>1</sub> = [r<sub>3</sub>]</code>	no_base_update_form	M2
	(qp) <code>lds<sub>sz</sub>.ldtype.ldhint r<sub>1</sub> = [r<sub>3</sub>], r<sub>2</sub></code>	reg_base_update_form	M2
	(qp) <code>lds<sub>sz</sub>.ldtype.ldhint r<sub>1</sub> = [r<sub>3</sub>], imm<sub>9</sub></code>	imm_base_update_form	M3
	(qp) <code>ld16.ldhint r<sub>1</sub>, ar.csd = [r<sub>3</sub>]</code>	sixteen_byte_form, no_base_update_form	M2
	(qp) <code>ld16.acq.ldhint r<sub>1</sub>, ar.csd = [r<sub>3</sub>]</code>	sixteen_byte_form, acquire_form, no_base_update_form	M2
	(qp) <code>ld8.fill.ldhint r<sub>1</sub> = [r<sub>3</sub>]</code>	fill_form, no_base_update_form	M2
	(qp) <code>ld8.fill.ldhint r<sub>1</sub> = [r<sub>3</sub>], r<sub>2</sub></code>	fill_form, reg_base_update_form	M2
	(qp) <code>ld8.fill.ldhint r<sub>1</sub> = [r<sub>3</sub>], imm<sub>9</sub></code>	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*<sub>3</sub>. The value is then zero extended and placed in GR *r*<sub>1</sub>. The values of the *sz* completer are given in [Table 2-32](#). The NaT bit corresponding to GR *r*<sub>1</sub> is cleared, except as described below for speculative loads. The *ldtype* completer specifies special load operations, which are described in [Table 2-33](#).

For the sixteen\_byte\_form, two 8-byte values are loaded as a single, 16-byte memory read. The value at the lowest address is placed in GR *r*<sub>1</sub>, and the value at the highest address is placed in the Compare and Store Data application register (AR[CSD]). The only load types supported for this sixteen\_byte\_form are *none* and *acq*.

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 [Section 4.4.4, “Control Speculation” on page 1:60](#) for details.

In the base update forms, the value in GR *r*<sub>3</sub> is added to either a signed immediate value (*imm*<sub>9</sub>) or a value from GR *r*<sub>2</sub>, and the result is placed back in GR *r*<sub>3</sub>. 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*<sub>2</sub> is set, then the NaT bit corresponding to GR *r*<sub>3</sub> is set and no fault is raised. Base register update is not supported for the ld16 instruction.

**Table 2-32. sz Completers**

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

**Table 2-33. Load Types**

ldtype Completer	Interpretation	Special Load Operation
<i>none</i>	Normal load	
<i>s</i>	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.
<i>a</i>	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 2-33. Load Types (Continued)**

<i>ldtype</i> 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 more details on ordered, biased, speculative, advanced and check loads see [Section 4.4.4, “Control Speculation” on page 1:60](#) and [Section 4.4.5, “Data Speculation” on page 1:63](#). For more details on ordered loads see [Section 4.4.7, “Memory Access Ordering” on page 1:73](#). See [Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69](#) for details on biased loads. Details on memory attributes are described in [Section 4.4, “Memory Attributes” on page 2:75](#).

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 2-34](#). 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. See [Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69](#) for details.

**Table 2-34. Load Hints**

<i>ldhint</i> Completer	Interpretation
none	Temporal locality, level 1

**Table 2-34. Load Hints (Continued)**

<i>ldhint</i> Completer	Interpretation
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.

Hardware support for `ld16` 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 `ld16` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.

For the sixteen\_byte\_form, Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See [Section 3.1.11, “Processor Identification Registers” on page 1:34](#) for details.

```

Operation:    if (PR[qp]) {
                    size = fill_form ? 8 : (sixteen_byte_form ? 16 : 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 = (acquire_form || ldtype == 'acq' || ldtype == 'c.clr.acq');
                    otype = acquire ? ACQUIRE : UNORDERED;
                    bias = (ldtype == 'bias') ? BIAS : 0 ;
                    translate_address = 1;
                    read_memory = 1;

                    itype = READ;
                    if (speculative) itype |= SPEC ;
                    if (advanced) itype |= ADVANCE ;
                    if (size == 16) itype |= UNCACHE_OPT ;

                    if (sixteen_byte_form && !instruction_implemented(LD16))
                        illegal_operation_fault();
                    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)) {
                        translate_address = alat_translate_address_on_hit(ldtype, GENERAL,
r1);
                        read_memory = alat_read_memory_on_hit(ldtype, GENERAL, r1);
                    }
                    if (!translate_address) {
                        if (check_clear || advanced) // remove any old alat entry
                            alat_inval_single_entry(GENERAL, r1);
                    } else {
                        if (!defer) {
                            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                &defer);
                            spontaneous_deferral(paddr, size, UM.be, mattr, otype,
                                bias | ldhint, &defer);
                        if (!defer && read_memory) {
                            if (size == 16) {
                                mem_read_pair(&val, &val_ar, paddr, size, UM.be, mattr,
                                    otype, ldhint);
                            }
                        else {

```

```

        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
        if (size == 16) {
            GR[r1] = val;
            AR[CSD] = val_ar;
        }
        else {
            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(ldtype, 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);
}

```

<b>Interruptions:</b>	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 Unsupported Data Reference fault
-----------------------	--	--

## ldf — Floating-point Load

<b>Format:</b>	(qp) ldfsz.fldtype.ldhint $f_1 = [r_3]$	no_base_update_form	M9
	(qp) ldfsz.fldtype.ldhint $f_1 = [r_3], r_2$	reg_base_update_form	M7
	(qp) ldfsz.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	M9
	(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	M9
	(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 [Section 5.1, “Data Types and Formats” on page 1:85](#) for details on conversion to floating-point register format. The values of the  $fsz$  completer are given in [Table 2-35](#). The  $fldtype$  completer specifies special load operations, which are described in [Table 2-36](#).

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 [Section 4.4.4, “Control Speculation” on page 1:60](#) 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.

**Table 2-35.  $fsz$  Completers**

$fsz$ Completer	Bytes Accessed	Memory Format
s	4 bytes	Single precision
d	8 bytes	Double precision
e	10 bytes	Extended precision

**Table 2-36. FP Load Types**

<i>fldtype</i> 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.
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.

**Table 2-36. FP Load Types (Continued)**

<i>fldtype</i> Completer	Interpretation	Special Load Operation
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.

For more details on speculative, advanced and check loads see [Section 4.4.4, “Control Speculation” on page 1:60](#) and [Section 4.4.5, “Data Speculation” on page 1:63](#). Details on memory attributes are described in [Section 4.4, “Memory Attributes” on page 2:75](#).

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* modifier specifies the locality of the memory access. The mnemonic values of *ldhint* are given in [Table 2-34 on page 3:152](#). 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. See [Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69](#) for details.

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;
    translate_address = 1;
    read_memory = 1;

    itype = READ;
    if (speculative) itype |= SPEC;
    if (advanced) itype |= ADVANCE;
    if (size == 10) itype |= UNCACHE_OPT;

    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)) {
        translate_address = alat_translate_address_on_hit(fldtype, FLOAT, f1);
        read_memory = alat_read_memory_on_hit(fldtype, FLOAT, f1);
    }

    if (!translate_address) {
        if (check_clear || advanced)           // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                   &defer);
            spontaneous_deferral(paddr, size, UM.be, mattr, UNORDERED,
                                  ldhint, &defer);
            if (!defer && read_memory)
                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(fldtype, FLOAT, f1, paddr, size);
        }
    }
}

```

```

        }

    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[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);
}

```

<b>Interruptions:</b>	Illegal Operation fault	Data NaT Page Consumption fault
	Disabled Floating-point Register fault	Data Key Miss fault
	Register NaT Consumption fault	Data Key Permission fault
	Unimplemented Data Address fault	Data Access Rights fault
	Data Nested TLB fault	Data Access Bit fault
	Alternate Data TLB fault	Data Debug fault
	VHPT Data fault	Unaligned Data Reference fault
	Data TLB fault	Unsupported Data Reference fault
	Data Page Not Present fault	

## ldfp — Floating-point Load Pair

<b>Format:</b>	(qp) <code>ldfps.fldtype.ldhint f<sub>1</sub>, f<sub>2</sub> = [r<sub>3</sub>]</code>	single_form, no_base_update_form	M11
	(qp) <code>ldfps.fldtype.ldhint f<sub>1</sub>, f<sub>2</sub> = [r<sub>3</sub>], 8</code>	single_form, base_update_form	M12
	(qp) <code>ldfpd.fldtype.ldhint f<sub>1</sub>, f<sub>2</sub> = [r<sub>3</sub>]</code>	double_form, no_base_update_form	M11
	(qp) <code>ldfpd.fldtype.ldhint f<sub>1</sub>, f<sub>2</sub> = [r<sub>3</sub>], 16</code>	double_form, base_update_form	M12
	(qp) <code>ldfp8.fldtype.ldhint f<sub>1</sub>, f<sub>2</sub> = [r<sub>3</sub>]</code>	integer_form, no_base_update_form	M11
	(qp) <code>ldfp8.fldtype.ldhint f<sub>1</sub>, f<sub>2</sub> = [r<sub>3</sub>], 16</code>	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 [Section 5.1, “Data Types and Formats” on page 1:85](#) for details on conversion to floating-point register format. The *fldtype* completer specifies special load operations, which are described in [Table 2-36 on page 3:157](#).

For more details on speculative, advanced and check loads see [Section 4.4.4, “Control Speculation” on page 1:60](#) and [Section 4.4.5, “Data Speculation” on page 1:63](#).

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 2-34 on page 3:152](#). 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. See [Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69](#) for details.

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;
                    translate_address = 1;
                    read_memory = 1;

                    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)) {
                        translate_address = alat_translate_address_on_hit(fldtype, FLOAT, f1);
                        read_memory = alat_read_memory_on_hit(fldtype, FLOAT, f1);
                    }

                    if (!translate_address) {
                        if (check_clear || advanced)           // remove any old ALAT entry
                            alat_inval_single_entry(FLOAT, f1);
                    } else {
                        if (!defer) {
                            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                                   &defer);
                            spontaneous_deferral(paddr, size, UM.be, mattr, UNORDERED,
                                                  ldhint, &defer);
                            if (!defer && read_memory)
                                mem_read_pair(&f1_val, &f2_val, 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
        FR[f1] = fp_mem_to_fr_format(f1_val, size/2, integer_form);
        FR[f2] = fp_mem_to_fr_format(f2_val, size/2, integer_form);

        if ((check_no_clear || advanced) && ma_is_speculative(mattr))
            // add entry to ALAT
            alat_write(fldtype, 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);
}

```

<b>Interruptions:</b>	Illegal Operation fault	Data Page Not Present fault
	Disabled Floating-point Register fault	Data NaT Page Consumption fault
	Register NaT Consumption fault	Data Key Miss fault
	Unimplemented Data Address fault	Data Key Permission fault
	Data Nested TLB fault	Data Access Rights fault
	Alternate Data TLB fault	Data Access Bit fault
	VHPT Data fault	Data Debug fault
	Data TLB fault	Unaligned Data Reference fault

## lfetch – Line Prefetch

<b>Format:</b>	(qp) lfetch.lftype.lfhint [r <sub>3</sub> ]	no_base_update_form	M18
	(qp) lfetch.lftype.lfhint [r <sub>3</sub> ], r <sub>2</sub>	reg_base_update_form	M20
	(qp) lfetch.lftype.lfhint [r <sub>3</sub> ], imm <sub>9</sub>	imm_base_update_form	M22
	(qp) lfetch.lftype.excl.lfhint [r <sub>3</sub> ]	no_base_update_form, exclusive_form	M18
	(qp) lfetch.lftype.excl.lfhint [r <sub>3</sub> ], r <sub>2</sub>	reg_base_update_form, exclusive_form	M20
	(qp) lfetch.lftype.excl.lfhint [r <sub>3</sub> ], imm <sub>9</sub>	imm_base_update_form, exclusive_form	M22

**Description:** The line containing the address specified by the value in GR r<sub>3</sub> 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, “Memory Access Instructions” on page 1:57](#) for details. The mnemonic values of *lfhint* are given in [Table 2-38](#).

The behavior of the memory read is also determined by the memory attribute associated with the accessed page. See [Chapter 4, “Addressing and Protection” in 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 2-37](#) defines these two options.

**Table 2-37. lftype Mnemonic Values**

<i>lftype</i> Mnemonic	Interpretation
none	No faults are raised
fault	Raise faults

In the base update forms, after being used to address memory, the value in GR r<sub>3</sub> is incremented by either the sign-extended value in imm<sub>9</sub> (in the imm\_base\_update\_form) or the value in GR r<sub>2</sub> (in the reg\_base\_update\_form). In the reg\_base\_update\_form, if the NaT bit corresponding to GR r<sub>2</sub> is set, then the NaT bit corresponding to GR r<sub>3</sub> 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<sub>3</sub> is clear, then the address specified by the value in GR r<sub>3</sub> 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<sub>3</sub> is not modified and no implicit prefetch hint is implied.

If the NaT bit corresponding to GR r<sub>3</sub> 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<sub>3</sub> 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 2-38. lfhint Mnemonic Values**

<i>lfhint</i> Mnemonic	Interpretation
<i>none</i>	Temporal locality, level 1
<i>nt1</i>	No temporal locality, level 1
<i>nt2</i>	No temporal locality, level 2
<i>nta</i>	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.

Both the non-faulting and the faulting forms of `lfetch` can be used speculatively. The purpose of raising faults on the faulting form is to allow the operating system to resolve problems with the address to the extent that it can do so relatively quickly. If problems with the address cannot be resolved quickly, the OS simply returns to the program, and forces the data prefetch to be skipped over.

Specifically, if a faulting `lfetch` takes any of the listed faults (other than Illegal Operation fault), the operating system must handle this fault to the extent that it can do so relatively quickly and invisibly to the interrupted program. If the fault cannot be handled quickly or cannot be handled invisibly (e.g., if handling the fault would involve terminating the program), the OS must return to the interrupted program, skipping over the data prefetch. This can easily be done by setting the IPSR.ed bit to 1 before executing an `rfi` to go back to the process, which will allow the `lfetch.fault` to perform its base register post-increment (if specified), but will suppress any prefetch request and hence any prefetch-related fault. Note that the OS can easily identify that a faulting `lfetch` was the cause of the fault by observing that ISR.na is 1, and ISR.code{3:0} is 4. The one exception to this is the Illegal Operation fault, which can be caused by an `lfetch.fault` if base register post-increment is specified, and the base register is outside of the current stack frame, or is GR0. Since this one fault is not related to the prefetch aspect of `lfetch.fault`, but rather to the base update portion, Illegal Operation faults on `lfetch.fault` should be handled the same as for any other instruction.

```

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);
                }

```

<b>Interruptions:</b>	Illegal Operation fault	Data Page Not Present fault
	Register NaT Consumption fault	Data NaT Page Consumption fault
	Unimplemented Data Address fault	Data Key Miss fault
	Data Nested TLB fault	Data Key Permission fault
	Alternate Data TLB fault	Data Access Rights fault
	VHPT Data fault	Data Access Bit fault
	Data TLB fault	Data Debug fault

## loadrs — 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 and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0; 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();
```

<b>Interruptions:</b>	Illegal Operation fault	Data NaT Page Consumption fault
	Unimplemented Data Address fault	Data Key Miss fault
	Data Nested TLB fault	Data Key Permission fault
	Alternate Data TLB fault	Data Access Rights fault
	VHPT Data fault	Data Access Bit fault
	Data TLB fault	Data Debug fault
	Data Page Not Present fault	

## **mf – 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.

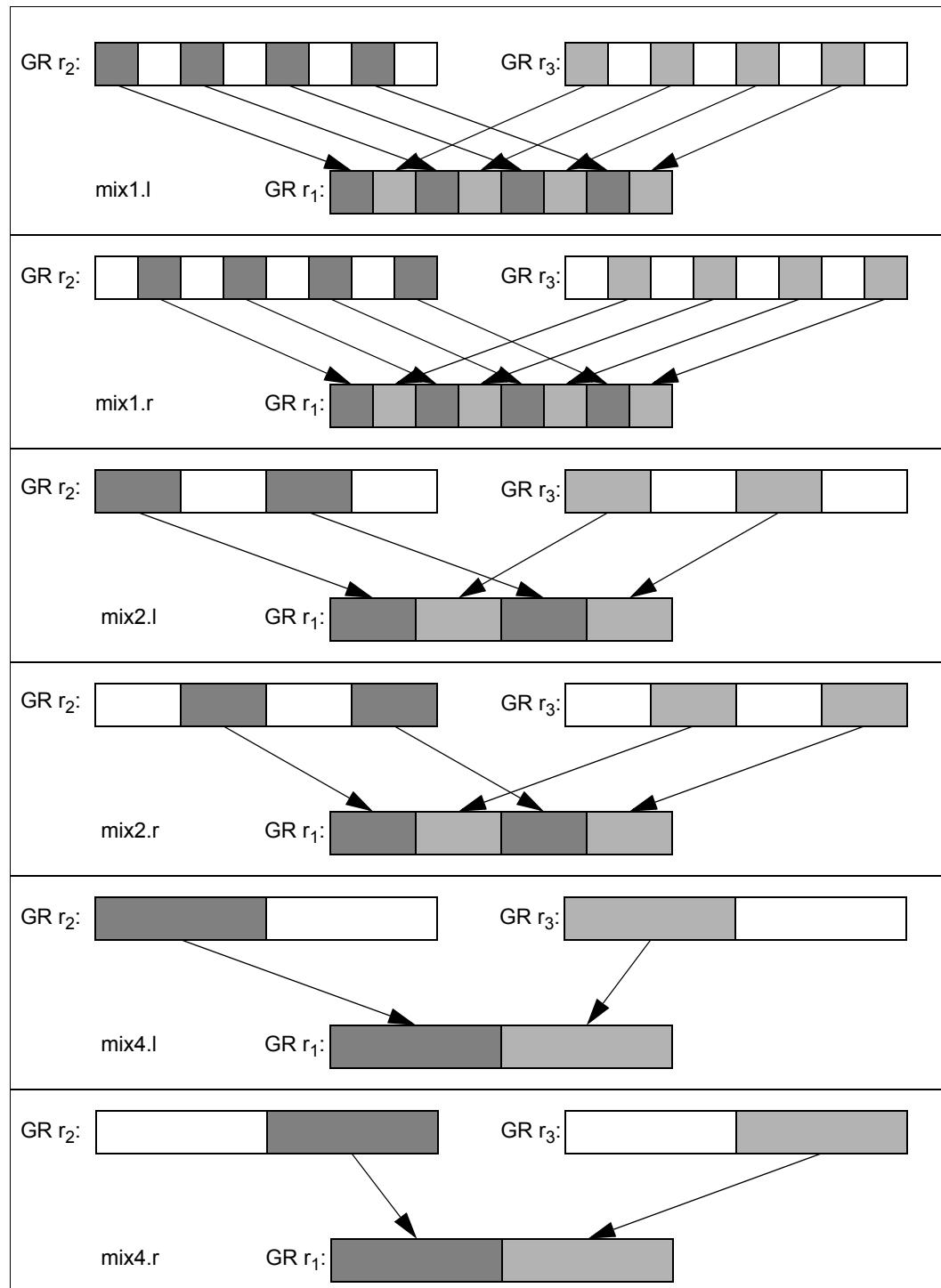
The definition of “acceptance” is platform dependent. The acceptance\_form is typically used to ensure the processor has “waited” until a memory-mapped I/O transaction has been “accepted” before initiating additional external transactions. The acceptance\_form does not ensure ordering, or acceptance to memory areas other than sequential pages.

**Interruptions:** None

## mix — Mix

<b>Format:</b>	(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 2-25](#), 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.

**Figure 2-25. Mix 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 (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

## mov — Move Application Register

<b>Format:</b>	(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 1:28](#) 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 || ar3 == RUC) && PSR.si && PSR.cpl != 0)
                            privileged_register_fault();

                        if ((ar3 == ITC || ar3 == RUC) && PSR.si && PSR.vm == 1)
                            virtualization_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_reg(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 || ar3 == RUC) && (PSR.cpl != 0))
                            privileged_register_fault();

                        if ((ar3 == ITC || ar3 == RUC) && PSR.vm == 1)
                            virtualization_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
                  Register NaT Consumption fault
                  Reserved Register/Field fault

```

Privileged Register fault  
Virtualization fault

## mov — Move Branch Register

<b>Format:</b>	(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 2-39](#). For a description of the *ih* hint completer see the Branch Prediction instruction and [Table 2-13](#) on [page 3:32](#).

**Table 2-39. Move to BR Whether Hints**

<i>mwh</i> Completer	Move to BR Whether Hint
<i>none</i>	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<sub>13</sub>* 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

## **mov – Move Control Register**

**Format:**  $(qp) \text{ mov } r_1 = cr_3$  from\_form M33  
 $(qp) \text{ mov } cr_3 = r_2$  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, and when PSR.vm is 0. Reading or writing an interruption control register (CR16-CR27), 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 (PSR.vm == 1)
            virtualization_fault();
        if (cr3 == IVR)
            check_interrupt_request();

        if (cr3 == ITIR)
            GR[r1] = impl_itir_cwi_mask(CR[ITIR]);
        else
            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 == IFA) && impl_check_mov_ifa() &&
            unimplemented_virtual_address(GR[r2], PSR.vm))
            unimplemented_data_address_fault(0);
        if (PSR.vm == 1)
            virtualization_fault();
        if (cr3 == EOI)
            end_of_interrupt();

        tmp_val = ignored_field_mask(CR_TYPE, cr3, GR[r2]);
        CR[cr3] = tmp_val;
        if (cr3 == IIIPA)
    }
}

```

```
        last_IP = tmp_val;  
    }  
}
```

<b>Interruptions:</b>	Illegal Operation fault Privileged Operation fault Register NaT Consumption fault	Reserved Register/Field fault Unimplemented Data Address fault Virtualization fault
<b>Serialization:</b>	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.	

## **mov — 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 “fmerge — Floating-point Merge” on page 3:80.

## **mov — Move General Register**

**Format:**      *(qp) mov r<sub>1</sub> = r<sub>3</sub>*

pseudo-op of: *(qp) adds r<sub>1</sub> = 0, r<sub>3</sub>*

**Description:**   The value of GR *r<sub>3</sub>* is copied to GR *r<sub>1</sub>*.

**Operation:**   See “[add — Add](#)” on page [3:14](#).

## **mov — 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 — Add” on page 3:14.

## mov — Move Indirect Register

<b>Format:</b>	(qp) mov $r_1 = \text{ireg}[r_3]$	from_form	M43
	(qp) mov $\text{ireg}[r_3] = r_2$	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 2-40](#) 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 2-40. Indirect Register File Mnemonics**

<i>ireg</i>	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 (PSR.vm == 1 && ireg != PMD_TYPE)
        virtualization_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)
            || ireg == CPUID_TYPE
            || is_reserved_field(ireg, tmp_index, GR[r2]))
            reserved_register_field_fault();
        if (PSR.vm == 1)
            virtualization_fault();
        if (ireg == PKR_TYPE && GR[r2]{0} == 1) { // writing valid prot key
            if ((tmp_slot = tlb_search_pktr(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;
    }
}
}
```

<b>Interruptions:</b>	Illegal Operation fault Privileged Operation fault Register NaT Consumption fault	Reserved Register/Field fault Virtualization 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.

## **mov — 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( $r_1$ );
    GR[ $r_1$ ] = IP;
    GR[ $r_1$ ].nat = 0;
}
```

**Interruptions:** Illegal Operation fault

## mov — Move Predicates

<b>Format:</b>	(qp) mov $r_1 = \text{pr}$	from_form	I25
	(qp) mov $\text{pr} = r_2, \text{mask}_{17}$	to_form	I23
	(qp) mov $\text{pr.rot} = \text{imm}_{44}$	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  $\text{mask}_{17}$  are written. The value  $\text{mask}_{17}$  is encoded in the instruction in an  $\text{imm}_{16}$  field such that:  $\text{imm}_{16} = \text{mask}_{17} >> 1$ . Predicate register 0 is always one. The  $\text{mask}_{17}$  value is sign extended. The most significant bit of  $\text{mask}_{17}$ , 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  $\text{imm}_{44}$  operand (which is encoded in the instruction in an  $\text{imm}_{28}$  field, such that:  $\text{imm}_{28} = \text{imm}_{44} >> 16$ ). 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

## **mov – Move Processor Status Register**

**Description:** The source operand is copied to the destination register. See [Section 3.3.2, "Processor Status Register \(PSR\)" on page 2:23](#).

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 {63:32} are ignored. Bits {31:0} of GR  $r_2$  corresponding to reserved fields of the PSR must be 0 or a Reserved Register/Field fault will result. An implementation may also raise Reserved Register/Field fault if bits {63:32} in GR  $r_2$  corresponding to reserved fields of the PSR are non-zero.

Moves to and from the PSR can only be performed at the most privileged level, and when PSR.vm is 0.

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) {
        if (PSR.vm == 1)
            virtualization_fault();
        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();

        if (PSR.vm == 1)
            virtualization_fault();
    }

    PSR{31:0} = GR[r2]{31:0};
}
}

```

<b>Interruptions:</b>	Illegal Operation fault Privileged Operation fault Register NaT Consumption fault	Reserved Register/Field fault Virtualization 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.

## **mov – Move User Mask**

**Description:** The source operand is copied to the destination register.

For move from user mask,  $\text{PSR}\{5:0\}$  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};
                    }
                }

```

<b>Interruptions:</b>	Illegal Operation fault Register NaT Consumption fault	Reserved Register/Field fault
-----------------------	---	-------------------------------

**Serialization:** All user mask modifications are observed by the next instruction group.

## movl – 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

## **mpy4 – Unsigned Integer Multiply**

**Format:** (qp) `mpy4 r1 = r2, r3`

I2

**Description:** The lower 32 bits of each of the two source operands are treated as unsigned values and are multiplied, and the result is placed in GR  $r_1$ . The upper 32 bits of each of the source operands are ignored.

```
Operation: if (PR[qp]) {
    if (!instruction_implemented(mpy4))
        illegal_operation_fault();
    check_target_register(r1);
    GR[r1] = zero_ext(GR[r2], 32) * zero_ext(GR[r3], 32);
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

**Interruptions:** Illegal Operation fault

## mpyshl4 — Unsigned Integer Shift Left and Multiply

**Format:** (qp) mpyshl4  $r_1 = r_2, r_3$

I2

**Description:** The upper 32 bits of GR  $r_2$  and the lower 32 bits of GR  $r_3$  are treated as unsigned values and are multiplied. The result of the multiplication is shifted left 32 bits, with the vacated bit positions filled with zeroes, and the result is placed in GR  $r_1$ . The lower 32 bits of GR  $r_2$  and the upper 32 bits of GR  $r_3$  are ignored.

This instruction can be used to perform a 64-bit integer multiply operation producing a 64-bit result ( $r_c = r_a * r_b$ ):

```
mpy4      r1 = r_a, r_b;; //partial product low 32 bits * low 32 bits
mpyshl4   r2 = r_a, r_b;; //partial product high 32 bits * low 32 bits
mpyshl4   r3 = r_b, r_a   //partial product low 32 bits * high 32 bits
add       r1 = r1, r2;; //partial sum
add       r_c = r1, r3   //final sum
```

```
Operation: if (PR[qp]) {
    if (!instruction_implemented(MPYSHL4))
        illegal_operation_fault();
    check_target_register(r1);

    GR[r1] = (zero_ext((GR[r2] >> 32), 32) * zero_ext(GR[r3], 32)) << 32;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

**Interruptions:** Illegal Operation fault

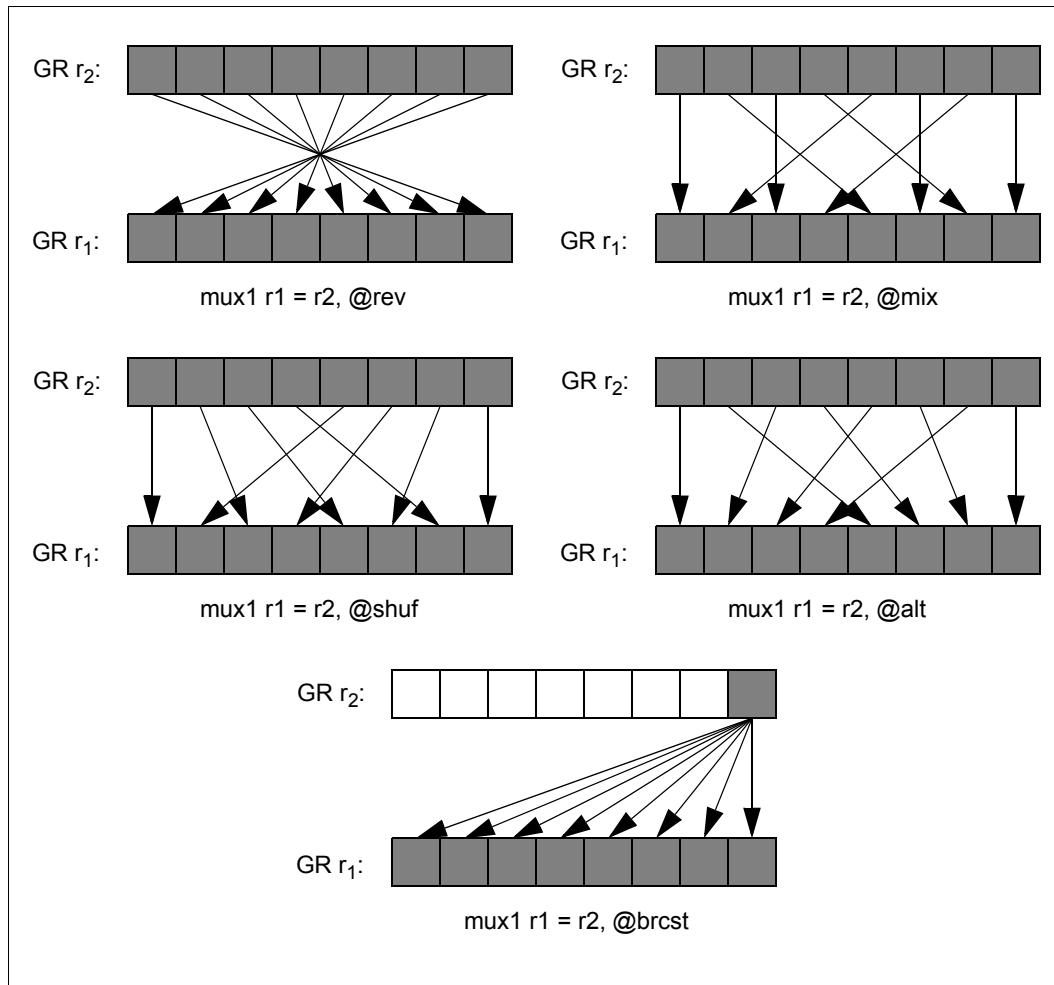
**mux — Mux**

**Format:**  $(qp) \text{ mux1 } r_1 = r_2, mbtype_4$  one\_byte\_form I3  
 $(qp) \text{ mux2 } r_1 = r_2, mhype_8$  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 2-41](#) and shown in [Figure 2-26](#).

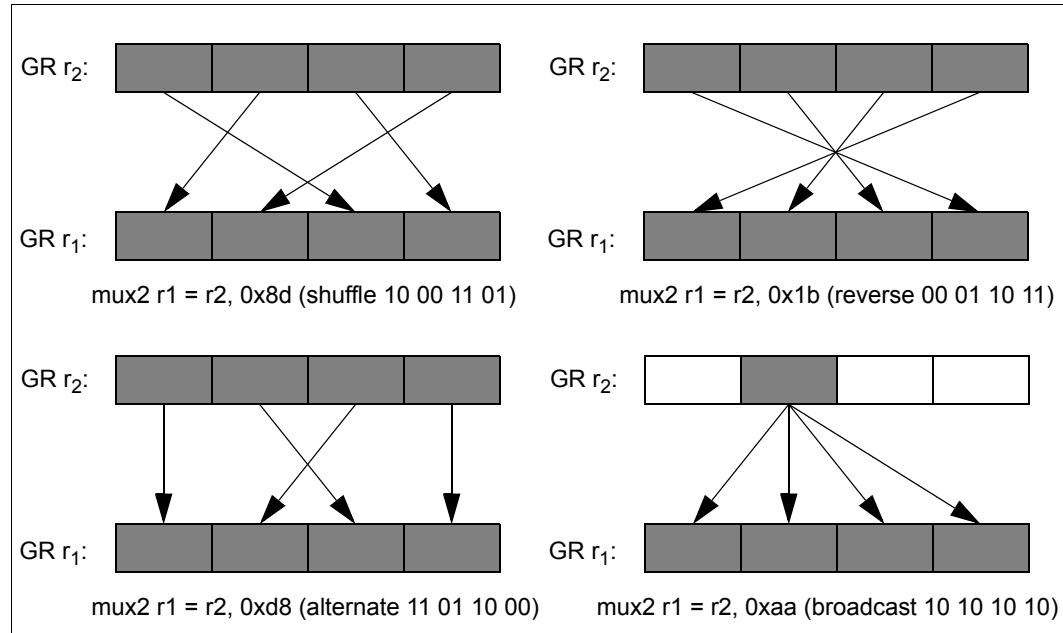
**Table 2-41. Mux Permutations for 8-bit Elements**

<i>mbtype<sub>4</sub></i>	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 $r_2$
@br cst	Perform a Broadcast operation on the least significant byte of GR $r_2$

**Figure 2-26. 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 2-27. 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

## nop — No Operation

<b>Format:</b>	(qp) nop <i>imm</i> <sub>21</sub>	pseudo-op	
	(qp) nop.i <i>imm</i> <sub>21</sub>	i_unit_form	I18
	(qp) nop.b <i>imm</i> <sub>21</sub>	b_unit_form	B9
	(qp) nop.m <i>imm</i> <sub>21</sub>	m_unit_form	M48
	(qp) nop.f <i>imm</i> <sub>21</sub>	f_unit_form	F16
	(qp) nop.x <i>imm</i> <sub>62</sub>	x_unit_form	X5

**Description:** No operation is done.

The immediate, *imm*<sub>21</sub> or *imm*<sub>62</sub>, 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*<sub>62</sub>.

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

*or*

## or — Logical Or

**Format:**  $(qp) \text{ or } r_1 = r_2, r_3$  register\_form A1  
                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 – Pack

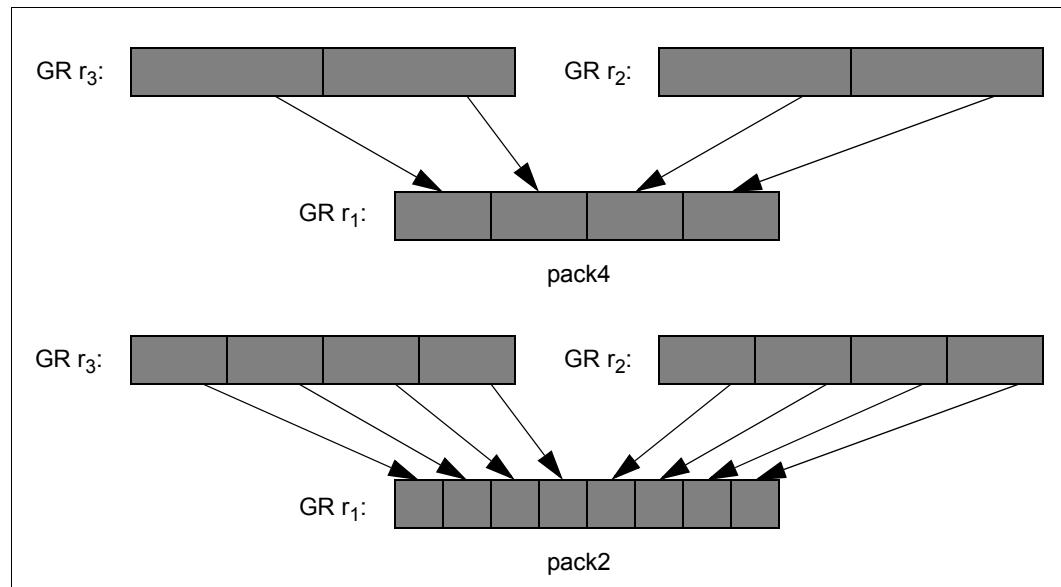
<b>Format:</b>	(qp) pack2.sss $r_1 = r_2, r_3$	two_byte_form, signed_saturation_form	I2
	(qp) pack2.uss $r_1 = r_2, r_3$	two_byte_form, unsigned_saturation_form	I2
	(qp) pack4.sss $r_1 = r_2, r_3$	four_byte_form, signed_saturation_form	I2

**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 2-42](#).

**Table 2-42. 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 2-28. 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(0xffff, 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

## padd – Parallel Add

<b>Format:</b>	(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 2-43](#). 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 2-44](#).

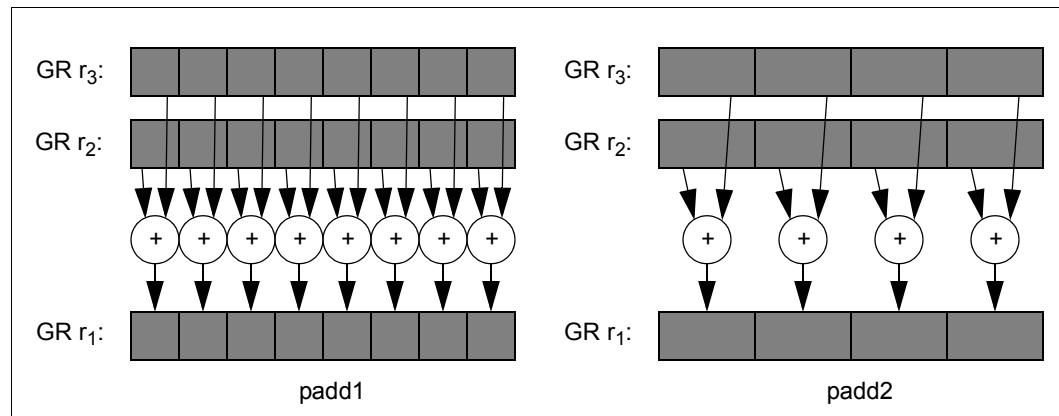
**Table 2-43. Parallel Add Saturation Completers**

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 2-44. 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 2-29. 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;
}

```

*padd*

**Interruptions:** Illegal Operation fault

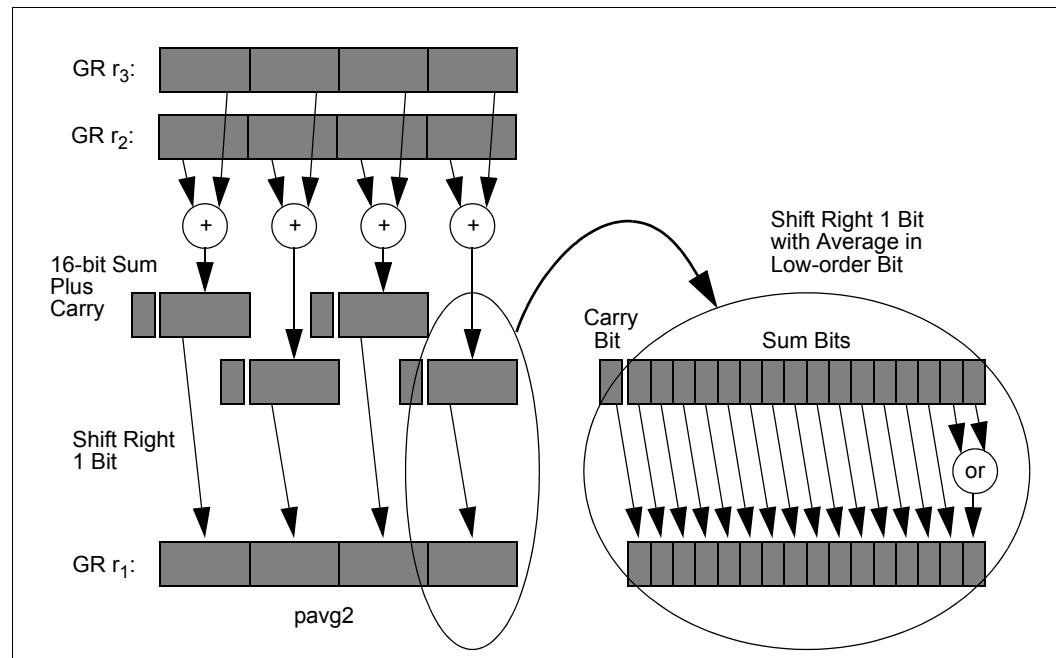
## pavg – Parallel Average

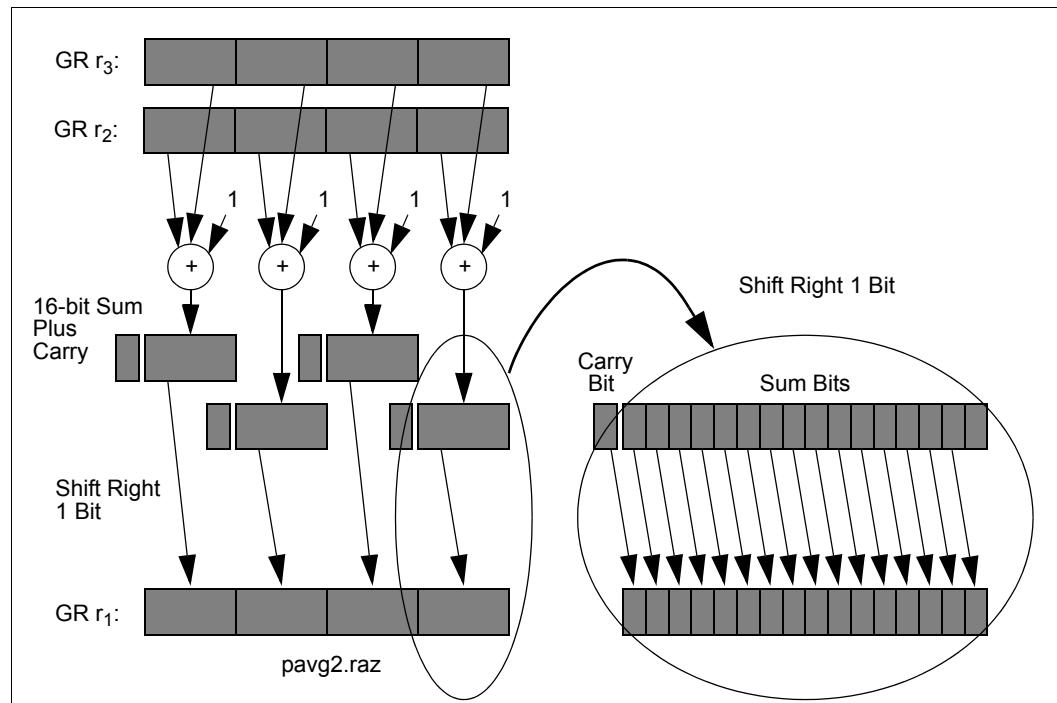
<b>Format:</b>	(qp) pavg1 $r_1 = r_2, r_3$	normal_form, one_byte_form	A9
	(qp) pavg1.raz $r_1 = r_2, r_3$	raz_form, one_byte_form	A9
	(qp) pavg2 $r_1 = r_2, r_3$	normal_form, two_byte_form	A9
	(qp) pavg2.raz $r_1 = r_2, r_3$	raz_form, two_byte_form	A9

**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 2-30. Parallel Average Example**



**Figure 2-31. 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;
}

```

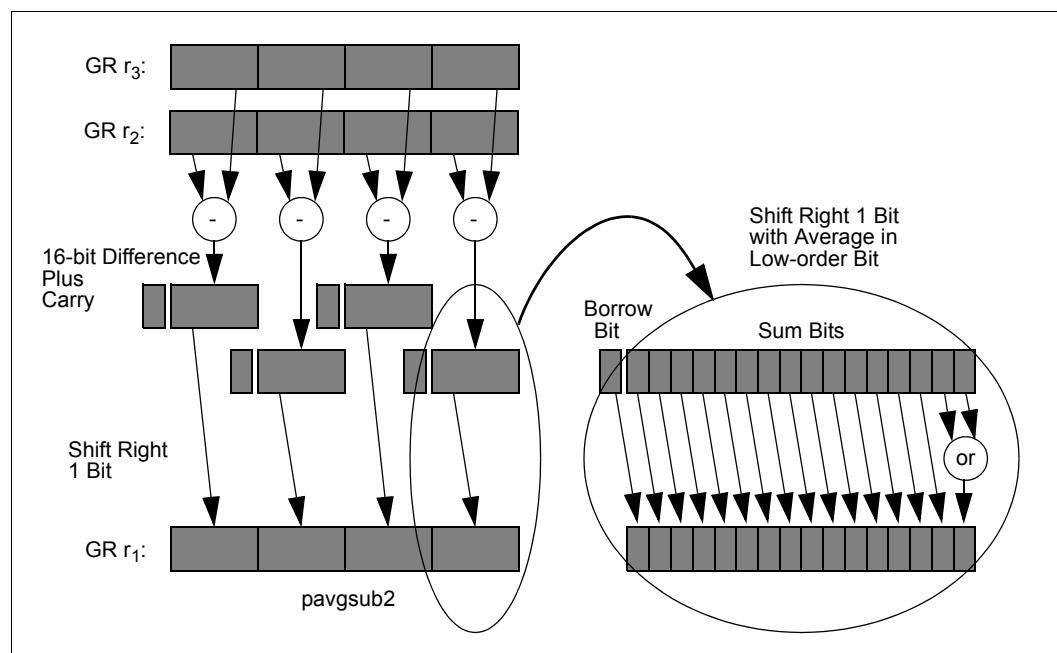
**Interruptions:** Illegal Operation fault

## pavgsb — Parallel Average Subtract

**Format:** (qp) pavgsb1  $r_1 = r_2, r_3$  one\_byte\_form A9  
 (qp) pavgsb2  $r_1 = r_2, r_3$  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 2-32. 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

## pcmp – Parallel Compare

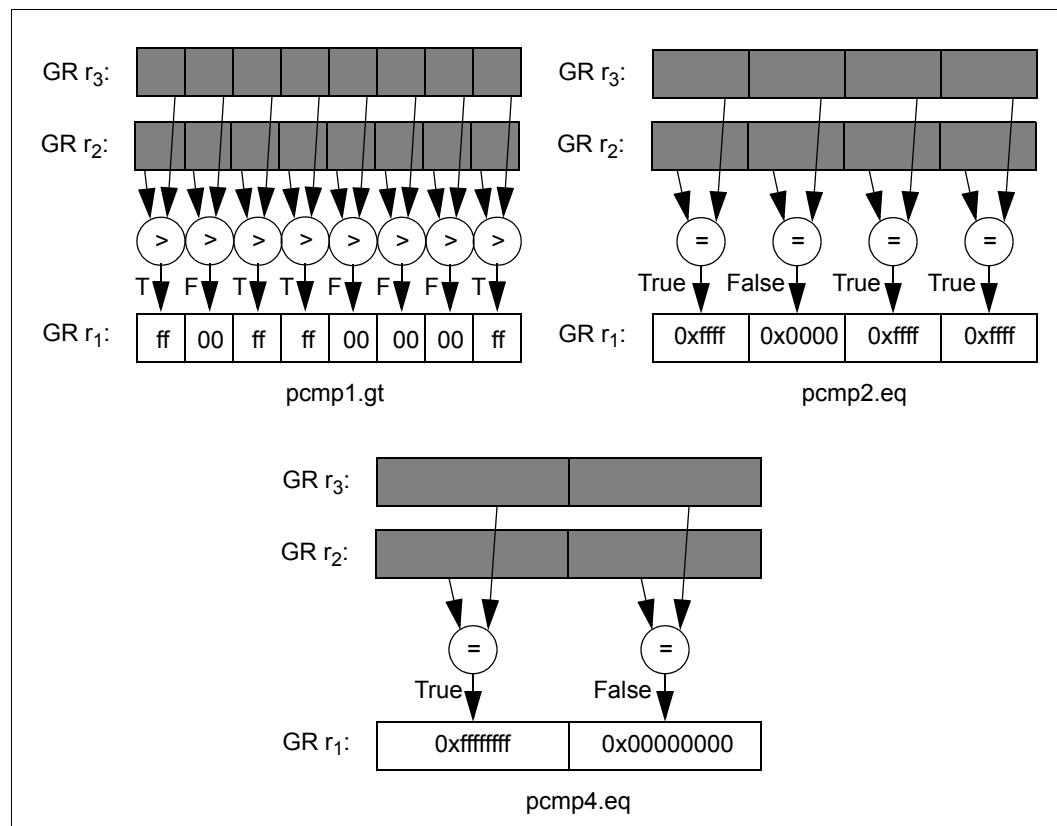
<b>Format:</b>	(qp) pcmp1. <i>prel</i> $r_1 = r_2, r_3$	one_byte_form	A9
	(qp) pcmp2. <i>prel</i> $r_1 = r_2, r_3$	two_byte_form	A9
	(qp) pcmp4. <i>prel</i> $r_1 = r_2, r_3$	four_byte_form	A9

**Description:** The two source operands are compared for one of the two relations shown in **Table 2-45**. 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 2-45. Pcmp Relations**

<i>prel</i>	Compare Relation ( $r_2 \text{ prel } r_3$ )
eq	$r_2 == r_3$
gt	$r_2 > r_3$ (signed)

**Figure 2-33. Parallel Compare 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};
        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;
}
```

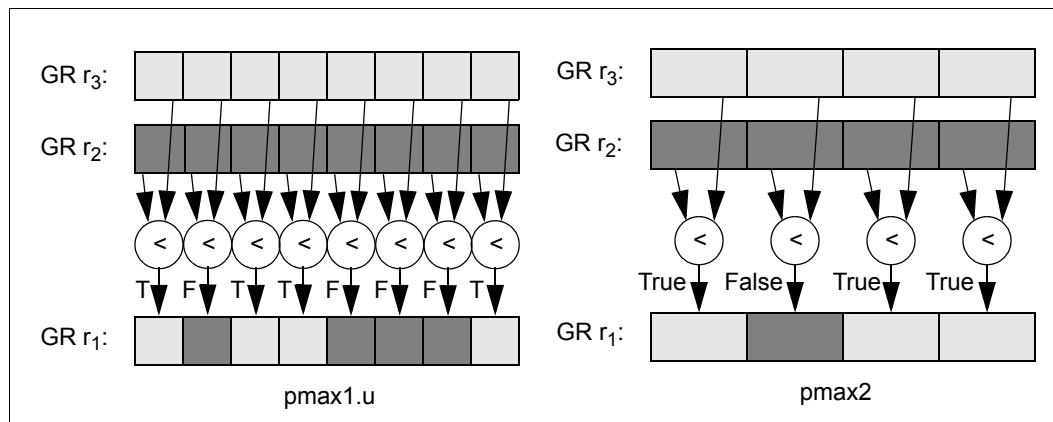
**Interruptions:** Illegal Operation fault

## pmax — 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 2-34. Parallel Maximum 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};
        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;
}

```

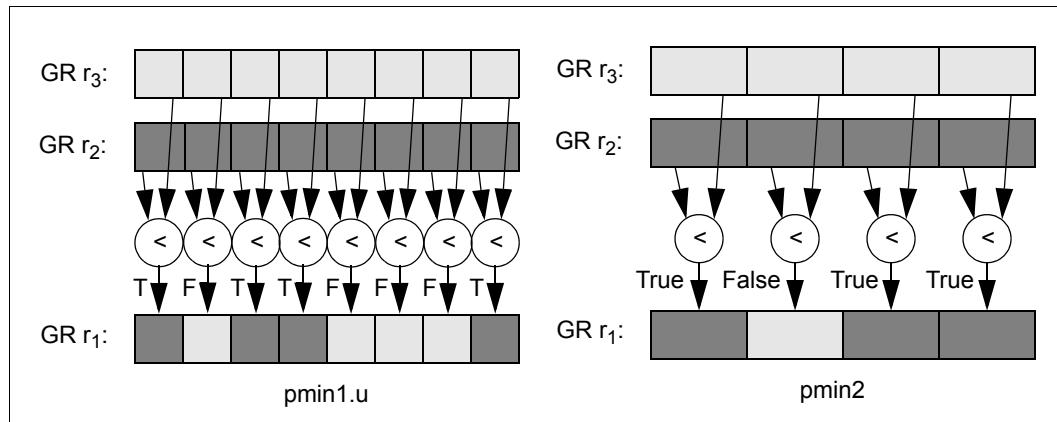
**Interruptions:** Illegal Operation fault

## pmin – Parallel Minimum

**Format:**  $(qp) \text{ pmin1.u } r_1 = r_2, r_3$       one\_byte\_form      I2  
 $(qp) \text{ pmin2 } r_1 = r_2, r_3$       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 2-35. Parallel Minimum 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};
        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

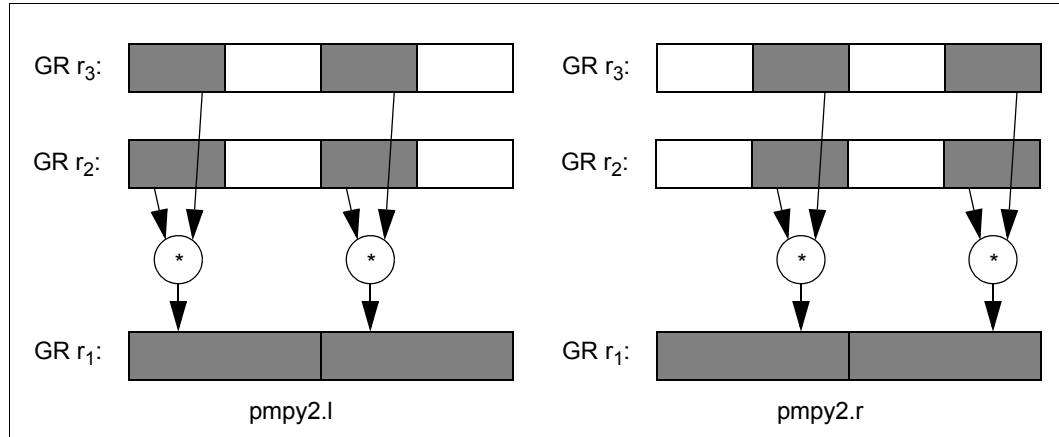
## pmpy — Parallel Multiply

**Format:** (qp) pmpy2.r  $r_1 = r_2, r_3$

right\_form I2  
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 2-36](#). The two 32-bit results are placed in GR  $r_1$ .

**Figure 2-36. Parallel Multiply Operation**



```
Operation: if (PR[qp]) {
    check_target_register( $r_1$ );

    if (right_form) {
        GR[ $r_1$ ]{31:0} = sign_ext(GR[ $r_2$ ]{15:0}, 16) *
                        sign_ext(GR[ $r_3$ ]{15:0}, 16);
        GR[ $r_1$ ]{63:32} = sign_ext(GR[ $r_2$ ]{47:32}, 16) *
                        sign_ext(GR[ $r_3$ ]{47:32}, 16);
    } else { // left_form
        GR[ $r_1$ ]{31:0} = sign_ext(GR[ $r_2$ ]{31:16}, 16) *
                        sign_ext(GR[ $r_3$ ]{31:16}, 16);
        GR[ $r_1$ ]{63:32} = sign_ext(GR[ $r_2$ ]{63:48}, 16) *
                        sign_ext(GR[ $r_3$ ]{63:48}, 16);
    }

    GR[ $r_1$ ].nat = GR[ $r_2$ ].nat || GR[ $r_3$ ].nat;
}
```

**Interruptions:** Illegal Operation fault

## pmpyshr — Parallel Multiply and Shift Right

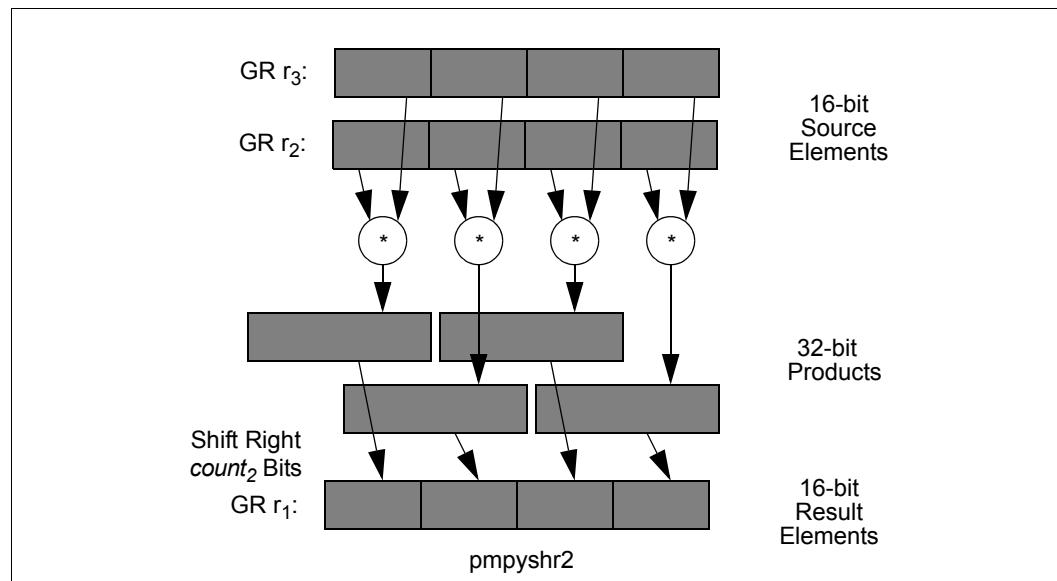
**Format:** (qp) pmpyshr2  $r_1 = r_2, r_3, count_2$       signed\_form I1  
 (qp) pmpyshr2.u  $r_1 = r_2, r_3, count_2$       unsigned\_form I1

**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 2-37](#). 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 2-46](#).

**Table 2-46. Parallel Multiply and Shift Right 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 2-37. 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

## **popcnt — Population Count**

**Format:** (qp) popcnt  $r_1 = r_3$

**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 — Probe Access

<b>Format:</b>	(qp) probe.r $r_1 = r_3, r_2$	regular_form, read_form, register_form	M38
	(qp) probe.w $r_1 = r_3, r_2$	regular_form, write_form, register_form	M38
	(qp) probe.r $r_1 = r_3, imm_2$	regular_form, read_form, immediate_form	M39
	(qp) probe.w $r_1 = r_3, imm_2$	regular_form, 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. In the regular\_form, 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 regular\_form 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 regular\_form probe returns the appropriate value, if not an Alternate Data TLB fault is raised if psr.ic is 1 or a Data Nested TLB fault is raised if psr.ic is 0 or in-flight.

When PSR.dt is 0, a fault\_form probe treats its address operand as a physical address, and takes no TLB related faults.

A regular\_form probe to an unimplemented virtual address returns 0. A fault\_form 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 faults generated by the different forms of the probe instruction are shown in [Table 2-47](#) below:

**Table 2-47. Faults for regular\_form and fault\_form Probe Instructions**

Probe Form Type	Faults
regular_form	Register NaT Consumption fault Virtualization fault <sup>a</sup> 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
fault_form	Register NaT Consumption fault Unimplemented Data Address fault Virtualization fault <sup>a</sup> 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

a. This instruction may optionally raise Virtualization faults, see [Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344](#) for details.

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.

When PSR.vm is 1, this instruction may optionally raise Virtualization faults, see [Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344](#) for details.

Please refer to the [\*\*Intel® Itanium® Software Conventions and Runtime Architecture Guide\*\*](#) for usage information of the `probe` instruction.

```

Operation:    if (PR[qp]) {
                    itype = NON_ACCESS;
                    itype |= (read_write_form) ? READ|WRITE : ((write_form) ? WRITE : READ);
                    itype |= (fault_form) ? PROBE_FAULT : PROBE;
                    itype |= (register_form) ? REGISTER_FORM : IMM_FORM;

                    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, &matr, &defer);
                    } else { // regular_form
                        if (impl_probe_intercept())
                            check_probe_virtualization_fault(itype, tmp_pl);
                        GR[r1] = tlb_grant_permission(GR[r3], itype, tmp_pl);
                        GR[r1].nat = 0;
                    }
                }
}

```

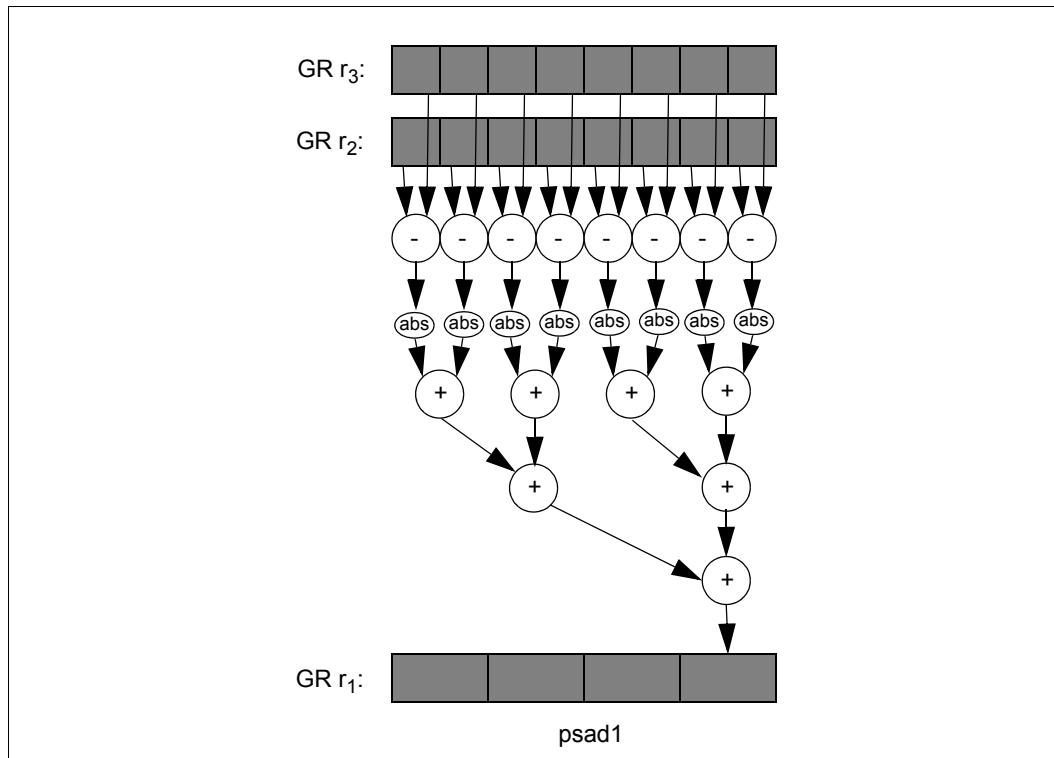
<b>Interruptions:</b>	Illegal Operation fault	Data Page Not Present fault
	Register NaT Consumption fault	Data NaT Page Consumption fault
	Unimplemented Data Address fault	Data Key Miss fault
	Virtualization fault	Data Key Permission fault
	Data Nested TLB fault	Data Access Rights fault
	Alternate Data TLB fault	Data Dirty Bit fault
	VHPT Data fault	Data Access Bit fault
	Data TLB fault	Data Debug fault

## psad — Parallel Sum of Absolute Difference

**Format:** (qp) psad1  $r_1 = r_2, r_3$

**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 2-38. 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;
}

```

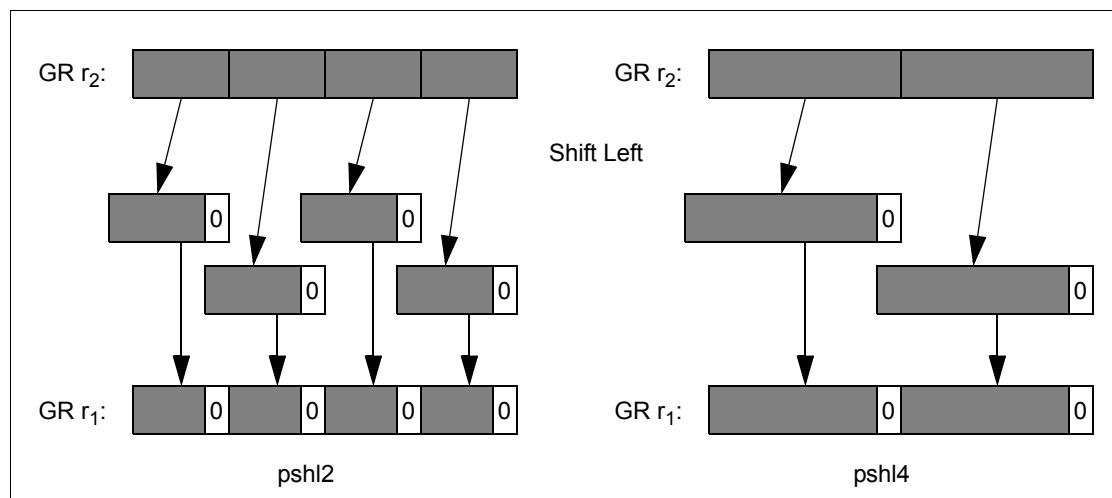
**Interruptions:** Illegal Operation fault

## pshl — Parallel Shift Left

<b>Format:</b>	(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 2-39. Parallel Shift Left Examples**



```

Operation: if (PR[qp]) {
    check_target_register( $r_1$ );

    shift_count = (variable_form ? GR[ $r_3$ ] : count5);
    tmp_nat = (variable_form ? GR[ $r_3$ ].nat : 0);

    if (two_byte_form) { // two_byte_form
        if (shift_count > 16)
            shift_count = 16;
        GR[ $r_1$ ]{15:0} = GR[ $r_2$ ]{15:0} << shift_count;
        GR[ $r_1$ ]{31:16} = GR[ $r_2$ ]{31:16} << shift_count;
        GR[ $r_1$ ]{47:32} = GR[ $r_2$ ]{47:32} << shift_count;
        GR[ $r_1$ ]{63:48} = GR[ $r_2$ ]{63:48} << shift_count;
    } else { // four_byte_form
        if (shift_count > 32)
            shift_count = 32;
        GR[ $r_1$ ]{31:0} = GR[ $r_2$ ]{31:0} << shift_count;
        GR[ $r_1$ ]{63:32} = GR[ $r_2$ ]{63:32} << shift_count;
    }

    GR[ $r_1$ ].nat = GR[ $r_2$ ].nat || tmp_nat;
}

```

**Interruptions:** Illegal Operation fault

## pshladd — 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( $r_1$ );

    x[0] = GR[ $r_2$ ]{15:0};      y[0] = GR[ $r_3$ ]{15:0};
    x[1] = GR[ $r_2$ ]{31:16};    y[1] = GR[ $r_3$ ]{31:16};
    x[2] = GR[ $r_2$ ]{47:32};   y[2] = GR[ $r_3$ ]{47:32};
    x[3] = GR[ $r_2$ ]{63:48};   y[3] = GR[ $r_3$ ]{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) << count_2;

        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[ $r_1$ ] = concatenate4(res[3], res[2], res[1], res[0]);
    GR[ $r_1$ ].nat = GR[ $r_2$ ].nat || GR[ $r_3$ ].nat;
}

```

**Interruptions:** Illegal Operation fault

## pshr — Parallel Shift Right

<b>Format:</b>	(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 > 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

## pshradd — 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

## psub – Parallel Subtract

<b>Format:</b>	(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 2-48](#). 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 2-49](#).

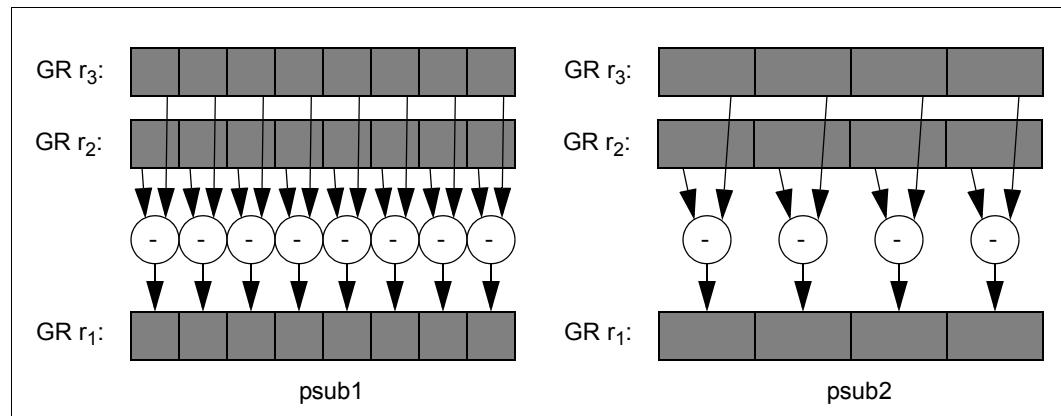
**Table 2-48. Parallel Subtract Saturation Completers**

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 2-49. 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 2-40. Parallel Subtract 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) {           // 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

**ptc.e – Purge Translation Cache Entry**

**Format:** (qp) ptc.e  $r_3$

M47

**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();
    }
}

```

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

```

Operation:    if (PR[ $qp$ ]) {
                    if (PSR.cpl != 0)
                        privileged_operation_fault(0);
                    if (GR[ $r_3$ ].nat)
                        register_nat_consumption_fault(0);
                    if (PSR.vm == 1)
                        virtualization_fault();
                    tlb_purge_translations(GR[ $r_3$ ]);
                }

```

<b>Interruptions:</b>	Privileged Operation fault Register NaT Consumption fault	Virtualization 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.

## ptc.g, ptc.ga — 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  $1 < \text{GR}[r_2]\{7:2\}$  bytes in size. See [Section 4.1.1.7, “Page Sizes” on page 2:57](#) for details on supported page sizes for TLB purges.

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. Serialization is still required to observe the side-effects of a translation being removed. If it is desired that the ptc.g become visible before any subsequent data memory accesses are made visible, a memory fence instruction (mf) should be executed immediately following the ptc.g.

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 coherence domain will then take whatever additional action is necessary to make their ALAT consistent. Matching entries in the local ALAT are optionally invalidated; software must perform a local ALAT invalidation via the invala instruction on the processor issuing the ptc.ga to ensure the local ALAT is coherent.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

Unless specifically supported by the processors and platform, 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. Implementations may optionally support multiple concurrent global purge transactions. The firmware returns if implementations support this optional behavior. It also returns the maximum number of simultaneous outstanding purges allowed.

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], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    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);
}

```

<b>Interruptions:</b>	Machine Check abort Privileged Operation fault Register NaT Consumption fault	Unimplemented Data Address fault Virtualization 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.

## ptc.I – Purge Local Translation Cache

**Format:** (qp) ptc.I  $r_3, r_2$

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  $1 << \text{GR}[r_2]\{7:2\}$  bytes in size. See [Section 4.1.1.7, “Page Sizes” on page 2:57](#) for details on supported page sizes for TLB purges.

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, and when PSR.vm is 0.

This is a local operation, no purge broadcast to other processors occurs in a multiprocessor system. This instruction ensures that all prior stores are made locally visible before the actual purge operation is performed.

**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], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    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  
Virtualization 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.

## ptr — Purge Translation Register

<b>Format:</b>	(qp) ptr.d $r_3, r_2$	data_form instruction_form	M45 M45
	(qp) ptr.i $r_3, r_2$		

**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  $1 < \text{GR}[r_2]\{7:2\}$  bytes in size. See [Section 4.1.1.7, “Page Sizes” on page 2:57](#) for details on supported page sizes for TLB purges.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

This is a local operation, no purge broadcast to other processors occurs in a multiprocessor system.

As described in [Section 4.1.1.2, “Translation Cache \(TC\)” on page 2:49](#), 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], PSR.vm))
                        unimplemented_data_address_fault(0);
                    if (PSR.vm == 1)
                        virtualization_fault();

                    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                              Unimplemented Data Address fault  
                   Register NaT Consumption fault                              Virtualization 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.

## rfi — 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, operation is undefined. 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, and when PSR.vm is 0. 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`.

Execution of this instruction is undefined if IPSR.ic is 0 and the current register stack frame is incomplete.

This instruction does not take Lower Privilege Transfer, Taken Branch or Single Step traps.

If this instruction sets PSR.ri to 2 and the target is an MLX bundle, 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.)

When entering the IA-32 instruction set, the size of the current stack frame is set to zero, and all stacked general registers are left in an undefined state. Software can not rely on the value of these registers across an instruction set transition. Software must ensure that `BSPSTORE==BSP` on entry to the IA-32 instruction set, otherwise undefined behavior may result.

If IPSR.is is 1, software must set other IPSR fields properly for IA-32 instruction set execution; otherwise processor operation is undefined. See [Table 3-2, "Processor Status Register Fields" on page 2:24](#) for details.

Software must issue a `mf` instruction before this instruction if memory ordering is required between IA-32 processor-consistent and Itanium unordered memory references. The processor does not ensure Itanium-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())
                  undefined_behavior();

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

                  if (PSR.vm == 1)
                      virtualization_fault();

                  taken_rfi = 1;

                  PSR = CR[IPSR];
                  if (CR[IPSR].is == 1) {           //resume IA-32 instruction set
                      if (CR[IPSR].ic == 0 || CR[IPSR].dt == 0 ||
                          CR[IPSR].mc == 1 || CR[IPSR].it == 0)
                          undefined_behavior();
                      tmp_IP = CR[IIP];
                      if (!impl_uia_fault_supported() &&
                          ((CR[IPSR].it && unimplemented_virtual_address(tmp_IP, IPSR.vm))
                           || (!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 Itanium instruction set
    tmp_IP = CR[IIP] & ~0xf;
    slot = CR[IPSR].ri;
    if ((CR[IPSR].it && unimplemented_virtual_address(tmp_IP, IPSR.vm))
        || (!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:** Privileged Operation fault  
Virtualization fault

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.

## rsm — Reset System Mask

**Format:** (qp) rsm *imm<sub>24</sub>*

M44

**Description:** The complement of the *imm<sub>24</sub>* operand is ANDed with the system mask (PSR{23:0}) and the result is placed in the system mask. See [Section 3.3.2, "Processor Status Register \(PSR\)" on page 2:23](#).

The PSR system mask can only be written at the most privileged level, and when PSR.vm is 0.

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 (PSR.vm == 1)
        virtualization_fault();

    if (imm24{1})      PSR{1} = 0;          // be
    if (imm24{2})      PSR{2} = 0;          // up
    if (imm24{3})      PSR{3} = 0;          // ac
    if (imm24{4})      PSR{4} = 0;          // mfl
    if (imm24{5})      PSR{5} = 0;          // mfh
    if (imm24{13})     PSR{13} = 0;         // ic
    if (imm24{14})     PSR{14} = 0;         // i
    if (imm24{15})     PSR{15} = 0;         // pk
    if (imm24{17})     PSR{17} = 0;         // dt
    if (imm24{18})     PSR{18} = 0;         // df1
    if (imm24{19})     PSR{19} = 0;         // dfh
    if (imm24{20})     PSR{20} = 0;         // sp
}

```

```

    if (imm24{21})      PSR{21} = 0;)      // pp
    if (imm24{22})      PSR{22} = 0;)      // di
    if (imm24{23})      PSR{23} = 0;)      // si
}

```

- |                       |   |                      |
|-----------------------|---|----------------------|
| <b>Interruptions:</b> | Privileged Operation fault<br>Reserved Register/Field fault   | Virtualization fault |
| <b>Serialization:</b> | 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. |                      |

## rum — Reset User Mask

**Format:** (qp) rum *imm*<sub>24</sub>

M44

**Description:** The complement of the *imm*<sub>24</sub> operand is ANDed with the user mask (PSR{5:0}) and the result is placed in the user mask. See [Section 3.3.2, “Processor Status Register \(PSR\)” on page 2:23](#).

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;          // be
    if (imm24{2} && PSR.sp == 0)           // non-secure perf monitor
        PSR{2} = 0;                      // up
    if (imm24{3})      PSR{3} = 0;          // ac
    if (imm24{4})      PSR{4} = 0;          // mfl
    if (imm24{5})      PSR{5} = 0;          // mfh
}

```

**Interruptions:** Reserved Register/Field fault

**Serialization:** All user mask modifications are observed by the next instruction group.

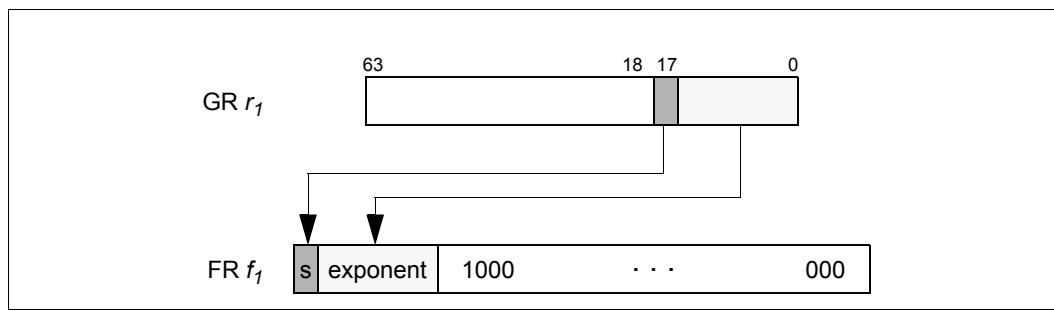
## setf — Set Floating-point Value, Exponent, or Significand

<b>Format:</b>	(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  $f_1$ , as shown in [Figure 5-4](#) and [Figure 5-5 on page 1:93](#), respectively.

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).

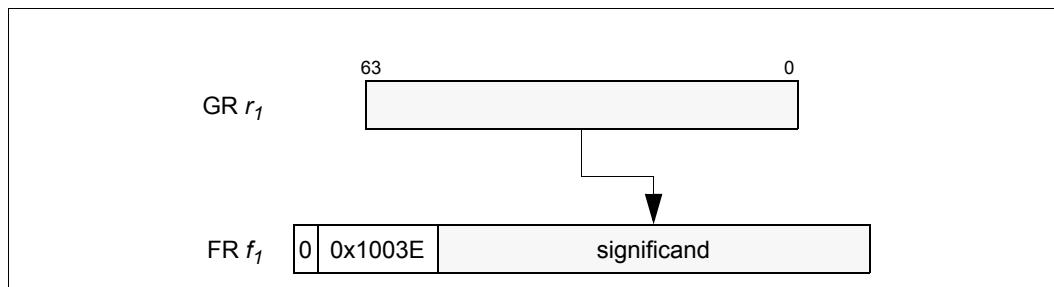
**Figure 2-41. Function of setf.exp**



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 2-42. 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

## shl — Shift Left

**Format:**       $(qp) \text{ shl } r_1 = r_2, r_3$   
 $(qp) \text{ shl } r_1 = r_2, count_6$       pseudo-op of:  $(qp) \text{ dep.z } r_1 = r_2, count_6, 64-count_6$

**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 “[dep — Deposit](#)” on page [3:51](#) 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

## shladd – 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;
}

```

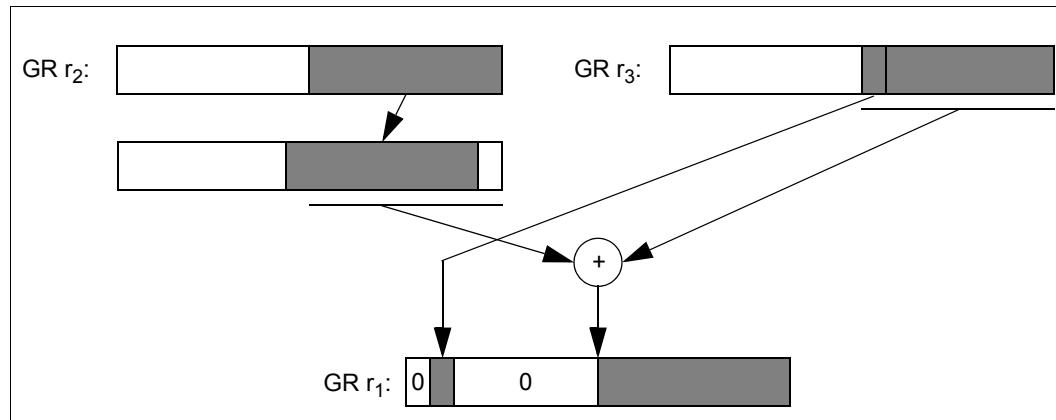
**Interruptions:** Illegal Operation fault

## shladdp4 — Shift Left and Add Pointer

**Format:** (qp) shladdp4  $r_1 = r_2, count_2, r_3$

**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 2-43. Shift Left and Add Pointer**



```
Operation: if (PR[qp]) {
    check_target_register( $r_1$ );

    tmp_res = (GR[ $r_2$ ] <<  $count_2$ ) + GR[ $r_3$ ];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[ $r_3$ ]{31:30};
    GR[ $r_1$ ] = tmp_res;
    GR[ $r_1$ ].nat = GR[ $r_2$ ].nat || GR[ $r_3$ ].nat;
}
```

**Interruptions:** Illegal Operation fault

## shr — Shift Right

<b>Format:</b>	(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 “[extr — Extract](#)” on page 3:54 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

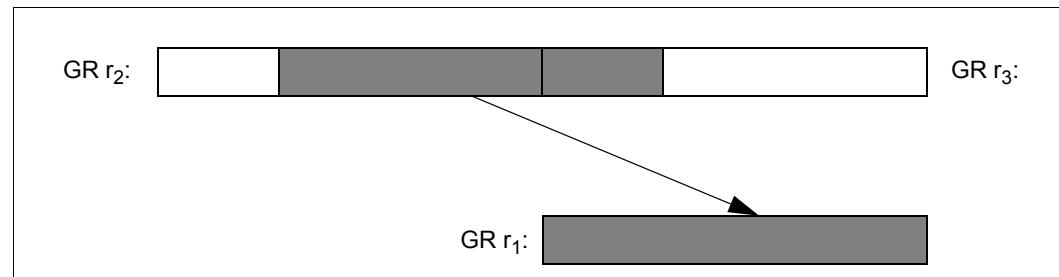
## shrp — Shift Right Pair

**Format:** (qp) shrp  $r_1 = r_2, r_3, count_6$

**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 2-44. Shift Right Pair**



```
Operation: if (PR[qp]) {
    check_target_register( $r_1$ );
    temp1 = shift_right_unsigned(GR[ $r_3$ ],  $count_6$ );
    temp2 = GR[ $r_2$ ] << (64 -  $count_6$ );
    GR[ $r_1$ ] = zero_ext(temp1, 64 -  $count_6$ ) | temp2;
    GR[ $r_1$ ].nat = GR[ $r_2$ ].nat || GR[ $r_3$ ].nat;
}
```

**Interruptions:** Illegal Operation fault

## srlz — Serialize

<b>Format:</b>	(qp) srlz.i (qp) srlz.d	instruction_form data_form	M24 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 [Section 3.2, “Serialization” on page 2:17](#) for details).

**Operation:**

```
if (PR[qp]) {
    if (instruction_form)
        instruction_serialize();
    else // data_form
        data_serialize();
}
```

**Interruptions:** None

## **ssm – Set System Mask**

**Format:** (qp) ssm *imm*<sub>24</sub>

M44

**Description:** The  $imm_{24}$  operand is ORed with the system mask (PSR{23:0}) and the result is placed in the system mask. See [Section 3.3.2, “Processor Status Register \(PSR\)”](#) on page [2:23](#).

The PSR system mask can only be written at the most privileged level, and when PSR.vm is 0.

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 (PSR.vm == 1)
                        virtualization_fault();

                    if (imm24{1})      PSR{1} = 1;          // be
                    if (imm24{2})      PSR{2} = 1;          // up
                    if (imm24{3})      PSR{3} = 1;          // ac
                    if (imm24{4})      PSR{4} = 1;          // mfl
                    if (imm24{5})      PSR{5} = 1;          // mfh
                    if (imm24{13})     PSR{13} = 1;         // ic
                    if (imm24{14})     PSR{14} = 1;         // i
                    if (imm24{15})     PSR{15} = 1;         // pk
                    if (imm24{17})     PSR{17} = 1;         // dt
                    if (imm24{18})     PSR{18} = 1;         // df1
                    if (imm24{19})     PSR{19} = 1;         // dfh
                    if (imm24{20})     PSR{20} = 1;         // sp
                    if (imm24{21})     PSR{21} = 1;         // pp
                    if (imm24{22})     PSR{22} = 1;         // di
                    if (imm24{23})     PSR{23} = 1;         // si
                }

```

**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 [Section 3.2, “Serialization” on page 2:17](#) for a description of serialization.

## st — Store

<b>Format:</b>	(qp) <i>sts</i> . <i>stype</i> . <i>sthint</i> [r <sub>3</sub> ] = r <sub>2</sub>	normal_form, no_base_update_form	M6
	(qp) <i>sts</i> . <i>stype</i> . <i>sthint</i> [r <sub>3</sub> ] = r <sub>2</sub> , imm <sub>9</sub>	normal_form, imm_base_update_form	M5
	(qp) st16. <i>stype</i> . <i>sthint</i> [r <sub>3</sub> ] = r <sub>2</sub> , ar.csd	sixteen_byte_form, no_base_update_form	M6
	(qp) st8.spill. <i>sthint</i> [r <sub>3</sub> ] = r <sub>2</sub>	spill_form, no_base_update_form	M6
	(qp) st8.spill. <i>sthint</i> [r <sub>3</sub> ] = r <sub>2</sub> , imm <sub>9</sub>	spill_form, imm_base_update_form	M5

**Description:** A value consisting of the least significant *sz* bytes of the value in GR r<sub>2</sub> is written to memory starting at the address specified by the value in GR r<sub>3</sub>. The values of the *sz* completer are given in [Table 2-32 on page 3:151](#). The *stype* completer specifies special store operations, which are described in [Table 2-50](#). If the NaT bit corresponding to GR r<sub>3</sub> is 1, or in sixteen\_byte\_form or normal\_form, if the NaT bit corresponding to GR r<sub>2</sub> is 1, a Register NaT Consumption fault is taken.

In the sixteen\_byte\_form, two 8-byte values are stored as a single, 16-byte atomic memory write. The value in GR r<sub>2</sub> is written to memory starting at the address specified by the value in GR r<sub>3</sub>. The value in the Compare and Store Data application register (AR[CSD]) is written to memory starting at the address specified by the value in GR r<sub>3</sub> plus 8.

In the spill\_form, an 8-byte value is stored, and the NaT bit corresponding to GR r<sub>2</sub> is copied to a bit in the UNAT application register. This instruction is used for spilling a register/NaT pair. See [Section 4.4.4, “Control Speculation” on page 1:60](#) for details.

In the imm\_base\_update form, the value in GR r<sub>3</sub> is added to a signed immediate value (imm<sub>9</sub>) and the result is placed back in GR r<sub>3</sub>. 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<sub>2</sub> and r<sub>3</sub> specify the same register). Base register update is not supported for the st16 instruction.

**Table 2-50. Store Types**

<i>stype</i> 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 [Section 4.4.7, “Memory Access Ordering” on page 1:73](#).

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 2-51](#). A prefetch hint is implied in the base update forms. The address specified by the value in GR r<sub>3</sub> after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *sthint*. See [Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69](#).

Hardware support for st16 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 st16 accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.

For the sixteen\_byte\_form, Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See [Section 3.1.11, "Processor Identification Registers" on page 1:34](#) for details.

**Table 2-51. Store Hints**

<i>sthint Completer</i>	<i>Interpretation</i>
none	Temporal locality, level 1
nta	Non-temporal locality, all levels

**Operation:**

```

if (PR[qp]) {
    size = spill_form ? 8 : (sixteen_byte_form ? 16 : sz);
    itype = WRITE;
    if (size == 16) itype |= UNCACHE_OPT;
    otype = (stype == 'rel') ? RELEASE : UNORDERED;

    if (sixteen_byte_form && !instruction_implemented(ST16))
        illegal_operation_fault();
    if (imm_base_update_form)
        check_target_register(r3);
    if (GR[r3].nat || ((sixteen_byte_form || normal_form) && GR[r2].nat))
        register_nat_consumption_fault(WRITE);

    paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                          &tmp_unused);
    if (spill_form && GR[r2].nat) {
        natd_gr_write(GR[r2], paddr, size, UM.be, mattr, otype, sthint);
    }
    else {
        if (sixteen_byte_form)
            mem_write16(GR[r2], AR[CSD], paddr, 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);
    }
}

```

<b>Interruptions:</b>	Illegal Operation fault Register NaT Consumption fault Unimplemented Data Address fault Data Nested TLB fault Alternate Data TLB fault VHPT Data fault	Data Key Miss fault Data Key Permission fault Data Access Rights fault Data Dirty Bit fault Data Access Bit fault Data Debug fault
-----------------------	---	---

Data TLB fault  
Data Page Not Present fault  
Data NaT Page Consumption fault

Unaligned Data Reference fault  
Unsupported Data Reference fault

## stf — Floating-point Store

<b>Format:</b>	(qp) stffsz.sthint [r <sub>3</sub> ] = f <sub>2</sub>	normal_form, no_base_update_form	M13
	(qp) stfsz.sthint [r <sub>3</sub> ] = f <sub>2</sub> , imm <sub>9</sub>	normal_form, imm_base_update_form	M10
	(qp) stf8.sthint [r <sub>3</sub> ] = f <sub>2</sub>	integer_form, no_base_update_form	M13
	(qp) stf8.sthint [r <sub>3</sub> ] = f <sub>2</sub> , imm <sub>9</sub>	integer_form, imm_base_update_form	M10
	(qp) stf.spill.sthint [r <sub>3</sub> ] = f <sub>2</sub>	spill_form, no_base_update_form	M13
	(qp) stf.spill.sthint [r <sub>3</sub> ] = f <sub>2</sub> , imm <sub>9</sub>	spill_form, imm_base_update_form	M10

**Description:** A value, consisting of *fsz* bytes, is generated from the value in FR *f*<sub>2</sub> and written to memory starting at the address specified by the value in GR *r*<sub>3</sub>. In the *normal\_form*, the value in FR *f*<sub>2</sub> is converted to the memory format and then stored. In the *integer\_form*, the significand of FR *f*<sub>2</sub> is stored. The values of the *fsz* completer are given in [Table 2-35 on page 3:157](#). In the *normal\_form* or the *integer\_form*, if the NaT bit corresponding to GR *r*<sub>3</sub> is 1 or if FR *f*<sub>2</sub> contains NaTVal, a Register NaT Consumption fault is taken. See [Section 5.1, “Data Types and Formats” on page 1:85](#) for details on conversion from floating-point register format.

In the *spill\_form*, a 16-byte value from FR *f*<sub>2</sub> is stored without conversion. This instruction is used for spilling a register. See [Section 4.4.4, “Control Speculation” on page 1:60](#) for details.

In the *imm\_base\_update* form, the value in GR *r*<sub>3</sub> is added to a signed immediate value (*imm*<sub>9</sub>) and the result is placed back in GR *r*<sub>3</sub>. 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 2-51 on page 3:252](#). A prefetch hint is implied in the base update forms. The address specified by the value in GR *r*<sub>3</sub> after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *sthint*. See [Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69](#).

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.

<b>Operation:</b>	<pre> 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 &amp;&amp; (FR[f2] == NATVAL)))         register_nat_consumption_fault(WRITE);      size = spill_form ? 16 : (integer_form ? 8 : fsz);     itype = WRITE;     if (size == 10) itype  = UNCACHE_OPT;      paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &amp;mattr, &amp;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);     } } </pre>																			
<b>Interruptions:</b>	<table border="0"> <tr> <td>Illegal Operation fault</td><td>Data NaT Page Consumption fault</td></tr> <tr> <td>Disabled Floating-point Register fault</td><td>Data Key Miss fault</td></tr> <tr> <td>Register NaT Consumption fault</td><td>Data Key Permission fault</td></tr> <tr> <td>Unimplemented Data Address fault</td><td>Data Access Rights fault</td></tr> <tr> <td>Data Nested TLB fault</td><td>Data Dirty Bit fault</td></tr> <tr> <td>Alternate Data TLB fault</td><td>Data Access Bit fault</td></tr> <tr> <td>VHPT Data fault</td><td>Data Debug fault</td></tr> <tr> <td>Data TLB fault</td><td>Unaligned Data Reference fault</td></tr> <tr> <td>Data Page Not Present fault</td><td>Unsupported Data Reference fault</td></tr> </table>	Illegal Operation fault	Data NaT Page Consumption fault	Disabled Floating-point Register fault	Data Key Miss fault	Register NaT Consumption fault	Data Key Permission fault	Unimplemented Data Address fault	Data Access Rights fault	Data Nested TLB fault	Data Dirty Bit fault	Alternate Data TLB fault	Data Access Bit fault	VHPT Data fault	Data Debug fault	Data TLB fault	Unaligned Data Reference fault	Data Page Not Present fault	Unsupported Data Reference fault	
Illegal Operation fault	Data NaT Page Consumption fault																			
Disabled Floating-point Register fault	Data Key Miss fault																			
Register NaT Consumption fault	Data Key Permission fault																			
Unimplemented Data Address fault	Data Access Rights fault																			
Data Nested TLB fault	Data Dirty Bit fault																			
Alternate Data TLB fault	Data Access Bit fault																			
VHPT Data fault	Data Debug fault																			
Data TLB fault	Unaligned Data Reference fault																			
Data Page Not Present fault	Unsupported Data Reference fault																			

**sub — Subtract**

<b>Format:</b>	(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

## sum – Set User Mask

**Format:** (qp) sum *imm*<sub>24</sub>

M44

**Description:** The *imm*<sub>24</sub> operand is ORed with the user mask (PSR{5:0}) and the result is placed in the user mask. See [Section 3.3.2, "Processor Status Register \(PSR\)" on page 2:23](#).

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;          // be
    if (imm24{2} && PSR.sp == 0)          //non-secure perf monitor
        PSR{2} = 1;                      // up
    if (imm24{3})      PSR{3} = 1;          // ac
    if (imm24{4})      PSR{4} = 1;          // mfl
    if (imm24{5})      PSR{5} = 1;          // mfh
}

```

**Interruptions:** Reserved Register/Field fault

**Serialization:** All user mask modifications are observed by the next instruction group.

## sxt — Sign Extend

**Format:** (qp) sxtxsz  $r_1 = r_3$

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 2-52](#).

**Table 2-52. 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

## sync — Memory Synchronization

**Format:** (qp) sync.i

M24

**Description:** sync.i ensures that when previously initiated Flush Cache (fc, fc.i) 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, fc.i) 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 and fc.i) 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.i L1          //flush stale datum from instruction/data cache
;;
sync.i           //require instruction boundary between fc.i and 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

## tak — 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 bits 31:8 of 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, only the DTLB is searched, 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, and when PSR.vm is 0.

```
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);

    if (PSR.vm == 1)
        virtualization_fault();

    GR[r1] = tlb_access_key(GR[r3], itype);
    GR[r1].nat = 0;
}
```

**Interruptions:** Illegal Operation fault  
Privileged Operation fault

Register NaT Consumption fault  
Virtualization fault

## tbit — Test Bit

**Format:** (qp) tbit.*treI 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 *treI* 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 2-15 on page 3:39](#).

The *treI* 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 2-53. Test Bit Relations for Normal and unc tbits**

<i>treI</i>	Test Relation	Pseudo-op of	
nz	selected bit == 1		
z	selected bit == 0	z	$p_1 \leftrightarrow p_2$

**Table 2-54. Test Bit Relations for Parallel tbits**

<i>treI</i>	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')
        tmp_rel = GR[r3]{pos6};                                // 'nz' - test for 1
    else
        tmp_rel = !GR[r3]{pos6};                            // 'z' - test for 0

    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

## tf — Test Feature

**Format:**  $(qp) \text{ tf.trel.ctype } p_1, p_2 = imm_5$

I30

**Description:** The  $imm_5$  value (in the range of 32-63) selects the feature bit defined in Table 2-57 to be tested from the features vector in CPUID[4]. See [Section 3.1.11, “Processor Identification Registers” on page 1:34](#) for details on CPUID registers. 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 2-15 on page 3:39](#).

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 2-55. Test Feature Relations for Normal and unc tf**

<i>trel</i>	Test Relation	Pseudo-op of
nz	selected feature available	<i>z</i>
z	selected feature unavailable	$p_1 \leftrightarrow p_2$

**Table 2-56. Test Feature Relations for Parallel tf**

<i>trel</i>	Test Relation
nz	selected feature available
z	selected feature unavailable

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 the compare type is unc.

**Table 2-57. Test Feature Features Assignment**

$imm_5$	Feature Symbol	Feature
32	@clz	clz feature
33	@mpy	mpy4, mpysh14 feature
34 - 63	none	Not currently defined

```

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

                    tmp_rel = (psr.vm && pal_vp_env_enabled() && VAC.a_tf) ?
                               vcpuid[4]{imm5} : cpuid[4]{imm5};

                    if (trel == 'z')                                // 'z' - test for 0, not 1
                        tmp_rel = !tmp_rel;

                    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

## t<sub>hash</sub> — Translation Hashed Entry Address

**Format:** (qp) t<sub>hash</sub> r<sub>1</sub> = r<sub>3</sub>

M46

**Description:** A Virtual Hashed Page Table (VHPT) entry address is generated based on the specified virtual address and the result is placed in GR r<sub>1</sub>. The virtual address is specified by GR r<sub>3</sub> and the region register selected by GR r<sub>3</sub> bits {63:61}.

If t<sub>hash</sub> 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 t<sub>hash</sub> is defined by the architected short format hash function. See [Section 4.1.5.3, "Region-based VHPT Short Format" on page 2:63](#).

When the processor is configured to use the long format VHPT (PTA.vf=1), t<sub>hash</sub> 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.

This instruction can only be executed when PSR.vm is 0.

**Operation:**

```

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

    if (PSR.vm == 1)
        virtualization_fault();

    if (GR[r3].nat || unimplemented_virtual_address(GR[r3], PSR.vm)) {
        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

Virtualization fault

## tnat — Test NaT

**Format:** (qp) tnat.*trel*.*ctype* *p*<sub>1</sub>, *p*<sub>2</sub> = *r*<sub>3</sub>

I17

**Description:** The NaT bit from GR *r*<sub>3</sub> 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*<sub>1</sub> and *p*<sub>2</sub>. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and [Table 2-15 on page 3:39](#).

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 2-58. Test NaT Relations for Normal and unc tnats**

<i>trel</i>	Test Relation	Pseudo-op of	
nz	selected bit == 1	z	<i>p</i> <sub>1</sub> ↔ <i>p</i> <sub>2</sub>
z	selected bit == 0		

**Table 2-59. Test NaT Relations for Parallel tnats**

<i>trel</i>	Test Relation
nz	selected bit == 1
z	selected bit == 0

If the two predicate register destinations are the same (*p*<sub>1</sub> and *p*<sub>2</sub> 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 ( $p_1 == p_2$ )
                        illegal_operation_fault();

                    if ( $trel == 'nz'$ )
                        tmp_rel = GR[r3].nat;
                    else
                        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 ( $p_1 == p_2$ )
                            illegal_operation_fault();
                        PR[p1] = 0;
                        PR[p2] = 0;
                    }
                }
}

```

**Interruptions:** Illegal Operation fault

## tpa — 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, only the DTLB is searched, because the VHPT walker is disabled. If no matching present translation is found in the DTLB, an Alternate Data TLB fault is raised if psr.ic is one or a Data Nested TLB fault is raised if psr.ic is zero.

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, and when PSR.vm is 0.

```
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;
}
```

<b>Interruptions:</b>	Illegal Operation fault	Alternate Data TLB fault
	Privileged Operation fault	VHPT Data fault
	Register NaT Consumption fault	Data TLB fault
	Unimplemented Data Address fault	Data Page Not Present fault
	Virtualization fault	Data NaT Page Consumption fault
	Data Nested TLB fault	

## ttag — 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.

This instruction can only be executed when PSR.vm is 0.

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

    if (PSR.vm == 1)
        virtualization_fault();

    if (GR[r3].nat || unimplemented_virtual_address(GR[r3], PSR.vm)) {
        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;
    }
}
```

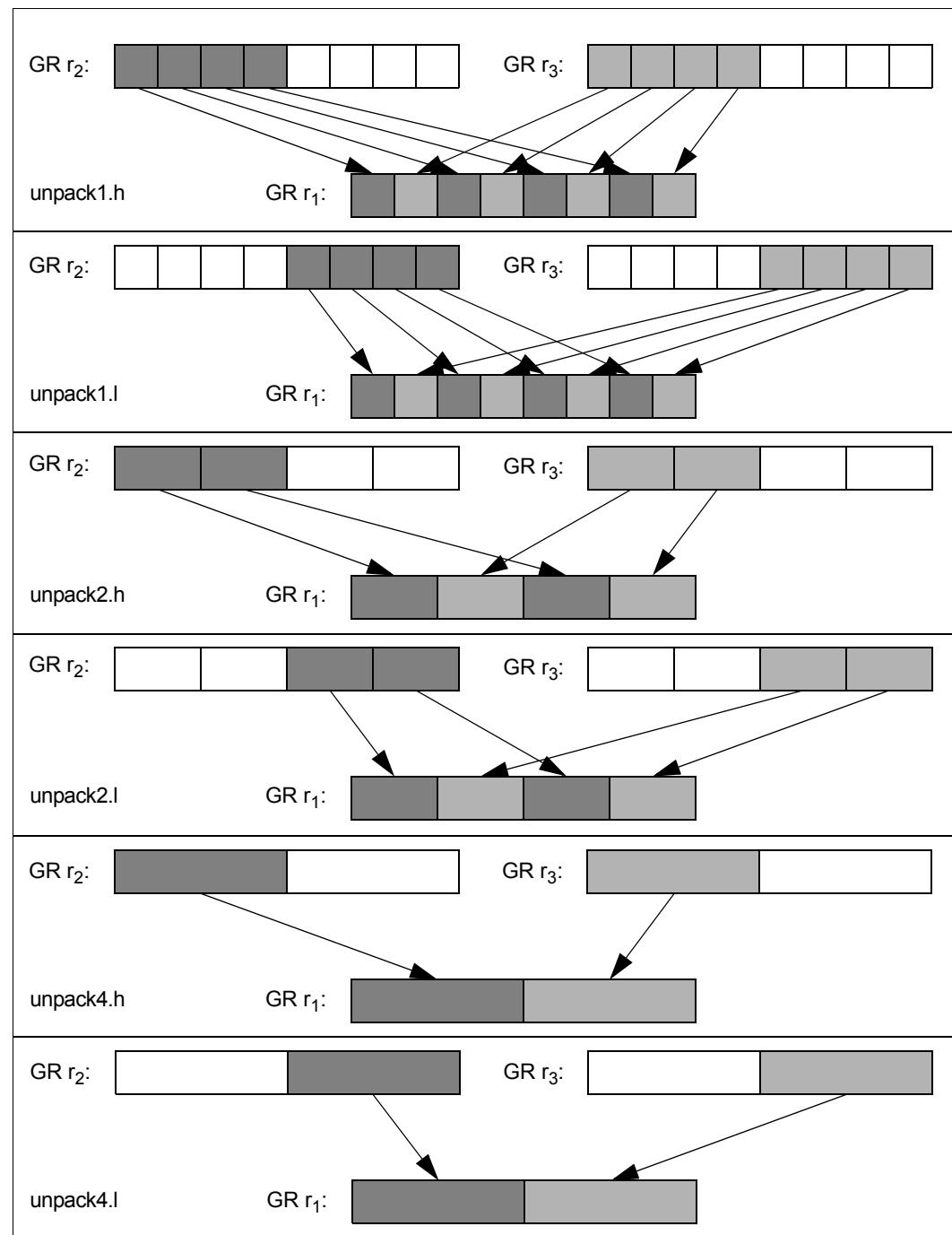
**Interruptions:** Illegal Operation fault

Virtualization fault

**unpack — Unpack**

<b>Format:</b>	(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.

**Figure 2-45. Unpack Operation**

```

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

## vmsw – Virtual Machine Switch

**Format:** vmsw.0 zero\_form B8  
vmsw.1 one\_form B8

**Description:** This instruction sets the PSR.vm bit to the specified value. This instruction can be used to implement transitions to/from virtual machine mode without the overhead of an interruption.

If instruction address translation is enabled and the page containing the `vmsw` instruction has access rights equal to 7, then the new value is written to the `PSR.vm` bit. In the `zero_form`, `PSR.vm` is set to 0, and in the `one_form`, `PSR.vm` is set to 1.

Instructions after the `vmsw` instruction in the same instruction group may be executed with the old or new value of PSR.vm. Instructions in subsequent instruction groups will be executed with PSR.vm equal to the new value.

If the above conditions are not met, this instruction takes a Virtualization fault.

This instruction can only be executed at the most privileged level. This instruction cannot be predicated.

Implementation of PSR.vm is optional. If it is not implemented, this instruction takes Illegal Operation fault. If it is implemented but either virtual machine features or the vmsw instruction are disabled, this instruction takes Virtualization fault when executed at the most privileged level.

```
Operation:    if (!implemented_vm())
                    illegal_operation_fault();

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

                if (!(PSR.it == 1 && itlb_ar() == 7) || vm_disabled() || vmsw_disabled())
                    virtualization_fault();

                if (zero_form) {
                    PSR.vm = 0;
                }
                else {
                    PSR.vm = 1;
                }
```

<b>Interruptions:</b>	Illegal Operation fault Privileged Operation fault	Virtualization fault
-----------------------	---	----------------------

## xchg — Exchange

**Format:** (qp) xchgsz.*lhint*  $r_1 = [r_3], r_2$  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 2-60](#).

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 2-60. Memory Exchange Size**

<b>sz Completer</b>	<b>Bytes Accessed</b>
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 [Section 4.4.7, "Sequentiality Attribute and Ordering" on page 2:82](#) 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 *lhint* completer specifies the locality of the memory access. The values of the *lhint* completer are given in [Table 2-34 on page 3:152](#). Locality hints do not affect program functionality and may be ignored by the implementation. See [Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69](#) for details.

```

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;
                }

```

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

## xma — Fixed-Point Multiply Add

<b>Format:</b>	(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:**  $f_1$  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( $f_1$ );
    if (tmp_isrcode = fp_reg_disabled( $f_1$ ,  $f_2$ ,  $f_3$ ,  $f_4$ ))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[ $f_2$ ]) || fp_is_natval(FR[ $f_3$ ]) ||
        fp_is_natval(FR[ $f_4$ ])) {
        FR[ $f_1$ ] = NATVAL;
    } else {
        if (low_form || high_form)
            tmp_res_128 =
                fp_I64_x_I64_to_I128(FR[ $f_3$ ].significand, FR[ $f_4$ ].significand);
        else // high_unsigned_form
            tmp_res_128 =
                fp_U64_x_U64_to_U128(FR[ $f_3$ ].significand, FR[ $f_4$ ].significand);

        tmp_res_128 =
            fp_U128_add(tmp_res_128, fp_U64_to_U128(FR[ $f_2$ ].significand));

        if (high_form || high_unsigned_form)
            FR[ $f_1$ ].significand = tmp_res_128.hi;
        else // low_form
            FR[ $f_1$ ].significand = tmp_res_128.lo;

        FR[ $f_1$ ].exponent = FP_INTEGER_EXP;
        FR[ $f_1$ ].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr( $f_1$ );
}

```

**Interruptions:** Disabled Floating-point Register fault

## xmpy – Fixed-Point Multiply

<b>Format:</b>	(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.lu $f_1 = f_3, f_4, f0$
	(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 $f_1 = f_3, f_4, f0$
<b>Description:</b>	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 “[xma – Fixed-Point Multiply Add](#)” on page 3:276.

## xor – Exclusive Or

**Format:**  $(qp) \text{ xor } r_1 = r_2, r_3$  register\_form A1  
 $(qp) \text{ xor } r_1 = imm_8, r_3$  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

## **zxt — Zero Extend**

**Format:** (qp) zxt<sub>xsz</sub>  $r_1 = r_3$

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 2-52 on page 3:258](#).

**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

§

This chapter contains a table of all pseudo-code functions used on the Itanium instruction pages.

**Table 3-1. Pseudo-code Functions**

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 <code>rtype</code> and the register address specified by <code>raddr</code> , 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 <code>paddr</code> and the access size specified by <code>size</code> . 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 <code>rtype</code> and the register address specified by <code>rega</code> . At most one matching ALAT entry is invalidated. No value is returned.
alat_read_memory_on_hit(ldtype, rtype, raddr)	Returns a one if the implementation requires that the requested check load should perform a memory access (requires prior address translation); returns a zero otherwise.
alat_translate_address_on_hit(ldtype, rtype, raddr)	Returns a one if the implementation requires that the requested check load should translate the source address and take associated faults; returns a zero otherwise.
alat_write(ldtype, rtype, raddr, paddr, size)	Allocates a new ALAT entry or updates an existing entry using the load type specified by <code>ldtype</code> , the register type specified by <code>rtype</code> , the register address specified by <code>raddr</code> , the physical memory address specified by <code>paddr</code> , and the access size specified by <code>size</code> . No value is returned. This function guarantees that at most only one ALAT entry exists for a given <code>raddr</code> . Based on the load type <code>ldtype</code> , if a <code>ld.c.nc</code> , <code>ldf.c.nc</code> , or <code>ldfp.c.nc</code> instruction's <code>raddr</code> matches an existing ALAT entry's register tag, but the instruction's <code>size</code> and/or <code>paddr</code> 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 <code>size</code> and <code>paddr</code> .
align_to_size_boundary(vaddr, size)	Returns <code>vaddr</code> aligned to the boundary specified by <code>size</code> .
branch_predict(wh, ih, ret, target, tag)	Implementation-dependent routine which updates the processor's branch prediction structures.

**Table 3-1. Pseudo-code Functions (Continued)**

Function	Operation
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. In addition, the result may depend on other implementation-dependent parameters.
check_probe_virtualization_fault(type, cpl)	If implemented, this function may raise virtualization faults for specific probe instructions. Please refer to the instruction page for probe instruction for details.
check_target_register(r1)	If the <code>r1</code> argument specifies an out-of-frame stacked register (as defined by CFM) or <code>r1</code> specifies GR0, an Illegal Operation fault is delivered, and this function does not return.
check_target_register_sof(r1, newsof)	If the <code>r1</code> argument specifies an out-of-frame stacked register (as defined by the <code>newsof</code> argument) or <code>r1</code> specifies GR0, 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.
execute_hint(hint)	Executes the hint specified by <code>hint</code> .
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 <code>fcmp</code> instruction.
fcvt_fx_exception_fault_check(fr2, signed_form, trunc_form, sf *tmp_fp_env)	Checks for all floating-point faulting conditions for the <code>fcvt.fx</code> , <code>fcvt.fxu</code> , <code>fcvt.fx.trunc</code> and <code>fcvt.fxu.trunc</code> 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 <code>fma</code> 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 <code>famax</code> , <code>famin</code> , <code>fmax</code> , and <code>fmin</code> instructions.
fms_fnma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the <code>fms</code> and <code>fnma</code> 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.
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_equal(fr1, fr2)	IEEE standard equality relationship test.
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 <code>size</code> parameter.
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.

**Table 3-1. Pseudo-code Functions (Continued)**

Function	Operation
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_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_mem_to_fr_format(mem, size)	Converts a floating-point value in memory format to floating-point register format.
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_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.
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_fnma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the fpms and fnma 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.
fprsqrt_a_exception_fault_check(f3, sf, *tmp_fp_env, *limits_check)	Checks for all floating-point faulting conditions for the fprsqrt_a 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.
frsqrt_a_exception_fault_check(f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the frsqrt_a 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.

**Table 3-1. Pseudo-code Functions (Continued)**

Function	Operation
impl_check_mov_itir()	Implementation-specific function that returns TRUE if ITIR is checked for reserved fields and encodings on a mov to ITIR instruction.
impl_check_mov_psr_l(gr)	Implementation-specific function to check bits {63:32} of gr corresponding to reserved fields of the PSR for Reserved Register/Field fault.
impl_check_tlb_itir()	Implementation-specific function that returns TRUE if all fields of ITIR are checked for reserved encodings on a TLB insert instruction regardless of whether the translation is present.
impl_gitc_enable()	Implementation-specific function that indicates whether guest MOV-from-AR.ITE optimization is enabled.
impl_ia32_ar_reserved_ignored(ar3)	Implementation-specific function which indicates how the reserved and ignored fields in the specified IA-32 application register, ar3, behave. If it returns FALSE, the reserved and/or ignored bits in the specified application register can be written, and when read they return the value most-recently written. If it returns TRUE, attempts to write a non-zero value to a reserved field in the specified application register cause a Reserved Register/Field fault, and reads return 0; writing to an ignored field in the specified application register is ignored, and reads return the constant value defined for that field.
impl_iib()	Implementation-specific function which indicates whether Interruption Instruction Bundle registers (IIB0-1) are implemented.
impl_itir_cwi_mask()	Implementation-specific function that either returns the value passed to it or the value passed to it masked with zeros in bit positions {63:32} and/or {1:0}.
impl_ito()	Implementation-specific function which indicates whether Interval Timer Offset (ITO) register is implemented.
impl_probe_intercept()	Implementation-specific function indicates whether probe interceptions are supported.
impl_ruc()	Implementation-specific function which indicates whether Resource Utilization Counter (RUC) application register is implemented.
impl_uia_fault_supported()	Implementation-specific function that either returns TRUE if the processor reports unimplemented instruction addresses with an Unimplemented Instruction Address fault, and returns FALSE if the processor reports them with an Unimplemented Instruction Address trap.
implemented_vm()	Returns TRUE if the processor implements the PSR.vm bit (regardless of whether virtual machine features are enabled or disabled).
instruction_implemented(inst)	Implementation-dependent routine which returns TRUE or FALSE, depending on whether inst is implemented.
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 Flush Cache operations are observed by the local data cache they are observed by the local instruction cache, and when prior Flush Cache 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 <a href="#">Section 3.3.5, “Interruption Control Registers” on page 2:36</a> ), 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 (Continued)**

Function	Operation
is_read_only_reg(rtype, raddr)	Returns a one if the register addressed by <code>raddr</code> in the register bank of type <code>rtype</code> 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 <code>regnum</code> is reserved in the <code>regclass</code> register file.
is_supported_hint(hint)	Returns true if the implementation supports the specified <code>hint</code> . This function may depend on factors other than the <code>hint</code> value, such as which execution unit it is executed on or the slot number the instruction was encoded in.
itlb_ar()	Returns the page access rights from the ITLB for the page addressed by the current IP, or <code>INVALID_AR</code> if PSR.it is 0.
make_icache_coherent(paddr)	The cache line addressed by the physical address <code>paddr</code> is flushed in an implementation-specific manner that ensures that the instruction cache is coherent with the data caches.
mem_flush(paddr)	The line addressed by the physical address <code>paddr</code> 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 <code>vaddr</code> to the location of the memory hierarchy specified by <code>hint</code> . This function is implementation dependent and can be ignored. The <code>type</code> allows the implementation to distinguish prefetches for different instruction types.
mem_promote(paddr, mtype, hint)	Moves the line addressed by <code>paddr</code> to the highest level of the memory hierarchy conditioned by the access hints specified by <code>hint</code> . Implementation dependent and can be ignored.
mem_read(paddr, size, border, mattr, otype, hint)	Returns the <code>size</code> bytes starting at the physical memory location specified by <code>paddr</code> with byte order specified by <code>border</code> , memory attributes specified by <code>mattr</code> , and access hint specified by <code>hint</code> . <code>otype</code> specifies the memory ordering attribute of this access, and must be <code>UNORDERED</code> or <code>ACQUIRE</code> .
mem_read_pair(*low_value, *high_value, paddr, size, border, mattr, otype, hint)	Reads the <code>size / 2</code> bytes of memory starting at the physical memory address specified by <code>paddr</code> into <code>low_value</code> , and the <code>size / 2</code> bytes of memory starting at the physical memory address specified by <code>(paddr + size / 2)</code> into <code>high_value</code> , with byte order specified by <code>border</code> , memory attributes specified by <code>mattr</code> , and access hint specified by <code>hint</code> . <code>otype</code> specifies the memory ordering attribute of this access, and must be <code>UNORDERED</code> or <code>ACQUIRE</code> . No value is returned.
mem_write(value, paddr, size, border, mattr, otype, hint)	Writes the least significant <code>size</code> bytes of <code>value</code> into memory starting at the physical memory address specified by <code>paddr</code> with byte order specified by <code>border</code> , memory attributes specified by <code>mattr</code> , and access hint specified by <code>hint</code> . <code>otype</code> specifies the memory ordering attribute of this access, and must be <code>UNORDERED</code> or <code>RELEASE</code> . No value is returned.
mem_write16(gr_value, ar_value, paddr, border, mattr, otype, hint)	Writes the 8 bytes of <code>gr_value</code> into memory starting at the physical memory address specified by <code>paddr</code> , and the 8 bytes of <code>ar_value</code> into memory starting at the physical memory address specified by <code>(paddr + 8)</code> , with byte order specified by <code>border</code> , memory attributes specified by <code>mattr</code> , and access hint specified by <code>hint</code> . <code>otype</code> specifies the memory ordering attribute of this access, and must be <code>UNORDERED</code> or <code>RELEASE</code> . No value is returned.
mem_xchg(data, paddr, size, byte_order, mattr, otype, hint)	Returns <code>size</code> bytes from memory starting at the physical address specified by <code>paddr</code> . The read is conditioned by the locality hint specified by <code>hint</code> . After the read, the least significant <code>size</code> bytes of data are written to <code>size</code> bytes in memory starting at the physical address specified by <code>paddr</code> . The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by <code>mattr</code> and the byte ordering in memory is specified by <code>byte_order</code> . <code>otype</code> specifies the memory ordering attribute of this access, and must be <code>ACQUIRE</code> .

**Table 3-1. Pseudo-code Functions (Continued)**

Function	Operation
mem_xchg_add(add_val, paddr, size, byte_order, mattr, otype, hint)	Returns <code>size</code> bytes from memory starting at the physical address specified by <code>paddr</code> . The read is conditioned by the locality hint specified by <code>hint</code> . The least significant <code>size</code> bytes of the sum of the value read from memory and <code>add_val</code> is then written to <code>size</code> bytes in memory starting at the physical address specified by <code>paddr</code> . The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by <code>mattr</code> and the byte ordering in memory is specified by <code>byte_order</code> . <code>otype</code> 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 <code>size</code> bytes from memory starting at the physical address specified by <code>paddr</code> . The read is conditioned by the locality hint specified by <code>hint</code> . If the value read from memory is equal to <code>cmp_val</code> , then the least significant <code>size</code> bytes of data are written to <code>size</code> bytes in memory starting at the physical address specified by <code>paddr</code> . 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 <code>mattr</code> and the byte ordering in memory is specified by <code>byte_order</code> . <code>otype</code> specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
mem_xchg16_cond(cmp_val, gr_data, ar_data, paddr, byte_order, mattr, otype, hint)	Returns 8 bytes from memory starting at the physical address specified by <code>paddr</code> . The read is conditioned by the locality hint specified by <code>hint</code> . If the value read from memory is equal to <code>cmp_val</code> , then the 8 bytes of <code>gr_data</code> are written to 8 bytes in memory starting at the physical address specified by <code>(paddr &amp; ~0x8)</code> , and the 8 bytes of <code>ar_data</code> are written to 8 bytes in memory starting at the physical address specified by <code>((paddr &amp; ~0x8) + 8)</code> . 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 <code>mattr</code> and the byte ordering in memory is specified by <code>byte_order</code> . The byte ordering only affects the ordering of bytes within each of the 8-byte values stored. <code>otype</code> 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.
partially_implemented_ip()	Implementation-dependent routine which returns TRUE if the implementation, on an Unimplemented Instruction Address trap, writes IIP with the sign-extended virtual address or zero-extended physical address for what would have been the next value of IP. Returns FALSE if the implementation, on this trap, simply writes IIP with the full address which would have been the next value of IP.
pending_virtual_interrupt()	Check for unmasked pending virtual interrupt.
pr_phys_to_virt(phys_id)	Returns the virtual register id of the predicate from the physical register id, <code>phys_id</code> 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 <code>number_of_bytes</code> may be zero. The resulting sequence of RSE loads may be interrupted. Mandatory RSE loads may cause an interruption; see <a href="#">Table 6-6, “RSE Interruption Summary” on page 6-145</a> .
rse_invalidate_non_current_REGS()	All registers outside the current frame are invalidated.

**Table 3-1. Pseudo-code Functions (Continued)**

Function	Operation
rse_load(type)	Restores a register or NaT collection from the backing store ( <code>load_address = RSE.BspLoad - 8</code> ). If <code>load_address{8:3}</code> is equal to <code>0x3f</code> then a NaT collection is loaded into a NaT dispersal register. (dispersal register may not be the same as AR[RNAT].) If <code>load_address{8:3}</code> is not equal to <code>0x3f</code> then the register <code>RSE.LoadReg - 1</code> is loaded and the NaT bit for that register is set to <code>dispersal_register{load_address{8:3}}</code> . If the load is successful <code>RSE.BspLoad</code> is decremented by 8. If the load is successful and a register was loaded <code>RSE.LoadReg</code> is decremented by 1 (possibly wrapping in the stacked registers). The load moves a register from the invalid partition to the current frame if <code>RSE.CFLE</code> is 1, or to the clean partition if <code>RSE.CFLE</code> is 0. For mandatory RSE loads, <code>type</code> is MANDATORY. Mandatory RSE loads may cause interruptions. See <a href="#">Table 6-6, "RSE Interruption Summary" on page 6-145</a> .
rse_new_frame( <code>current_frame_size</code> , <code>new_frame_size</code> )	A new frame is defined without changing any register renaming. The new frame size is completely defined by the <code>new_frame_size</code> parameter (successive calls are not cumulative). If <code>new_frame_size</code> is larger than <code>current_frame_size</code> 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 <a href="#">Table 6-6, "RSE Interruption Summary" on page 6-145</a> .
rse_preserve_frame( <code>preserved_frame_size</code> )	The number of registers specified by <code>preserved_frame_size</code> are marked to be preserved by the RSE. Register renaming causes the <code>preserved_frame_size</code> registers after <code>GR[32]</code> to be renamed to <code>GR[32]</code> . <code>AR[BSP]</code> is updated to contain the backing store address where the new <code>GR[32]</code> will be stored.
rse_restore_frame( <code>preserved_sol</code> , <code>growth</code> , <code>current_frame_size</code> )	The first two parameters define how the current frame is about to be updated by a branch return or rfi: <code>preserved_sol</code> defines how many registers need to be restored below RSE.BOF; <code>growth</code> defines by how many registers the top of the current frame will grow (growth will generally be negative). The number of registers specified by <code>preserved_sol</code> are marked to be restored. Register renaming causes the <code>preserved_sol</code> registers before <code>GR[32]</code> to be renamed to <code>GR[32]</code> . <code>AR[BSP]</code> is updated to contain the backing store address where the new <code>GR[32]</code> will be stored. If the number of dirty and clean registers is less than <code>preserved_sol</code> 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, <code>current_frame_size</code> , is used to force the returned to frame to zero (see <a href="#">Section 6.5.5, "Bad PFS used by Branch Return" on page 2:143</a> ).
rse_store(type)	Saves a register or NaT collection to the backing store ( <code>store_address = AR[BSPSTORE]</code> ). If <code>store_address{8:3}</code> is equal to <code>0x3f</code> then the NaT collection AR[RNAT] is stored. If <code>store_address{8:3}</code> is not equal to <code>0x3f</code> then the register <code>RSE.StoreReg</code> is stored and the NaT bit from that register is deposited in <code>AR[RNAT]{store_address{8:3}}</code> . If the store is successful <code>AR[BSPSTORE]</code> is incremented by 8. If the store is successful and a register was stored <code>RSE.StoreReg</code> 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, <code>type</code> is MANDATORY. Mandatory RSE stores may cause interruptions. See <a href="#">Table 6-6, "RSE Interruption Summary" on page 6-145</a> .
rse_update_internal_stack_pointers( <code>new_store_pointer</code> )	Given a new value for <code>AR[BSPSTORE]</code> ( <code>new_store_pointer</code> ) this function computes the new value for <code>AR[BSP]</code> . This value is equal to <code>new_store_pointer</code> 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 <code>AR[BSPSTORE]</code> . All clean registers are moved to the invalid partition.
sign_ext( <code>value</code> , <code>pos</code> )	Returns a 64 bit number with bits <code>pos-1</code> through 0 taken from <code>value</code> and bit <code>pos-1</code> of <code>value</code> replicated in bit positions <code>pos</code> through 63. If <code>pos</code> is greater than or equal to 64, <code>value</code> is returned.

**Table 3-1. Pseudo-code Functions (Continued)**

Function	Operation
spontaneous_deferral(paddr, size, border, mattr, otype, hint, *defer)	Implementation-dependent routine which optionally forces <code>*defer</code> to TRUE if all of the following are true: spontaneous deferral is enabled, spontaneous deferral is permitted by the programming model, and the processor determines it would be advantageous to defer the speculative load (e.g., based on a miss in some particular level of cache).
spontaneous_deferral_enabled()	Implementation-dependent routine which returns TRUE or FALSE, depending on whether spontaneous deferral of speculative loads is enabled or disabled in the processor.
tlb_access_key(vaddr, itype)	This function returns, in bits 31:8, the access key from the TLB for the entry corresponding to <code>vaddr</code> and <code>itype</code> ; bits 63:32 and 7:0 return 0. If <code>vaddr</code> is an unimplemented virtual address, or a matching present translation is not found, the value 1 is returned.
tlb_broadcast_purge(rid, vaddr, size, type)	Sends a broadcast purge DTC and ITC transaction to other processors in the multiprocessor coherency domain, where the region identifier ( <code>rid</code> ), virtual address ( <code>vaddr</code> ) and page size ( <code>size</code> ) specify the translation entry to purge. The operation waits until all processors that receive the purge have completed the purge operation. The purge type ( <code>type</code> ) 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 ( <code>vaddr</code> ) and privilege level ( <code>pl</code> ). The access type ( <code>type</code> ) specifies either read or write. The following faults are checked: <ul style="list-style-type: none"> <li>• Data Nested TLB fault</li> <li>• Alternate Data TLB fault</li> <li>• VHPT Data fault</li> <li>• Data TLB fault</li> <li>• Data Page Not Present fault</li> <li>• Data NaT Page Consumption fault</li> <li>• Data Key Miss fault</li> </ul> If a fault is generated, this function does not return.
tlb_insert_data(slot, pte0, pte1, vaddr, rid, tr)	Inserts an entry into the DTLB, at the specified slot number. <code>pte0</code> , <code>pte1</code> compose the translation. <code>vaddr</code> and <code>rid</code> specify the virtual address and region identifier for the translation. If <code>tr</code> 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. <code>pte0</code> , <code>pte1</code> compose the translation. <code>vaddr</code> and <code>rid</code> specify the virtual address and region identifier for the translation. If <code>tr</code> 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 ( <code>vaddr</code> ), region identifier ( <code>rid</code> ) and page size ( <code>size</code> ). 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.

**Table 3-1. Pseudo-code Functions (Continued)**

Function	Operation
<code>tlb_may_purge_itc_entries(rid, vaddr, size)</code>	May locally purge ITC entries that match the specified virtual address ( <code>vaddr</code> ), region identifier ( <code>rid</code> ) and page size ( <code>size</code> ). 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.
<code>tlb_must_purge_dtc_entries(rid, vaddr, size)</code>	Purges all local, possibly overlapping, DTC entries matching the specified region identifier ( <code>rid</code> ), virtual address ( <code>vaddr</code> ) and page size ( <code>size</code> ). <code>vaddr{63:61}</code> (VRN) is ignored in the purge, i.e all entries that match <code>vaddr{60:0}</code> 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.
<code>tlb_must_purge_dtr_entries(rid, vaddr, size)</code>	Purges all local, possibly overlapping, DTR entries matching the specified region identifier ( <code>rid</code> ), virtual address ( <code>vaddr</code> ) and page size ( <code>size</code> ). <code>vaddr{63:61}</code> (VRN) is ignored in the purge, i.e all entries that match <code>vaddr{60:0}</code> 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.
<code>tlb_must_purge_itc_entries(rid, vaddr, size)</code>	Purges all local, possibly overlapping, ITC entry matching the specified region identifier ( <code>rid</code> ), virtual address ( <code>vaddr</code> ) and page size ( <code>size</code> ). <code>vaddr{63:61}</code> (VRN) is ignored in the purge, i.e all entries that match <code>vaddr{60:0}</code> 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.
<code>tlb_must_purge_itr_entries(rid, vaddr, size)</code>	Purges all local, possibly overlapping, ITR entry matching the specified region identifier ( <code>rid</code> ), virtual address ( <code>vaddr</code> ) and page size ( <code>size</code> ). <code>vaddr{63:61}</code> (VRN) is ignored in the purge, i.e all entries that match <code>vaddr{60:0}</code> 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.
<code>tlb_purge_translation_cache(loop)</code>	Removes 1 to N translations from the local processor's ITC and DTC. The number of entries removed is implementation specific. The parameter <code>loop</code> is used to generate an implementation-specific purge parameter.
<code>tlb_replacement_algorithm(tlb)</code>	Returns the next ITC or DTC slot number to replace. Replacement algorithms are implementation specific. <code>tlb</code> specifies to perform the algorithm on the ITC or DTC.
<code>tlb_search_pktr(key)</code>	Searches for a valid protection key register with a matching protection <code>key</code> . The search algorithm is implementation specific. Returns the PKR register slot number if found, otherwise returns Not Found.

**Table 3-1. Pseudo-code Functions (Continued)**

Function	Operation
<code>tlb_translate(vaddr, size, type, cpl, *attr, *defer)</code>	Returns the translated data physical address for the specified virtual memory address ( <code>vaddr</code> ) when translation enabled; otherwise, returns <code>vaddr</code> . <code>size</code> specifies the size of the access, <code>type</code> specifies the type of access (e.g., <code>read</code> , <code>write</code> , <code>advance</code> , <code>spec</code> ). <code>cpl</code> specifies the privilege level for access checking purposes. <code>*attr</code> returns the mapped physical memory attribute. If any fault conditions are detected and deferred, <code>tlb_translate</code> returns with <code>*defer</code> set. If a fault is generated but the fault is not deferred, <code>tlb_translate</code> does not return. <code>tlb_translate</code> checks the following faults: <ul style="list-style-type: none"> <li>• Unimplemented Data Address fault</li> <li>• Data Nested TLB fault</li> <li>• Alternate Data TLB fault</li> <li>• VHPT Data fault</li> <li>• Data TLB fault</li> <li>• Data Page Not Present fault</li> <li>• Data NaT Page Consumption fault</li> <li>• Data Key Miss fault</li> <li>• Data Key Permission fault</li> <li>• Data Access Rights fault</li> <li>• Data Dirty Bit fault</li> <li>• Data Access Bit fault</li> <li>• Data Debug fault</li> <li>• Unaligned Data Reference fault</li> <li>• Unsupported Data Reference fault</li> </ul>
<code>tlb_translate_nonaccess(vaddr, type)</code>	Returns the translated data physical address for the specified virtual memory address ( <code>vaddr</code> ). <code>type</code> specifies the type of access (e.g., <code>FC</code> , <code>TPA</code> ). If a fault is generated, <code>tlb_translate_nonaccess</code> does not return. The following faults are checked: <ul style="list-style-type: none"> <li>• Unimplemented Data Address fault</li> <li>• Virtualization fault (<code>tpa</code> only)</li> <li>• Data Nested TLB fault</li> <li>• Alternate Data TLB fault</li> <li>• VHPT Data fault</li> <li>• Data TLB fault</li> <li>• Data Page Not Present fault</li> <li>• Data NaT Page Consumption fault</li> <li>• Data Access Rights fault (<code>fc</code> only)</li> </ul>
<code>tlb_vhpt_hash(vrn, vaddr61, rid, size)</code>	Generates a VHPT entry address for the specified virtual region number ( <code>vrn</code> ) and 61-bit virtual offset ( <code>vaddr61</code> ), region identifier ( <code>rid</code> ) and page size ( <code>size</code> ). <code>Tlb_vhpt_hash</code> hashes <code>vaddr</code> , <code>rid</code> and <code>size</code> 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.
<code>tlb_vhpt_tag(vaddr, rid, size)</code>	Generates a VHPT tag identifier for the specified virtual address ( <code>vaddr</code> ), region identifier ( <code>rid</code> ) and page size ( <code>size</code> ). <code>Tlb_vhpt_tag</code> hashes the <code>vaddr</code> , <code>rid</code> and <code>size</code> 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).
<code>undefined()</code>	Returns an undefined 64-bit value.
<code>undefined_behavior()</code>	Causes undefined processor behavior. Extent of undefined behavior is described in <a href="#">Section 3.5, “Undefined Behavior” on page 1:44</a> .

**Table 3-1. Pseudo-code Functions (Continued)**

Function	Operation
unimplemented_physical_address(paddr)	Return TRUE if the presented physical address is unimplemented on this processor model; FALSE otherwise. This function is model specific.
unimplemented_virtual_address(vaddr, vm)	Return TRUE if the presented virtual address is unimplemented on this processor model; FALSE otherwise. If vm is 1, one additional bit of virtual address is treated as unimplemented. This function is model specific.
vm_all_probes()	Returns TRUE if the processor is configured to virtualize all probe instructions when PSR.vm is 1. See <a href="#">Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344</a> for details.
vm_disabled()	Returns TRUE if the processor implements the PSR.vm bit and virtual machine features are disabled. See <a href="#">Section 3.4, “Processor Virtualization” on page 2:44 in SDM</a> and <a href="#">“PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:446 in SDM</a> for details.
vm_select_probes()	Returns TRUE if the processor is configured to virtualize selected probe instructions when PSR.vm is 1. See <a href="#">Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344</a> for details.
vmsw_disabled()	Returns TRUE if the processor implements the PSR.vm bit and the vmsw instruction is disabled. See <a href="#">Section 3.4, “Processor Virtualization” on page 2:44 in SDM</a> and <a href="#">“PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:446 in SDM</a> for details.
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.

§



Each Itanium 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:

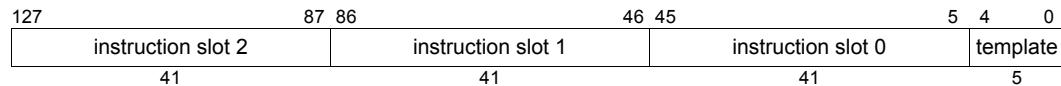
**Table 4-1. Relationship between Instruction Type and Execution Unit Type**

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 <sup>a</sup>

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 1:38 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 <sup>a</sup>
05	M-unit	L-unit	X-unit <sup>a</sup>
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 5 instruction types — ALU (A), Integer (I), Memory (M), Floating-point (F), and Branch (B). Bundle template bits are used to distinguish among the 4 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](#), brown in the color version) cause an Illegal Operation fault.
- Reserved if PR[qp] is 1 major ops (dark gray in the gray scale version of [Table 4-3](#), purple in the color version) 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](#), cyan in the color version) 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 (purple) only in their RAW dependency behavior (see “[RAW Dependency Table](#)” on [page 3:374](#)).

**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		Sys/Mem Mgmt 1	FP Misc 1	Indirect Call 1	
2			2	Indirect Predict/Nop 2	
3		3	3		3
4	Deposit 4	Int Ld +Reg/getf 4	FP Compare 4	IP-relative Branch 4	
5	Shift/Test Bit 5	Int Ld/St +Imm 5	FP Class 5	IP-rel Call 5	
6		FP Ld/St +Reg/setf 6	6		movl 6
7	MM Mpy/Shift 7	FP Ld/St +Imm 7	7	IP-relative Predict 7	
8	ALU/MM ALU 8	ALU/MM ALU 8	fma 8	e 8	
9	Add Imm <sub>22</sub> 9	Add Imm <sub>22</sub> 9	fma 9	e 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 on page 3:296](#) summarizes all the instruction formats. The instruction fields are color-coded for ease of identification, as described in [Table 4-5 on page 3:298](#). A color version of this chapter is available for those heavily involved in working with the instruction encodings.

The instruction field names, used throughout this chapter, are described in [Table 4-6 on page 3:298](#). The set of special notations (such as whether an instruction is privileged) are listed in [Table 4-7 on page 3:299](#). 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<sub>14</sub> instruction (format A4) is formed from the imm<sub>7b</sub>, imm<sub>6d</sub>, and s fields. [Table 4-74 on page 3:368](#) shows how the immediates are formed from the instruction fields for each instruction which has an immediate.

**Table 4-4. Instruction Format Summary**

	4	03938373635343332313029282726252423222120191817161514131211109876543210
ALU	A1	8 $x_{2a} v_e x_4 x_{2b}$ $r_3$ $r_2$ $r_1$ qp
Shift L and Add	A2	8 $x_{2a} v_e x_4 ct_{2d}$ $r_3$ $r_2$ $r_1$ qp
ALU Imm <sub>8</sub>	A3	8 s $x_{2a} v_e x_4 x_{2b}$ $r_3$ imm <sub>7b</sub> $r_1$ qp
Add Imm <sub>14</sub>	A4	8 s $x_{2a} v_e$ imm <sub>6d</sub> $r_3$ imm <sub>7b</sub> $r_1$ qp
Add Imm <sub>22</sub>	A5	9 s imm <sub>9d</sub> imm <sub>5c</sub> $r_3$ imm <sub>7b</sub> $r_1$ qp
Compare	A6	C - E $t_b x_2 t_a p_2$ $r_3$ $r_2$ C p <sub>1</sub> qp
Compare to Zero	A7	C - E $t_b x_2 t_a p_2$ $r_3$ 0 C p <sub>1</sub> qp
Compare Imm <sub>8</sub>	A8	C - E s $x_2 t_a p_2$ $r_3$ imm <sub>7b</sub> C p <sub>1</sub> qp
MM ALU	A9	8 Za $x_{2a} z_b x_4 x_{2b}$ $r_3$ $r_2$ $r_1$ qp
MM Shift and Add	A10	8 Za $x_{2a} z_b$ ct <sub>2d</sub> $r_3$ $r_2$ $r_1$ qp
MM Multiply Shift	I1	7 Za $x_{2a} z_b v_e ct_{2d} x_{2b}$ $r_3$ $r_2$ $r_1$ qp
MM Mpy/Mix/Pack	I2	7 Za $x_{2a} z_b v_e x_{2c} x_{2b}$ $r_3$ $r_2$ $r_1$ qp
MM Mux1	I3	7 Za $x_{2a} z_b v_e x_{2c} x_{2b}$ mbt <sub>4c</sub> $r_2$ $r_1$ qp
MM Mux2	I4	7 Za $x_{2a} z_b v_e x_{2c} x_{2b}$ mht <sub>8c</sub> $r_2$ $r_1$ qp
Shift R Variable	I5	7 Za $x_{2a} z_b v_e x_{2c} x_{2b}$ $r_3$ $r_2$ $r_1$ qp
MM Shift R Fixed	I6	7 Za $x_{2a} z_b v_e x_{2c} x_{2b}$ $r_3$ count <sub>5b</sub> $r_1$ qp
Shift L Variable	I7	7 Za $x_{2a} z_b v_e x_{2c} x_{2b}$ $r_3$ $r_2$ $r_1$ qp
MM Shift L Fixed	I8	7 Za $x_{2a} z_b v_e x_{2c} x_{2b}$ ccount <sub>5c</sub> $r_2$ $r_1$ qp
Bit Strings	I9	7 Za $x_{2a} z_b v_e x_{2c} x_{2b}$ $r_3$ 0 $r_1$ qp
Shift Right Pair	I10	5 x <sub>2</sub> X count <sub>6d</sub> $r_3$ $r_2$ $r_1$ qp
Extract	I11	5 x <sub>2</sub> X len <sub>6d</sub> $r_3$ pos <sub>6b</sub> y $r_1$ qp
Dep.Z	I12	5 x <sub>2</sub> X len <sub>6d</sub> y cpos <sub>6c</sub> $r_2$ $r_1$ qp
Dep.Z Imm <sub>8</sub>	I13	5 s x <sub>2</sub> X len <sub>6d</sub> y cpos <sub>6c</sub> imm <sub>7b</sub> $r_1$ qp
Deposit Imm <sub>1</sub>	I14	5 s x <sub>2</sub> X len <sub>6d</sub> $r_3$ cpos <sub>6b</sub> $r_1$ qp
Deposit	I15	4 cpos <sub>6d</sub> len <sub>4d</sub> $r_3$ $r_2$ $r_1$ qp
Test Bit	I16	5 t <sub>b</sub> x <sub>2</sub> t <sub>a</sub> p <sub>2</sub> $r_3$ pos <sub>6b</sub> y C p <sub>1</sub> qp
Test NaT	I17	5 t <sub>b</sub> x <sub>2</sub> t <sub>a</sub> p <sub>2</sub> $r_3$ x y C p <sub>1</sub> qp
Nop/Hint	I18	0 I x <sub>3</sub> x <sub>6</sub> y imm <sub>20a</sub> $r_1$ qp
Break	I19	0 I x <sub>3</sub> x <sub>6</sub> imm <sub>20a</sub> $r_1$ qp
Int Spec Check	I20	0 s x <sub>3</sub> imm <sub>13c</sub> $r_2$ imm <sub>7a</sub> $r_1$ qp
Move to BR	I21	0 x <sub>3</sub> timm <sub>9c</sub> If x wh $r_2$ b <sub>1</sub> $r_1$ qp
Move from BR	I22	0 x <sub>3</sub> x <sub>6</sub> $r_2$ b <sub>2</sub> $r_1$ qp
Move to Pred	I23	0 s x <sub>3</sub> mask <sub>8c</sub> $r_2$ mask <sub>7a</sub> $r_1$ qp
Move to Pred Imm <sub>44</sub>	I24	0 s x <sub>3</sub> imm <sub>27a</sub> $r_1$ qp
Move from Pred/IP	I25	0 x <sub>3</sub> x <sub>6</sub> $r_1$ qp
Move to AR	I26	0 x <sub>3</sub> x <sub>6</sub> ar <sub>3</sub> $r_2$ $r_1$ qp
Move to AR Imm <sub>8</sub>	I27	0 s x <sub>3</sub> x <sub>6</sub> ar <sub>3</sub> imm <sub>7b</sub> $r_1$ qp
Move from AR	I28	0 x <sub>3</sub> x <sub>6</sub> ar <sub>3</sub> $r_1$ qp
Sxt/Zxt/Czx	I29	0 x <sub>3</sub> x <sub>6</sub> $r_3$ $r_1$ qp
Test Feature	I30	5 t <sub>b</sub> x <sub>2</sub> t <sub>a</sub> p <sub>2</sub> 0 x imm <sub>5b</sub> y C p <sub>1</sub> qp
Int Load	M1	4 m x <sub>6</sub> hint x $r_3$ $r_1$ qp
Int Load +Reg	M2	4 m x <sub>6</sub> hint x $r_3$ $r_1$ qp
Int Load +Imm	M3	5 s x <sub>6</sub> hint i $r_3$ imm <sub>7b</sub> $r_1$ qp
Int Store	M4	4 m x <sub>6</sub> hint x $r_3$ $r_2$ $r_1$ qp
Int Store +Imm	M5	5 s x <sub>6</sub> hint i $r_3$ $r_2$ imm <sub>7a</sub> $r_1$ qp
FP Load	M6	6 m x <sub>6</sub> hint x $r_3$ f <sub>1</sub> $r_1$ qp
FP Load +Reg	M7	6 m x <sub>6</sub> hint x $r_3$ f <sub>1</sub> $r_1$ qp
FP Load +Imm	M8	7 s x <sub>6</sub> hint i $r_3$ imm <sub>7b</sub> f <sub>1</sub> $r_1$ qp
FP Store	M9	6 m x <sub>6</sub> hint x $r_3$ f <sub>2</sub> $r_1$ qp
FP Store +Imm	M10	7 s x <sub>6</sub> hint i $r_3$ f <sub>2</sub> imm <sub>7a</sub> $r_1$ qp
FP Load Pair	M11	6 m x <sub>6</sub> hint x $r_3$ f <sub>2</sub> f <sub>1</sub> $r_1$ qp
FP Load Pair +Imm	M12	6 m x <sub>6</sub> hint x $r_3$ f <sub>2</sub> f <sub>1</sub> $r_1$ qp
Line Prefetch	M13	6 m x <sub>6</sub> hint x $r_3$ $r_1$ qp
Line Prefetch +Reg	M14	6 m x <sub>6</sub> hint x $r_3$ $r_2$ $r_1$ qp
Line Prefetch +Imm	M15	7 s x <sub>6</sub> hint i $r_3$ imm <sub>7b</sub> $r_1$ qp
(Cmp &) Exchg	M16	4 m x <sub>6</sub> hint x $r_3$ $r_2$ $r_1$ qp
Fetch & Add	M17	4 m x <sub>6</sub> hint x $r_3$ s i <sub>2b</sub> $r_1$ qp
Set FR	M18	6 m x <sub>6</sub> x $r_2$ f <sub>1</sub> $r_1$ qp
Get FR	M19	4 m x <sub>6</sub> x $r_2$ f <sub>2</sub> $r_1$ qp

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

**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
Int Spec Check	M20	1	s	x <sub>3</sub>																																qp						
FP Spec Check	M21	1	s	x <sub>3</sub>																																qp						
Int ALAT Check	M22	0	s	x <sub>3</sub>																																qp						
FP ALAT Check	M23	0	s	x <sub>3</sub>																																qp						
Sync/Srlz/ALAT	M24	0		x <sub>3</sub>	x <sub>2</sub>	x <sub>4</sub>																													qp							
RSE Control	M25	0		x <sub>3</sub>	x <sub>2</sub>	x <sub>4</sub>																													0							
Int ALAT Inval	M26	0		x <sub>3</sub>	x <sub>2</sub>	x <sub>4</sub>																													qp							
FP ALAT Inval	M27	0		x <sub>3</sub>	x <sub>2</sub>	x <sub>4</sub>																													qp							
Flush Cache	M28	1	x	x <sub>3</sub>		x <sub>6</sub>																													qp							
Move to AR	M29	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Move to AR Imm <sub>8</sub>	M30	0	s	x <sub>3</sub>	x <sub>2</sub>	x <sub>4</sub>																													qp							
Move from AR	M31	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Move to CR	M32	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Move from CR	M33	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Alloc	M34	1		x <sub>3</sub>		sor																														qp						
Move to PSR	M35	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Move from PSR	M36	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Break	M37	0	i	x <sub>3</sub>	x <sub>2</sub>	x <sub>4</sub>																													qp							
Probe	M38	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Probe Imm <sub>2</sub>	M39	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Probe Fault Imm <sub>2</sub>	M40	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
TC Insert	M41	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Mv to Ind/TR Ins	M42	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Mv from Ind	M43	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Set/Reset Mask	M44	0	i	x <sub>3</sub>	i <sub>2d</sub>	x <sub>4</sub>																													qp							
Translation Purge	M45	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Translation Access	M46	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
TC Entry Purge	M47	1		x <sub>3</sub>		x <sub>6</sub>																													qp							
Nop/Hint	M48	0	I	x <sub>3</sub>	x <sub>2</sub>	x <sub>4</sub>	y																												qp							
IP-Relative Branch	B1	4	s	d	wh																															btype	qp					
Counted Branch	B2	4	s	d	wh																															0						
IP-Relative Call	B3	5	s	d	wh																															b1	qp					
Indirect Branch	B4	0		d	wh	x <sub>6</sub>																													btype	qp						
Indirect Call	B5	1		d	wh																															b1	qp					
IP-Relative Predict	B6	7	s	lh	t <sub>2e</sub>																														wh							
Indirect Predict	B7	2		lh	t <sub>2e</sub>	x <sub>6</sub>																													wh							
Misc	B8	0				x <sub>6</sub>																															0					
Break/Nop/Hint	B9	0/2	i			x <sub>6</sub>																														qp						
FP Arithmetic	F1	8 - D	x	sf	r <sub>a</sub>	t <sub>4</sub>																													qp							
Fixed Multiply Add	F2	E	x	x <sub>2</sub>		t <sub>4</sub>																													qp							
FP Select	F3	E	x			t <sub>4</sub>																													qp							
FP Compare	F4	4	r <sub>b</sub>	sf	r <sub>a</sub>	p <sub>2</sub>																												p <sub>1</sub>	qp							
FP Class	F5	5		fc <sub>2</sub>		p <sub>2</sub>																													p <sub>1</sub>	qp						
FP Recip Approx	F6	0 - 1	q	sf	x	p <sub>2</sub>																													f <sub>1</sub>	qp						
FP Recip Sqrt App	F7	0 - 1	q	sf	x	p <sub>2</sub>																													f <sub>1</sub>	qp						
FP Min/Max/Pcmp	F8	0 - 1		sf	x	x <sub>6</sub>																													f <sub>1</sub>	qp						
FP Merge/Logical	F9	0 - 1			x	x <sub>6</sub>																													f <sub>1</sub>	qp						
Convert FP to Fixed	F10	0 - 1		sf	x	x <sub>6</sub>																													f <sub>1</sub>	qp						
Convert Fixed to FP	F11	0			x	x <sub>6</sub>																													f <sub>1</sub>	qp						
FP Set Controls	F12	0		sf	x	x <sub>6</sub>																														qp						
FP Clear Flags	F13	0		sf	x	x <sub>6</sub>																														qp						
FP Check Flags	F14	0	s	sf	x	x <sub>6</sub>																														qp						
Break	F15	0	i		x	x <sub>6</sub>																													qp							
Nop/Hint	F16	0	I		x	x <sub>6</sub>	y																												qp							
Break	X1	0	I	x <sub>3</sub>	x <sub>6</sub>																														imm <sub>41</sub>							
Move Imm <sub>64</sub>	X2	6	I		imm <sub>9d</sub>																														imm <sub>41</sub>							
Long Branch	X3	C	I	d	wh																														imm <sub>39</sub>							
Long Call	X4	D	I	d	wh																														imm <sub>39</sub>							
Nop/Hint	X5	0	I	x <sub>3</sub>	x <sub>6</sub>	y																													imm <sub>41</sub>							

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

**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

**Table 4-6. Instruction Field Names**

Field Name	Description
ar <sub>3</sub>	application register source/target
b <sub>1</sub> , b <sub>2</sub>	branch register source/target
btype	branch type opcode extension
c	complement compare relation opcode extension
ccount <sub>5c</sub>	multimedia shift left complemented shift count immediate
count <sub>5b</sub> , count <sub>6d</sub>	multimedia shift right/shift right pair shift count immediate
cpos <sub>x</sub>	deposit complemented bit position immediate
cr <sub>3</sub>	control register source/target
ct <sub>2d</sub>	multimedia multiply shift/shift and add shift count immediate
d	branch cache deallocation hint opcode extension
f <sub>n</sub>	floating-point register source/target
fc <sub>2</sub> , fclass <sub>7c</sub>	floating-point class immediate
hint	memory reference hint opcode extension
i, i <sub>2b</sub> , i <sub>2d</sub> , imm <sub>x</sub>	immediate of length 1, 2, or x
ih	branch importance hint opcode extension
len <sub>4d</sub> , len <sub>6d</sub>	extract/deposit length immediate
m	memory reference post-modify opcode extension
mask <sub>x</sub>	predicate immediate mask
mbt <sub>4c</sub> , mht <sub>8c</sub>	multimedia mux1/mux2 immediate
p	sequential prefetch hint opcode extension
p <sub>1</sub> , p <sub>2</sub>	predicate register target
pos <sub>6b</sub>	test bit/extract bit position immediate
q	floating-point reciprocal/reciprocal square-root opcode extension
qp	qualifying predicate register source
r <sub>n</sub>	general register source/target
s	immediate sign bit
sf	floating-point status field opcode extension

**Table 4-6. Instruction Field Names (Continued)**

Field Name	Description
sof, sol, sor	alloc size of frame, size of locals, size of rotating immediates
t <sub>a</sub> , t <sub>b</sub>	compare type opcode extension
t <sub>2e</sub> , timm <sub>x</sub>	branch predict tag immediate
v <sub>x</sub>	reserved opcode extension field
wh	branch whether hint opcode extension
x, x <sub>n</sub>	opcode extension of length 1 or n
y	extract/deposit/test bit/test NaT/hint opcode extension
z <sub>a</sub> , z <sub>b</sub>	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 (cyan) encodings and non-branch instructions with a qualifying predicate, when its PR[qp] is 1, or for Reserved (brown) encodings, unconditionally
f	instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0
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

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 3:310, “M-Unit Instruction Encodings” on page 3:323, “B-Unit Instruction Encodings” on page 3:349, “F-Unit Instruction Encodings” on page 3:356, and “X-Unit Instruction Encodings” on page 3:365.

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 3:296. 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 color entries in the tables) execute as `nop` instructions.
- Reserved instructions (light gray color in the gray scale version of the tables, brown color in the color version) cause an Illegal Operation fault.
- Reserved if PR[qp] is 1 instructions (dark gray in the gray scale version of the tables, purple in the color version) 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, cyan in the color version) 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 (purple) only in their RAW dependency behavior (see “RAW Dependency Table” on page 3:374).

Some processors may implement the Reserved if PR[qp] is 1 (purple) and Reserved if PR[qp] is 1 B-unit (cyan) encodings in the L+X opcode space as Reserved (brown). These encodings appear in the L+X column of [Table 4-3 on page 3:295](#), and in [Table 4-69 on page 3:366](#), [Table 4-70 on page 3:366](#), [Table 4-71 on page 3:367](#), and [Table 4-72 on page 3:367](#). On processors which implement these encodings as Reserved (brown), the operating system is required to provide an Illegal Operation fault handler which emulates them as Reserved if PR[qp] is 1 (cyan/purple) 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.

Unused opcode hint extension values (white color entries in Hint Completer tables) should not be used by software. Processors must perform the architected functional behavior of the instruction independent of the hint extension value (whether defined or unused), but different processor models may interpret unused opcode hint extension values in different ways, resulting in undesirable performance effects.

## 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 3:306](#) 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 ( <a href="#">Table 4-9</a> )	
	1	Multimedia ALU 1-bit+2-bit Ext ( <a href="#">Table 4-12</a> )	
	2	adds – imm <sub>14</sub> A4	
	3	addp4 – imm <sub>14</sub> A4	

**Table 4-9. Integer ALU 4-bit+2-bit Opcode Extensions**

Opcode Bits 40:37	x <sub>2a</sub> Bits 35:34	v <sub>e</sub> Bit 33	x <sub>4</sub> Bits 32:29	x <sub>2b</sub> 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 <sub>8</sub> A3		
			A				
			B	and - imm <sub>8</sub> A3	andcm - imm <sub>8</sub> A3	or - imm <sub>8</sub> A3	xor - imm <sub>8</sub> A3
			C				
			D				
			E				
			F				

#### 4.2.1.1 Integer ALU – Register-Register

A1	40 37 36 35 34 33 32 29 28 27 26 20 19 13 12 6 5 0																		
	<table border="1"> <tr> <td>8</td> <td>x<sub>2a</sub></td> <td>v<sub>e</sub></td> <td>x<sub>4</sub></td> <td>x<sub>2b</sub></td> <td>r<sub>3</sub></td> <td>r<sub>2</sub></td> <td>r<sub>1</sub></td> <td>qp</td> </tr> <tr> <td>4</td> <td>1</td> <td>2</td> <td>1</td> <td>4</td> <td>2</td> <td>7</td> <td>7</td> <td>6</td> </tr> </table>	8	x <sub>2a</sub>	v <sub>e</sub>	x <sub>4</sub>	x <sub>2b</sub>	r <sub>3</sub>	r <sub>2</sub>	r <sub>1</sub>	qp	4	1	2	1	4	2	7	7	6
8	x <sub>2a</sub>	v <sub>e</sub>	x <sub>4</sub>	x <sub>2b</sub>	r <sub>3</sub>	r <sub>2</sub>	r <sub>1</sub>	qp											
4	1	2	1	4	2	7	7	6											

Instruction	Operands	Opcode	Extension			
			x <sub>2a</sub>	v <sub>e</sub>	x <sub>4</sub>	x <sub>2b</sub>
add	r <sub>1</sub> = r <sub>2</sub> , r <sub>3</sub> r <sub>1</sub> = r <sub>2</sub> , r <sub>3</sub> , 1	8	0	0	0	0
sub	r <sub>1</sub> = r <sub>2</sub> , r <sub>3</sub> r <sub>1</sub> = r <sub>2</sub> , r <sub>3</sub> , 1				1	1
addp4					1	0
and					2	0
andcm	r <sub>1</sub> = r <sub>2</sub> , r <sub>3</sub>				3	0
or		8	0	0	0	1
xor					2	2

#### 4.2.1.2 Shift Left and Add

A2	40 37 36 35 34 33 32 29 28 27 26 20 19 13 12 6 5 0																		
	<table border="1"> <tr> <td>8</td> <td>x<sub>2a</sub></td> <td>v<sub>e</sub></td> <td>x<sub>4</sub></td> <td>ct<sub>2d</sub></td> <td>r<sub>3</sub></td> <td>r<sub>2</sub></td> <td>r<sub>1</sub></td> <td>qp</td> </tr> <tr> <td>4</td> <td>1</td> <td>2</td> <td>1</td> <td>4</td> <td>2</td> <td>7</td> <td>7</td> <td>6</td> </tr> </table>	8	x <sub>2a</sub>	v <sub>e</sub>	x <sub>4</sub>	ct <sub>2d</sub>	r <sub>3</sub>	r <sub>2</sub>	r <sub>1</sub>	qp	4	1	2	1	4	2	7	7	6
8	x <sub>2a</sub>	v <sub>e</sub>	x <sub>4</sub>	ct <sub>2d</sub>	r <sub>3</sub>	r <sub>2</sub>	r <sub>1</sub>	qp											
4	1	2	1	4	2	7	7	6											

Instruction	Operands	Opcode	Extension		
			x <sub>2a</sub>	v <sub>e</sub>	x <sub>4</sub>
shladd	r <sub>1</sub> = r <sub>2</sub> , count <sub>2</sub> , r <sub>3</sub>	8	0	0	4
shladdp4					6

#### 4.2.1.3 Integer ALU – Immediate<sub>8</sub>-Register

A3	40	37	36	35	34	33	32	29	28	27	26	20	19	13	12	6	5	0
	8	S	x <sub>2a</sub>	v <sub>e</sub>	x <sub>4</sub>	x <sub>2b</sub>	r <sub>3</sub>		imm <sub>7b</sub>		r <sub>1</sub>		qp					
	4	1	2	1	4	2	7		7		7		6					

Instruction	Operands	Opcode	Extension			
			x <sub>2a</sub>	v <sub>e</sub>	x <sub>4</sub>	x <sub>2b</sub>
sub					9	1
and					0	
andcm	r <sub>1</sub> = imm <sub>8</sub> , r <sub>3</sub>	8	0	0	B	1
or					2	
xor					3	

#### 4.2.1.4 Add Immediate<sub>14</sub>

A4	40	37	36	35	34	33	32	27	26	20	19	13	12	6	5	0
	8	S	x <sub>2a</sub>	v <sub>e</sub>	imm <sub>6d</sub>	r <sub>3</sub>		imm <sub>7b</sub>		r <sub>1</sub>		qp				
	4	1	2	1	6	7		7		7		6				

Instruction	Operands	Opcode	Extension	
			x <sub>2a</sub>	v <sub>e</sub>
adds	r <sub>1</sub> = imm <sub>14</sub> , r <sub>3</sub>	8	2	0
addp4			3	

#### 4.2.1.5 Add Immediate<sub>22</sub>

A5	40	37	36	35	27	26	22	21	20	19	13	12	6	5	0
	9	S	imm <sub>9d</sub>	imm <sub>5c</sub>	r <sub>3</sub>		imm <sub>7b</sub>		r <sub>1</sub>		qp				
	4	1	9	5	2		7		7		6				

Instruction	Operands	Opcode
addl	r <sub>1</sub> = imm <sub>22</sub> , r <sub>3</sub>	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<sub>2</sub>) in bits 35:34 and three 1-bit opcode extension fields in bits 33 (t<sub>a</sub>), 36 (t<sub>b</sub>), 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<sub>2</sub>) in bits 35:34 and two 1-bit opcode extension fields in bits 33 (t<sub>a</sub>) and 12 (c), as shown in Table 4-11.

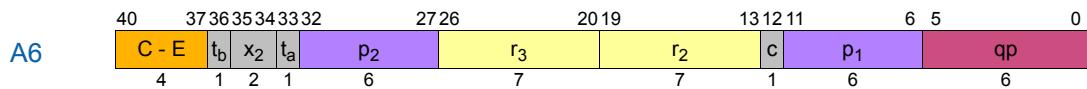
**Table 4-10. Integer Compare Opcode Extensions**

x <sub>2</sub> Bits 35:34	t <sub>b</sub> Bit 36	t <sub>a</sub> Bit 33	c Bit 12	Opcode Bits 40:37		
				C	D	E
0	0	0	0	cmp.lt A6	cmp.lt.u A6	cmp.eq A6
			1	cmp.lt.unc A6	cmp.lt.u.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
1	0	0	0	cmp4.lt A6	cmp4.lt.u A6	cmp4.eq A6
			1	cmp4.lt.unc A6	cmp4.lt.u.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**

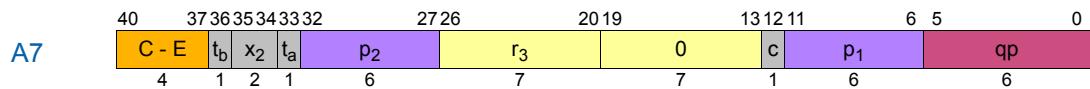
x <sub>2</sub> Bits 35:34	t <sub>a</sub> Bit 33	c Bit 12	Opcode Bits 40:37		
			C	D	E
2	0	0	cmp.lt – imm <sub>8</sub> A8	cmp.lt.u – imm <sub>8</sub> A8	cmp.eq – imm <sub>8</sub> A8
		1	cmp.lt.unc – imm <sub>8</sub> A8	cmp.lt.u.unc – imm <sub>8</sub> A8	cmp.eq.unc – imm <sub>8</sub> A8
	1	0	cmp.eq.and – imm <sub>8</sub> A8	cmp.eq.or – imm <sub>8</sub> A8	cmp.eq.or.andcm – imm <sub>8</sub> A8
		1	cmp.ne.and – imm <sub>8</sub> A8	cmp.ne.or – imm <sub>8</sub> A8	cmp.ne.or.andcm – imm <sub>8</sub> A8
3	0	0	cmp4.lt – imm <sub>8</sub> A8	cmp4.lt.u – imm <sub>8</sub> A8	cmp4.eq – imm <sub>8</sub> A8
		1	cmp4.lt.unc – imm <sub>8</sub> A8	cmp4.lt.u.unc – imm <sub>8</sub> A8	cmp4.eq.unc – imm <sub>8</sub> A8
	1	0	cmp4.eq.and – imm <sub>8</sub> A8	cmp4.eq.or – imm <sub>8</sub> A8	cmp4.eq.or.andcm – imm <sub>8</sub> A8
		1	cmp4.ne.and – imm <sub>8</sub> A8	cmp4.ne.or – imm <sub>8</sub> A8	cmp4.ne.or.andcm – imm <sub>8</sub> A8

#### 4.2.2.1 Integer Compare – Register-Register



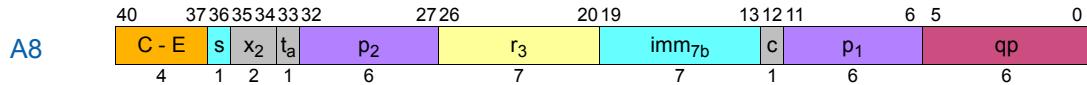
Instruction	Operands	Opcode	Extension			
			$x_2$	$t_b$	$t_a$	$c$
cmp.lt	$p_1, p_2 = r_2, r_3$	C	0	0	0	0
cmp.ltu		D				1
cmp.eq		E				1
cmp.lt.unc		C	0	0	1	0
cmp.ltu.unc		D				0
cmp.eq.unc		E				1
cmp.eq.and		C	1	0	1	0
cmp.eq.or		D				1
cmp.eq.or.andcm		E				1
cmp.ne.and		C	1	0	1	0
cmp.ne.or		D				1
cmp.ne.or.andcm		E				1
cmp4.lt		C	0	0	0	0
cmp4.ltu		D				1
cmp4.eq		E				1
cmp4.lt.unc		C	1	0	1	0
cmp4.ltu.unc		D				0
cmp4.eq.unc		E				1
cmp4.eq.and		C	1	0	1	0
cmp4.eq.or		D				1
cmp4.eq.or.andcm		E				1
cmp4.ne.and		C	1	0	1	0
cmp4.ne.or		D				1
cmp4.ne.or.andcm		E				1

#### 4.2.2.2 Integer Compare to Zero – Register



Instruction	Operands	Opcode	Extension			
			x <sub>2</sub>	t <sub>b</sub>	t <sub>a</sub>	c
cmp.gt.and	$p_1, p_2 = r0, r_3$	C	0	0	0	0
cmp.gt.or		D				1
cmp.gt.or.andcm		E				
cmp.le.and		C	1	1	1	0
cmp.le.or		D				1
cmp.le.or.andcm		E				
cmp.ge.and		C	1	0	0	0
cmp.ge.or		D				1
cmp.ge.or.andcm		E				
cmp.lt.and		C	1	0	0	0
cmp.lt.or		D				1
cmp.lt.or.andcm		E				
cmp4.gt.and		C	1	0	0	0
cmp4.gt.or		D				1
cmp4.gt.or.andcm		E				
cmp4.le.and		C	1	0	0	0
cmp4.le.or		D				1
cmp4.le.or.andcm		E				
cmp4.ge.and		C	1	1	1	0
cmp4.ge.or		D				1
cmp4.ge.or.andcm		E				
cmp4.lt.and		C	1	1	1	0
cmp4.lt.or		D				1
cmp4.lt.or.andcm		E				

#### 4.2.2.3 Integer Compare – Immediate-Register



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			1
cmp.eq		E			
cmp.lt.unc		C			
cmp.ltu.unc		D			
cmp.eq.unc		E			
cmp.eq.and		C		0	0
cmp.eq.or		D		1	1
cmp.eq.or.andcm		E			
cmp.ne.and		C			
cmp.ne.or		D			
cmp.ne.or.andcm		E			
cmp4.lt		C	3	0	0
cmp4.ltu		D			1
cmp4.eq		E			
cmp4.lt.unc		C		0	0
cmp4.ltu.unc		D		1	1
cmp4.eq.unc		E			
cmp4.eq.and		C		0	0
cmp4.eq.or		D		1	1
cmp4.eq.or.andcm		E			
cmp4.ne.and		C			
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 3:307](#).

**Table 4-12. Multimedia ALU 2-bit+1-bit Opcode Extensions**

Opcodes Bits 40:37	$x_{2a}$ Bits 35:34	$z_a$ Bit 36	$z_b$ Bit 33	
8	1	0	0	Multimedia ALU Size 1 ( <a href="#">Table 4-13</a> )
			1	Multimedia ALU Size 2 ( <a href="#">Table 4-14</a> )
	1	1	0	Multimedia ALU Size 4 ( <a href="#">Table 4-15</a> )
			1	

**Table 4-13. Multimedia ALU Size 1 4-bit+2-bit Opcode Extensions**

Opcode Bits 40:37	x <sub>2a</sub> Bits 35:34	z <sub>a</sub> Bit 36	z <sub>b</sub> Bit 33	x <sub>4</sub> Bits 32:29	x <sub>2b</sub> 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				

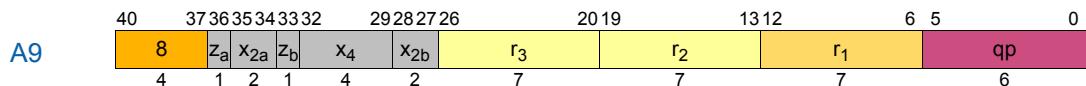
**Table 4-14. Multimedia ALU Size 2 4-bit+2-bit Opcode Extensions**

Opcode Bits 40:37	x <sub>2a</sub> Bits 35:34	z <sub>a</sub> Bit 36	z <sub>b</sub> Bit 33	x <sub>4</sub> Bits 32:29	x <sub>2b</sub> 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**

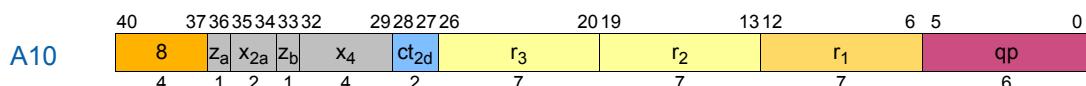
Opcode Bits 40:37	x <sub>2a</sub> Bits 35:34	z <sub>a</sub> Bit 36	z <sub>b</sub> Bit 33	x <sub>4</sub> Bits 32:29	x <sub>2b</sub> Bits 28:27			
					0	1	2	3
8	1	1	0	0	padd4 A9			
				1	psub4 A9			
				2				
				3				
				4				
				5				
				6				
				7				
				8				
				9	pcmp4.eq A9	pcmp4.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

#### 4.2.3.1 Multimedia ALU



Instruction	Operands	Opcode	Extension				
			$x_{2a}$	$z_a$	$z_b$	$x_4$	$x_{2b}$
padd1	$r_1 = r_2, r_3$	8	0	0	0		
padd2			0	1	1		0
padd4			1	0			
padd1.sss			0	0			
padd2.sss			0	1			1
padd1.uuu			0	0			2
padd2.uuu			0	1			
padd1.uus			0	0			3
padd2.uus			0	1			
psub1			0	0			
psub2			0	1			0
psub4			1	0			
psub1.sss			0	0			
psub2.sss			0	1			1
psub1.uuu			0	0			2
psub2.uuu			0	1			
psub1.uus			0	0			3
psub2.uus			0	1			
pavg1			0	0			
pavg2			0	1			2
pavg1.raz			0	0			
pavg2.raz			0	1			3
pavgsub1			0	0		3	2
pavgsub2			0	1			
pcmp1.eq			0	0			
pcmp2.eq			0	1			0
pcmp4.eq			1	0			
pcmp1.gt			0	0			
pcmp2.gt			0	1			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	$r_1 = r_2, count_2, r_3$	8	1	0	1	4
pshradd2						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 ( <a href="#">Table 4-17</a> )	
		1	Multimedia Size 2 ( <a href="#">Table 4-18</a> )	
	1	0	Multimedia Size 4 ( <a href="#">Table 4-19</a> )	
		1	Variable Shift ( <a href="#">Table 4-20</a> )	

**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 <a href="#">I2</a>	mix1.r <a href="#">I2</a>			
				1		pmin1.u <a href="#">I2</a>	pmax1.u <a href="#">I2</a>			
				2		unpack1.l <a href="#">I2</a>	mix1.l <a href="#">I2</a>			
				3				psad1 <a href="#">I2</a>		
		3		0						
				1						
				2				mux1 <a href="#">I3</a>		
				3						

**Table 4-18. Multimedia Opcode 7 Size 2 2-bit Opcode Extensions**

Opcode Bits 40:37	z <sub>a</sub> Bit 36	z <sub>b</sub> Bit 33	v <sub>e</sub> Bit 32	x <sub>2a</sub> Bits 35:34	x <sub>2b</sub> Bits 29:28	x <sub>2c</sub> Bits 31:30						
						0	1	2	3			
7	0	1	0	0	0	0	psh2.u – var I5	pshl2 – var I7				
						1		pmpysh2.u I1				
						2	psh2 – var I5					
						3		pmpysh2 I1				
	1				1	0						
						1	psh2.u – fixed I6		popcnt I9			
						2			clz I9			
						3	psh2 – fixed I6					
	2				2	0	pack2.uss I2	unpack2.h I2	mix2.r I2			
						1						
						2	pack2.sss I2	unpack2.l I2	mix2.l I2			
						3	pmin2 I2	pmax2 I2	pmpy2.l I2			
	3				3	0						
						1		pshl2 – fixed I8				
						2			mux2 I4			
						3						

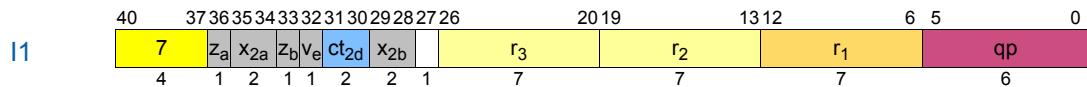
**Table 4-19. Multimedia Opcode 7 Size 4 2-bit Opcode Extensions**

Opcode Bits 40:37	z <sub>a</sub> Bit 36	z <sub>b</sub> Bit 33	v <sub>e</sub> Bit 32	x <sub>2a</sub> Bits 35:34	x <sub>2b</sub> Bits 29:28	x <sub>2c</sub> Bits 31:30						
						0	1	2	3			
7	1	0	0	0	0	0	psh4.u – var I5	pshl4 – var I7				
						1			mphy4 I2			
						2	psh4 – var I5					
						3			mphysh4 I2			
	1				1	0						
						1	psh4.u – fixed I6					
						2						
						3	psh4 – fixed I6					
	2				2	0		unpack4.h I2	mix4.r I2			
						1						
						2	pack4.sss I2	unpack4.l I2	mix4.l I2			
						3						
	3				3	0						
						1		pshl4 – fixed I8				
						2						
						3						

**Table 4-20. Variable Shift Opcode 7 2-bit Opcode Extensions**

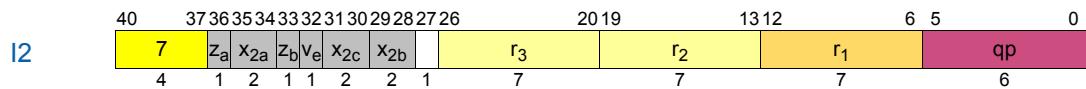
Opcode Bits 40:37	z <sub>a</sub> Bit 36	z <sub>b</sub> Bit 33	v <sub>e</sub> Bit 32	x <sub>2a</sub> Bits 35:34	x <sub>2b</sub> Bits 29:28	x <sub>2c</sub> Bits 31:30			
						0	1	2	3
7	1	1	0	0	0	shr.u - var  5	shl - var  7		
					1				
					2	shr - var  5			
					3				
				1	0				
					1				
					2				
					3				
				2	0				
					1				
					2				
					3				
				3	0				
					1				
					2				
					3				

#### 4.3.1.1 Multimedia Multiply and Shift



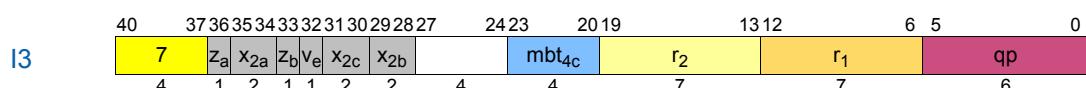
Instruction	Operands	Opcode	Extension				
			z <sub>a</sub>	z <sub>b</sub>	v <sub>e</sub>	x <sub>2a</sub>	x <sub>2b</sub>
pmpyshr2	r <sub>1</sub> = r <sub>2</sub> , r <sub>3</sub> , count <sub>2</sub>	7	0	1	0	0	3
pmpyshr2.u							1

#### 4.3.1.2 Multimedia Multiply/Mix/Pack/Unpack



Instruction	Operands	Opcode	Extension					
			$z_a$	$z_b$	$v_e$	$x_{2a}$	$x_{2b}$	$x_{2c}$
mpy4	$r_1 = r_2, r_3$	7	1	0	0	1	3	
mpyshl4			0	1		1	3	
pmpy2.r			0	0		0		
pmpy2.l			0	1		0		
mix1.r			1	0		2		
mix2.r			0	0		2		
mix4.r			0	1		0		
mix1.l			1	0		2		
mix2.l			0	0		0		
mix4.l			0	1		2		
pack2.uss			1	0		0		
pack2.sss			0	1		2		
pack4.sss			1	0		0		
unpack1.h			0	0		1		
unpack2.h			0	1		2		
unpack4.h			1	0		0		
unpack1.l			0	0		2		
unpack2.l			0	1		0		
unpack4.l			1	0		1		
pmin1.u			0	0		1	0	
pmax1.u			0	0		1	1	
pmin2			0	1		3	0	
pmax2			0	0		3	1	
psad1			0	0		3	2	

#### 4.3.1.3 Multimedia Mux1



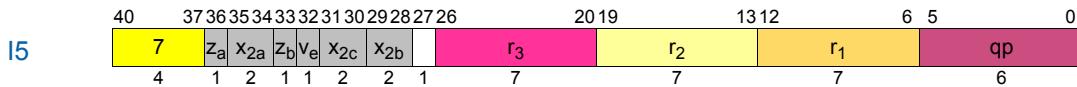
Instruction	Operands	Opcode	Extension					
			$z_a$	$z_b$	$v_e$	$x_{2a}$	$x_{2b}$	$x_{2c}$
mux1	$r_1 = r_2, mbtype_4$	7	0	0	0	3	2	2

#### 4.3.1.4 Multimedia Mux2



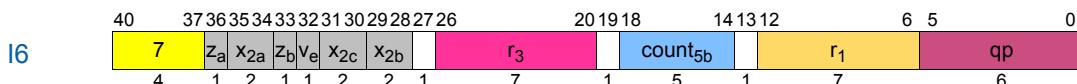
Instruction	Operands	Opcode	Extension					
			$z_a$	$z_b$	$v_e$	$x_{2a}$	$x_{2b}$	$x_{2c}$
mux2	$r_1 = r_2, mhtype_8$	7	0	1	0	3	2	2

#### 4.3.1.5 Shift Right – Variable



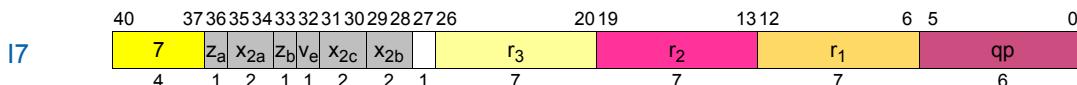
Instruction	Operands	Opcode	Extension					
			$z_a$	$z_b$	$v_e$	$x_{2a}$	$x_{2b}$	$x_{2c}$
pshr2	$r_1 = r_3, r_2$	7	0	1				
pshr4			1	0				
shr			1	1				
pshr2.u	$r_1 = r_3, r_2$	7	0	1	0	0	2	0
pshr4.u			1	0				
shr.u			1	1				

#### 4.3.1.6 Multimedia Shift Right – Fixed



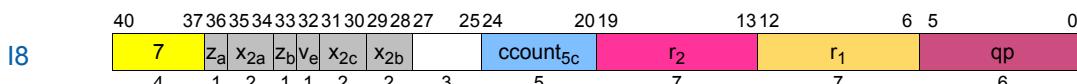
Instruction	Operands	Opcode	Extension					
			$z_a$	$z_b$	$v_e$	$x_{2a}$	$x_{2b}$	$x_{2c}$
pshr2	$r_1 = r_3, count_5$	7	0	1				
pshr4			1	0	0	1	3	0
pshr2.u			0	1				
pshr4.u			1	0			1	

#### 4.3.1.7 Shift Left – Variable



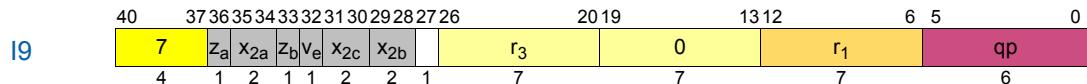
Instruction	Operands	Opcode	Extension					
			$z_a$	$z_b$	$v_e$	$x_{2a}$	$x_{2b}$	$x_{2c}$
pshl2	$r_1 = r_2, r_3$	7	0	1				
pshl4			1	0	0	0	0	1
shl			1	1				

#### 4.3.1.8 Multimedia Shift Left – Fixed



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				
pshl4			1	0	0	3	1	1

### 4.3.1.9 Bit Strings



Instruction	Operands	Opcode	Extension					
			<b><math>z_a</math></b>	<b><math>z_b</math></b>	<b><math>v_e</math></b>	<b><math>x_{2a}</math></b>	<b><math>x_{2b}</math></b>	<b><math>x_{2c}</math></b>
popcnt	$r_1 = r_3$	7	0	1	0	1	1	2
clz								3

### 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.

**Table 4-21. Integer Shift/Test Bit/Test NaT 2-bit Opcode Extensions**

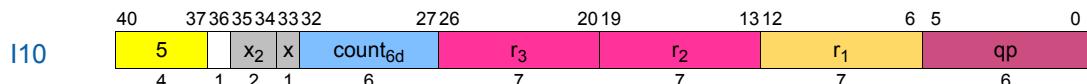
Opcode Bits 40:37	<b><math>x_2</math></b> Bits 35:34	<b><math>x</math></b> Bit 33	<b><math>y</math></b> Bit 13	
			0	1
5	0	0	Test Bit (Table 4-23)	Test NaT/Test Feature (Table 4-23)
	1		extr.u I11	extr I11
	2			
	3		shrp I10	

Most deposit instructions also have a 1-bit opcode extension field in bit 26 ( $y$ ). Table 4-22 shows these assignments.

**Table 4-22. Deposit Opcode Extensions**

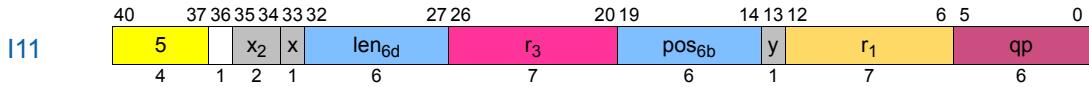
Opcode Bits 40:37	<b><math>x_2</math></b> Bits 35:34	<b><math>x</math></b> Bit 33	<b><math>y</math></b> Bit 26	
			0	1
5	0	1	Test Bit/Test NaT/Test Feature (Table 4-23)	
	1		dep.z I12	dep.z – imm <sub>8</sub> I13
	2			
	3		dep – imm <sub>1</sub> I14	

### 4.3.2.1 Shift Right Pair



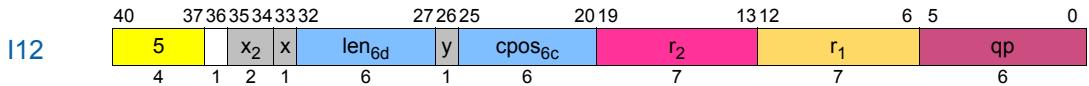
Instruction	Operands	Opcode	Extension	
			<b><math>x_2</math></b>	<b><math>x</math></b>
shrp	$r_1 = r_2, r_3, count_6$	5	3	0

#### 4.3.2.2 Extract



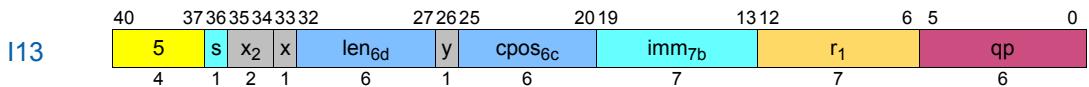
Instruction	Operands	Opcode	Extension		
			<b>x<sub>2</sub></b>	<b>x</b>	<b>y</b>
extr.u			1	0	0
extr				1	1

#### 4.3.2.3 Zero and Deposit



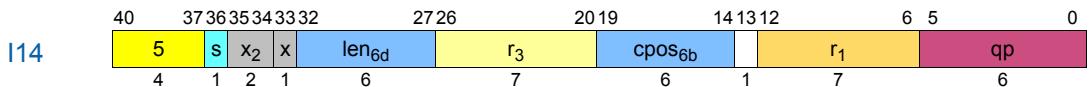
Instruction	Operands	Opcode	Extension		
			<b>x<sub>2</sub></b>	<b>x</b>	<b>y</b>
dep.z	$r_1 = r_2, pos_6, len_6$	5	1	1	0

#### 4.3.2.4 Zero and Deposit Immediate<sub>8</sub>



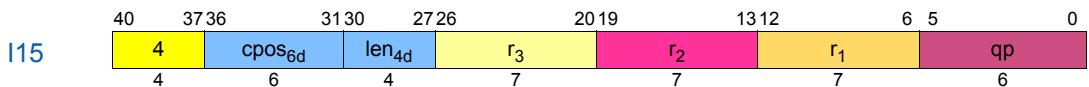
Instruction	Operands	Opcode	Extension		
			<b>x<sub>2</sub></b>	<b>x</b>	<b>y</b>
dep.z	$r_1 = imm_8, pos_6, len_6$	5	1	1	1

#### 4.3.2.5 Deposit Immediate<sub>1</sub>



Instruction	Operands	Opcode	Extension	
			<b>x<sub>2</sub></b>	<b>x</b>
dep	$r_1 = imm_1, r_3, pos_6, len_6$	5	3	1

#### 4.3.2.6 Deposit



Instruction	Operands	Opcode
dep	$r_1 = r_2, r_3, pos_6, len_4$	4

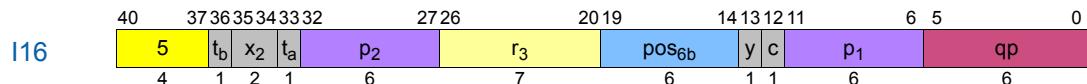
#### 4.3.3 Test Bit

All test bit instructions are encoded within major opcode 5 using a 2-bit opcode extension field in bits 35:34 ( $x_2$ ) plus five 1-bit opcode extension fields in bits 33 ( $t_a$ ), 36 ( $t_b$ ), 12 ( $c$ ), 13 ( $y$ ) and 19 ( $x$ ). Table 4-23 summarizes these assignments.

**Table 4-23. Test Bit Opcode Extensions**

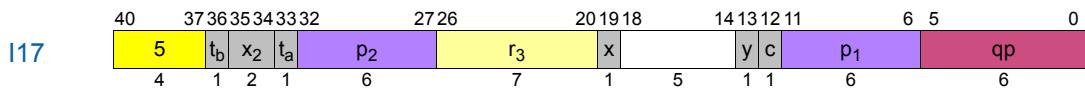
Opcode Bits 40:37	x <sub>2</sub> Bits 35:34	t <sub>a</sub> Bit 33	t <sub>b</sub> Bit 36	c Bit 12	y Bit 13	x Bit 19	
						0	1
5	0	0	0	0	0	tbit.z I16	
				1	1	tnat.z I17	tf.z I30
			1	0	0	tbit.z.unc I16	
				1	1	tnat.z.unc I17	tf.z.unc I30
		1	0	0	0	tbit.z.and I16	
				1	1	tnat.z.and I17	tf.z.and I30
			1	0	0	tbit.nz.and I16	
				1	1	tnat.nz.and I17	tf.nz.and I30
		1	0	0	0	tbit.z.or I16	
				1	1	tnat.z.or I17	tf.z.or I30
			1	0	0	tbit.nz.or I16	
				1	1	tnat.nz.or I17	tf.nz.or I30
			0	0	0	tbit.z.or.andcm I16	
				1	1	tnat.z.or.andcm I17	tf.z.or.andcm I30
			1	0	0	tbit.nz.or.andcm I16	
				1	1	tnat.nz.or.andcm I17	tf.nz.or.andcm I30

#### 4.3.3.1 Test Bit



Instruction	Operands	Opcode	Extension				
			x <sub>2</sub>	t <sub>a</sub>	t <sub>b</sub>	y	c
tbit.z				0	0	0	0
tbit.z.unc				0	1	1	1
tbit.z.and				1	0	0	0
tbit.nz.and				1	1	1	1
tbit.z.or				0	0	0	0
tbit.nz.or				1	1	1	1
tbit.z.or.andcm				0	0	0	0
tbit.nz.or.andcm				1	1	1	1

### 4.3.3.2 Test NaT



Instruction	Operands	Opcode	Extension							
			x <sub>2</sub>	t <sub>a</sub>	t <sub>b</sub>	y	x	c		
tnat.z	$p_1, p_2 = r_3$	5	0	0	0	1	0	0		
tnat.z.unc					1			1		
tnat.z.and			1	0	0			0		
tnat.nz.and								1		
tnat.z.or			0	1	1	0	0	0		
tnat.nz.or								1		
tnat.z.or.andcm			1	0	1	0	0	0		
tnat.nz.or.andcm								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 <sub>3</sub> Bits 35:33	
0	0	6-bit Ext (Table 4-25)
	1	chk.s.i – int I20
	2	mov to pr.rot – imm <sub>44</sub> 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	1	2
0	0	0	break.i  29	zxt1  29	
		1	1-bit Ext (Table 4-26)	zxt2  29	
		2		zxt4  29	
		3			mov from pr  25
		4		sxt1  29	
		5		sxt2  29	
		6		sxt4  29	
		7			
		8		czx1.i  29	
		9		czx2.i  29	
		A	mov.i to ar – imm <sub>8</sub>  27		mov.i to ar  26
		B			
		C		czx1.r  29	
		D		czx2.r  29	
		E			
		F			

#### 4.3.4.1 Nop/Hint (I-Unit)

I-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 ( $x_3$ ), a 6-bit opcode extension field in bits 32:27 ( $x_6$ ), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-26.

**Table 4-26. Misc I-Unit 1-bit Opcode Extensions**

Opcode Bits 40:37	$x_3$ Bits 35:33	$x_6$ Bits 32:27	y Bit 26	
0	0	01	0	nop.i
			1	hint.i
40  18	37 0	36 i	35 $x_3$	33 $x_6$
32 4	32 1	32 3	27 6	26 1
25 20	25 20	25 20	25 20	25 20
6 5 0	6 5 0	6 5 0	6 5 0	6 5 0
Instruction	Operands	Opcode	Extension	
			$x_3$	$x_6$
nop.i <sup>1</sup>	$imm_{21}$	0	0	01
hint.i				0
				1

#### 4.3.4.2 Break (I-Unit)

I19	40 373635 3332 272625	imm <sub>20a</sub>	6 5 0
	0 1 x <sub>3</sub> 6 1 20	x <sub>6</sub>	qp 6

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
break.i <sup>i</sup>	imm <sub>21</sub>	0	0	00

#### 4.3.4.3 Integer Speculation Check (I-Unit)

I20	40 373635 3332 2019 1312 6 5 0	imm <sub>13c</sub>	r <sub>2</sub>	imm <sub>7a</sub>	qp
	0 1 s x <sub>3</sub> 13 7 7 6	imm <sub>13c</sub>	r <sub>2</sub>	imm <sub>7a</sub>	qp

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	
chk.s.i	r <sub>2</sub> , target <sub>25</sub>	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 3:318 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 4-27.

**Table 4-27. 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 4-56 on page 3:354](#)).

#### 4.3.5.1 Move to BR

I21	40 373635 3332 242322212019 1312 9 8 6 5 0	timm <sub>9c</sub>	ih x wh	r <sub>2</sub>	b <sub>1</sub>	qp
	0 1 x <sub>3</sub> 9 1 1 2 7 4 3 6	timm <sub>9c</sub>	ih x wh	r <sub>2</sub>	b <sub>1</sub>	qp

Instruction	Operands	Opcode	Extension			
			x <sub>3</sub>	x	ih	wh
mov.mwh.ih	b <sub>1</sub> = r <sub>2</sub> , tag <sub>13</sub>	0	7	0	See Table 4-56 on page 3:354	See Table 4-27 on page 3:320
mov.ret.mwh.ih				1		

#### 4.3.5.2 Move from BR

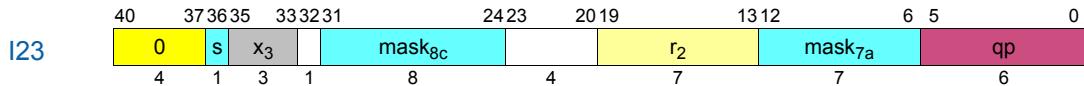


Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
mov	r <sub>1</sub> = b <sub>2</sub>	0	0	31

#### 4.3.6 GR/Predicate/IP Moves

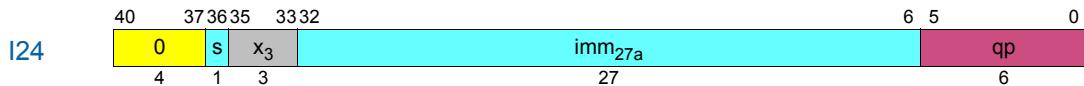
The GR/Predicate/IP move instructions are encoded in major opcode 0. See “Miscellaneous I-Unit Instructions” on page 3:318 for a summary of the opcode extensions.

##### 4.3.6.1 Move to Predicates – Register



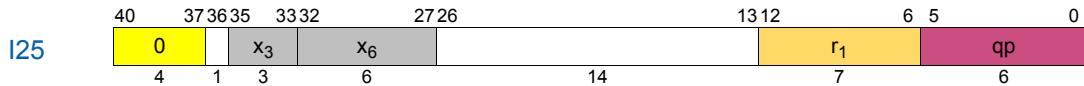
Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	
mov	pr = r <sub>2</sub> , mask <sub>17</sub>	0		3

##### 4.3.6.2 Move to Predicates – Immediate<sub>44</sub>



Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	
mov	pr.rot = imm <sub>44</sub>	0		2

##### 4.3.6.3 Move from Predicates/IP

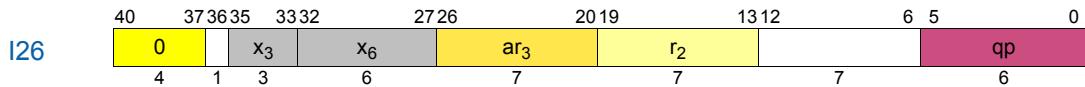


Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
mov	r <sub>1</sub> = ip r <sub>1</sub> = pr	0	0	30 33

#### 4.3.7 GR/AR Moves (I-Unit)

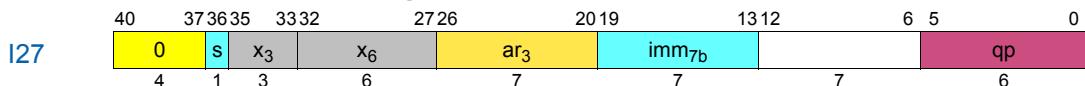
The I-Unit GR/AR move instructions are encoded in major opcode 0. (Some ARs are accessed using system/memory management instructions on the M-unit. See “GR/AR Moves (M-Unit)” on page 3:342.) See “Miscellaneous I-Unit Instructions” on page 3:318 for a summary of the I-Unit GR/AR opcode extensions.

#### 4.3.7.1 Move to AR – Register (I-Unit)



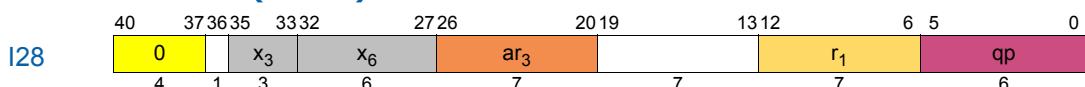
Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
mov.i	ar <sub>3</sub> = r <sub>2</sub>	0	0	2A

#### 4.3.7.2 Move to AR – Immediate<sub>8</sub> (I-Unit)



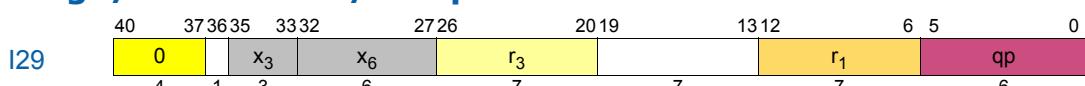
Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
mov.i	ar <sub>3</sub> = imm <sub>8</sub>	0	0	0A

#### 4.3.7.3 Move from AR (I-Unit)



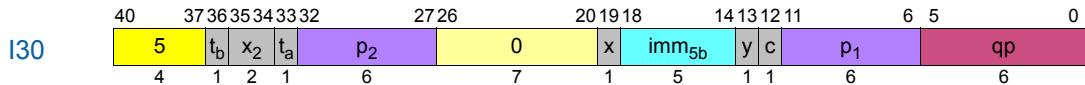
Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
mov.i	r <sub>1</sub> = ar <sub>3</sub>	0	0	32

#### 4.3.8 Sign/Zero Extend/Compute Zero Index



Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
zxt1				10
zxt2				11
zxt4				12
sxt1				14
sxt2				15
sxt4				16
czx1.l				18
czx2.l				19
czx1.r				1C
czx2.r				1D

### 4.3.9 Test Feature



Instruction	Operands	Opcode	Extension					
			x <sub>2</sub>	t <sub>a</sub>	t <sub>b</sub>	y	x	c
tf.z	$p_1, p_2 = imm_5$	5	0	0	0	1	1	0
tf.z.unc					1			1
tf.z.and			1	0	0	1	1	0
tf.nz.and								1
tf.z.or			1	1	1	0	0	0
tf.nz.or								1
tf.z.or.andcm								0
tf.nz.or.andcm								1

## 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<sub>6</sub>). 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 4-28](#). 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 4-29](#).

**Table 4-28. 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 ( <a href="#">Table 4-30</a> )
	0	1	Semaphore/get FR ( <a href="#">Table 4-33</a> )
	1	0	Load +Reg ( <a href="#">Table 4-31</a> )
	1	1	

**Table 4-29. 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 ( <a href="#">Table 4-34</a> )
	0	1	FP Load Pair/set FR ( <a href="#">Table 4-37</a> )
	1	0	FP Load +Reg ( <a href="#">Table 4-35</a> )
	1	1	FP Load Pair +Imm ( <a href="#">Table 4-38</a> )

The integer load/store opcode extensions are summarized in [Table 4-30 on page 3:324](#), [Table 4-31 on page 3:324](#), and [Table 4-32 on page 3:325](#), and the semaphore and get FR opcode extensions in [Table 4-33 on page 3:325](#). The floating-point load/store

opcode extensions are summarized in [Table 4-34 on page 3:326](#), [Table 4-35 on page 3:326](#), and [Table 4-36 on page 3:327](#), the floating-point load pair and set FR opcode extensions in [Table 4-37 on page 3:327](#) and [Table 4-38 on page 3:328](#).

**Table 4-30. Integer Load/Store Opcode Extensions**

Opcode Bits 40:37	m Bit 36	x Bit 27	x <sub>6</sub>				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	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	st1 M6	st2 M6	st4 M6	st8 M6
			D	st1.rel M6	st2.rel M6	st4.rel M6	st8.rel M6
			E				st8.spill M6
			F				

**Table 4-31. Integer Load +Reg Opcode Extensions**

Opcode Bits 40:37	m Bit 36	x Bit 27	x <sub>6</sub>				
			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 4-32. Integer Load/Store +Imm Opcode Extensions**

Opcode Bits 40:37	x <sub>6</sub>				
	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 4-33. Semaphore/Get FR/16-Byte Opcode Extensions**

Opcode Bits 40:37	m Bit 36	x Bit 27	x <sub>6</sub>				
			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	cmp8xchg16.acq M16			
			9	cmp8xchg16.rel M16			
			A	ld16 M2			
			B	ld16.acq M2			
			C	st16 M6			
			D	st16.rel M6			
			E				
			F				

**Table 4-34. Floating-point Load/Store/Lfetch Opcode Extensions**

Opcode Bits 40:37	m Bit 36	x Bit 27	x <sub>6</sub>				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	0	0	0	ldfe M9	ldf8 M9	ldfs M9	ldfd M9
			1	ldfe.s M9	ldf8.s M9	ldfs.s M9	ldfd.s M9
			2	ldfe.a M9	ldf8.a M9	ldfs.a M9	ldfd.a M9
			3	ldfe.sa M9	ldf8.sa M9	ldfs.sa M9	ldfd.sa M9
			4				
			5				
			6				ldf.fill M9
			7				
			8	ldfe.c.clr M9	ldf8.c.clr M9	ldfs.c.clr M9	ldfd.c.clr M9
			9	ldfe.c.nc M9	ldf8.c.nc M9	ldfs.c.nc M9	ldfd.c.nc M9
			A				
			B	lfetch M18	lfetch.excl M18	lfetch.fault M18	lfetch.fault.excl M18
			C	stfe M13	stf8 M13	stfs M13	stfd M13
			D				
			E				stf.spill M13
			F				

**Table 4-35. Floating-point Load/Lfetch +Reg Opcode Extensions**

Opcode Bits 40:37	m Bit 36	x Bit 27	x <sub>6</sub>				
			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 M20	lfetch.excl M20	lfetch.fault M20	lfetch.fault.excl M20
			C				
			D				
			E				
			F				

**Table 4-36. Floating-point Load/Store/Lfetch +Imm Opcode Extensions**

Opcode Bits 40:37	x <sub>6</sub>				
	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 M22	lfetch.excl M22	lfetch.fault M22	lfetch.fault.excl M22
	C	stfe M10	stf8 M10	stfs M10	stfd M10
	D				
	E				stf.spill M10
	F				

**Table 4-37. Floating-point Load Pair/Set FR Opcode Extensions**

Opcode Bits 40:37	m Bit 36	x Bit 27	x <sub>6</sub>				
			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 4-38. Floating-point Load Pair +Imm Opcode Extensions**

Opcode Bits 40:37	m Bit 36	x Bit 27	Bits 35:32	x <sub>6</sub>			
				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 cache locality opcode hint extension field in bits 29:28 (hint). Table 4-39 and Table 4-40 summarize these assignments.

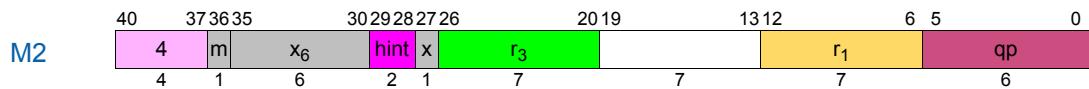
**Table 4-39. Load Hint Completer**

hint Bits 29:28	ldhint
0	none
1	.nt1
2	
3	.nta

**Table 4-40. Store Hint Completer**

hint Bits 29:28	sthint
0	none
1	
2	
3	.nta

#### 4.4.1.1 Integer Load



Instruction	Operands	Opcode	Extension			
			m	x	x <sub>6</sub>	hint
ld1.lhint					00	
ld2.lhint					01	
ld4.lhint					02	
ld8.lhint					03	
ld1.s.lhint					04	
ld2.s.lhint					05	
ld4.s.lhint					06	
ld8.s.lhint					07	
ld1.a.lhint					08	
ld2.a.lhint					09	
ld4.a.lhint					0A	
ld8.a.lhint					0B	
ld1.sa.lhint					0C	
ld2.sa.lhint					0D	
ld4.sa.lhint					0E	
ld8.sa.lhint					0F	
ld1.bias.lhint	r <sub>1</sub> = [r <sub>3</sub> ]	4	0	0	10	See Table 4-39 on page 3:328
ld2.bias.lhint					11	
ld4.bias.lhint					12	
ld8.bias.lhint					13	
ld1.acq.lhint					14	
ld2.acq.lhint					15	
ld4.acq.lhint					16	
ld8.acq.lhint					17	
ld8.fill.lhint					1B	
ld1.c.clr.lhint					20	
ld2.c.clr.lhint					21	
ld4.c.clr.lhint					22	
ld8.c.clr.lhint					23	
ld1.c.nc.lhint					24	
ld2.c.nc.lhint					25	
ld4.c.nc.lhint					26	
ld8.c.nc.lhint					27	
ld1.c.clr.acq.lhint					28	
ld2.c.clr.acq.lhint					29	
ld4.c.clr.acq.lhint					2A	
ld8.c.clr.acq.lhint					2B	
ld16.lhint	r <sub>1</sub> , ar.cs = [r <sub>3</sub> ]	0	1		28	
ld16.acq.lhint					2C	

#### 4.4.1.2 Integer Load – Increment by Register



Instruction	Operands	Opcode	Extension			
			m	x	x <sub>6</sub>	hint
ld1.Idhint					00	
ld2.Idhint					01	
ld4.Idhint					02	
ld8.Idhint					03	
ld1.s.Idhint					04	
ld2.s.Idhint					05	
ld4.s.Idhint					06	
ld8.s.Idhint					07	
ld1.a.Idhint					08	
ld2.a.Idhint					09	
ld4.a.Idhint					0A	
ld8.a.Idhint					0B	
ld1.sa.Idhint					0C	
ld2.sa.Idhint					0D	
ld4.sa.Idhint					0E	
ld8.sa.Idhint					0F	
ld1.bias.Idhint	r <sub>1</sub> = [r <sub>3</sub> ], r <sub>2</sub>	4	1	0	10	See Table 4-39 on page 3:328
ld2.bias.Idhint					11	
ld4.bias.Idhint					12	
ld8.bias.Idhint					13	
ld1.acq.Idhint					14	
ld2.acq.Idhint					15	
ld4.acq.Idhint					16	
ld8.acq.Idhint					17	
ld8.fill.Idhint					1B	
ld1.c.clr.Idhint					20	
ld2.c.clr.Idhint					21	
ld4.c.clr.Idhint					22	
ld8.c.clr.Idhint					23	
ld1.c.nc.Idhint					24	
ld2.c.nc.Idhint					25	
ld4.c.nc.Idhint					26	
ld8.c.nc.Idhint					27	
ld1.c.clr.acq.Idhint					28	
ld2.c.clr.acq.Idhint					29	
ld4.c.clr.acq.Idhint					2A	
ld8.c.clr.acq.Idhint					2B	

#### 4.4.1.3 Integer Load – Increment by Immediate



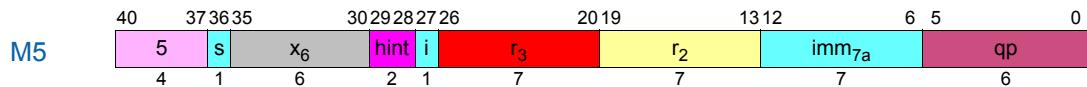
Instruction	Operands	Opcode	Extension	
			x <sub>6</sub>	hint
ld1. <i>ldhint</i>			00	
ld2. <i>ldhint</i>			01	
ld4. <i>ldhint</i>			02	
ld8. <i>ldhint</i>			03	
ld1.s. <i>ldhint</i>			04	
ld2.s. <i>ldhint</i>			05	
ld4.s. <i>ldhint</i>			06	
ld8.s. <i>ldhint</i>			07	
ld1.a. <i>ldhint</i>			08	
ld2.a. <i>ldhint</i>			09	
ld4.a. <i>ldhint</i>			0A	
ld8.a. <i>ldhint</i>			0B	
ld1.sa. <i>ldhint</i>			0C	
ld2.sa. <i>ldhint</i>			0D	
ld4.sa. <i>ldhint</i>			0E	
ld8.sa. <i>ldhint</i>			0F	
ld1.bias. <i>ldhint</i>	$r_1 = [r_3], imm_9$	5	10	See Table 4-39 on page 3:328
ld2.bias. <i>ldhint</i>			11	
ld4.bias. <i>ldhint</i>			12	
ld8.bias. <i>ldhint</i>			13	
ld1.acq. <i>ldhint</i>			14	
ld2.acq. <i>ldhint</i>			15	
ld4.acq. <i>ldhint</i>			16	
ld8.acq. <i>ldhint</i>			17	
ld8.fill. <i>ldhint</i>			1B	
ld1.c.clr. <i>ldhint</i>			20	
ld2.c.clr. <i>ldhint</i>			21	
ld4.c.clr. <i>ldhint</i>			22	
ld8.c.clr. <i>ldhint</i>			23	
ld1.c.nc. <i>ldhint</i>			24	
ld2.c.nc. <i>ldhint</i>			25	
ld4.c.nc. <i>ldhint</i>			26	
ld8.c.nc. <i>ldhint</i>			27	
ld1.c.clr.acq. <i>ldhint</i>			28	
ld2.c.clr.acq. <i>ldhint</i>			29	
ld4.c.clr.acq. <i>ldhint</i>			2A	
ld8.c.clr.acq. <i>ldhint</i>			2B	

#### 4.4.1.4 Integer Store



Instruction	Operands	Opcode	Extension			
			<b>m</b>	<b>x</b>	<b>x<sub>6</sub></b>	<b>hint</b>
st1.sthint					30	
st2.sthint					31	
st4.sthint					32	
st8.sthint					33	
st1.rel.sthint	[r <sub>3</sub> ] = r <sub>2</sub>	4	0	0	34	See Table 4-40 on page 3:328
st2.rel.sthint					35	
st4.rel.sthint					36	
st8.rel.sthint					37	
st8.spill.sthint					3B	
st16.sthint	[r <sub>3</sub> ] = r <sub>2</sub> , ar.csd		0	1	30	
st16.rel.sthint					34	

#### 4.4.1.5 Integer Store – Increment by Immediate



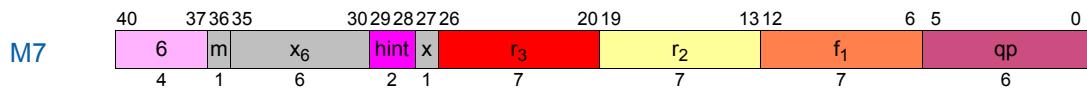
Instruction	Operands	Opcode	Extension	
			<b>x<sub>6</sub></b>	<b>hint</b>
st1.sthint			30	
st2.sthint			31	
st4.sthint			32	
st8.sthint			33	
st1.rel.sthint	[r <sub>3</sub> ] = r <sub>2</sub> , imm <sub>9</sub>	5	34	See Table 4-40 on page 3:328
st2.rel.sthint			35	
st4.rel.sthint			36	
st8.rel.sthint			37	
st8.spill.sthint			3B	

#### 4.4.1.6 Floating-point Load



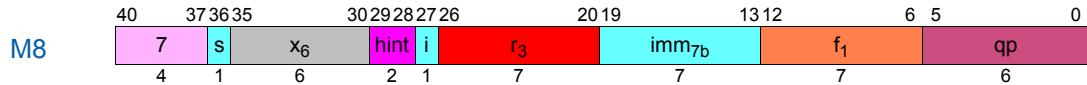
Instruction	Operands	Opcode	Extension			
			m	x	x <sub>6</sub>	hint
ldfs. <i>dhint</i>					02	
ldfd. <i>dhint</i>					03	
ldf8. <i>dhint</i>					01	
ldfe. <i>dhint</i>					00	
ldfs.s. <i>dhint</i>					06	
ldfd.s. <i>dhint</i>					07	
ldf8.s. <i>dhint</i>					05	
ldfe.s. <i>dhint</i>					04	
ldfs.a. <i>dhint</i>					0A	
ldfd.a. <i>dhint</i>					0B	
ldf8.a. <i>dhint</i>					09	
ldfe.a. <i>dhint</i>					08	
ldfs.sa. <i>dhint</i>	f <sub>1</sub> = [r <sub>3</sub> ]	6	0	0	0E	See Table 4-39 on page 3:328
ldfd.sa. <i>dhint</i>					0F	
ldf8.sa. <i>dhint</i>					0D	
ldfe.sa. <i>dhint</i>					0C	
ldf.fill. <i>dhint</i>					1B	
ldfs.c.clr. <i>dhint</i>					22	
ldfd.c.clr. <i>dhint</i>					23	
ldf8.c.clr. <i>dhint</i>					21	
ldfe.c.clr. <i>dhint</i>					20	
ldfs.c.nc. <i>dhint</i>					26	
ldfd.c.nc. <i>dhint</i>					27	
ldf8.c.nc. <i>dhint</i>					25	
ldfe.c.nc. <i>dhint</i>					24	

#### 4.4.1.7 Floating-point Load – Increment by Register



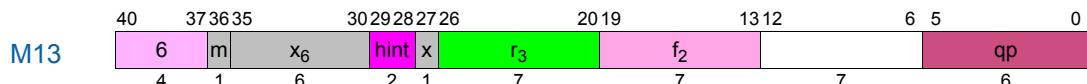
Instruction	Operands	Opcode	Extension			
			m	x	x <sub>6</sub>	hint
ldfs.ldhint					02	
ldfd.ldhint					03	
ldf8.ldhint					01	
ldfe.ldhint					00	
ldfs.s.ldhint					06	
ldfd.s.ldhint					07	
ldf8.s.ldhint					05	
ldfe.s.ldhint					04	
ldfs.a.ldhint					0A	
ldfd.a.ldhint					0B	
ldf8.a.ldhint					09	
ldfe.a.ldhint					08	
ldfs.sa.ldhint	$f_1 = [r_3], r_2$	6	1	0	0E	See Table 4-39 on page 3:328
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



Instruction	Operands	Opcode	Extension	
			x <sub>6</sub>	hint
ldfs. <i>ldhint</i>			02	
ldfd. <i>ldhint</i>			03	
ldf8. <i>ldhint</i>			01	
ldfe. <i>ldhint</i>			00	
ldfs.s. <i>ldhint</i>			06	
ldfd.s. <i>ldhint</i>			07	
ldf8.s. <i>ldhint</i>			05	
ldfe.s. <i>ldhint</i>			04	
ldfs.a. <i>ldhint</i>			0A	
ldfd.a. <i>ldhint</i>			0B	
ldf8.a. <i>ldhint</i>			09	
ldfe.a. <i>ldhint</i>			08	
ldfs.sa. <i>ldhint</i>	f <sub>1</sub> = [r <sub>3</sub> ], imm <sub>9</sub>	7	0E	See Table 4-39 on page 3:328
ldfd.sa. <i>ldhint</i>			0F	
ldf8.sa. <i>ldhint</i>			0D	
ldfe.sa. <i>ldhint</i>			0C	
ldf.fill. <i>ldhint</i>			1B	
ldfs.c.clr. <i>ldhint</i>			22	
ldfd.c.clr. <i>ldhint</i>			23	
ldf8.c.clr. <i>ldhint</i>			21	
ldfe.c.clr. <i>ldhint</i>			20	
ldfs.c.nc. <i>ldhint</i>			26	
ldfd.c.nc. <i>ldhint</i>			27	
ldf8.c.nc. <i>ldhint</i>			25	
ldfe.c.nc. <i>ldhint</i>			24	

#### 4.4.1.9 Floating-point Store



Instruction	Operands	Opcode	Extension			
			m	x	x <sub>6</sub>	hint
stfs. <i>sthint</i>			32			
stfd. <i>sthint</i>			33			
stf8. <i>sthint</i>	[r <sub>3</sub> ] = f <sub>2</sub>	6	31			
stfe. <i>sthint</i>			30			
stf.spill. <i>sthint</i>			3B			
						See Table 4-40 on page 3:328

#### 4.4.1.10 Floating-point Store – Increment by Immediate

M10	40	37 36 35	30 29 28 27 26	20 19	13 12	6 5	0	
	7	s	x <sub>6</sub>	hint i	r <sub>3</sub>	f <sub>2</sub>	imm <sub>7a</sub>	qp
	4	1	6	2 1	7	7	7	6

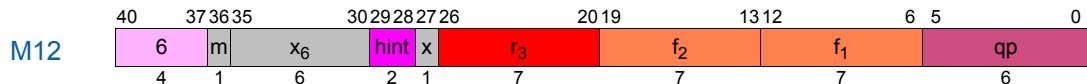
Instruction	Operands	Opcode	Extension	
			x <sub>6</sub>	hint
stfs.sthint	[r <sub>3</sub> ] = f <sub>2</sub> , imm <sub>9</sub>	7	32	See Table 4-40 on page 3:328
stfd.sthint			33	
stf8.sthint			31	
stfe.sthint			30	
stf.spill.sthint			3B	

#### 4.4.1.11 Floating-point Load Pair

M11	40	37 36 35	30 29 28 27 26	20 19	13 12	6 5	0	
	6	m	x <sub>6</sub>	hint x	r <sub>3</sub>	f <sub>2</sub>	f <sub>1</sub>	qp
	4	1	6	2 1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x <sub>6</sub>	hint
ldfps.ldhint	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ]	6	02	0	1	See Table 4-39 on page 3:328
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 <sub>6</sub>	hint
ldfps. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 8	6	1	1	02	See Table 4-39 on page 3:328
ldfpd. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				03	
ldfp8. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				01	
ldfps.s. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 8				06	
ldfpd.s. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				07	
ldfp8.s. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				05	
ldfps.a. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 8				0A	
ldfpd.a. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				0B	
ldfp8.a. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				09	
ldfps.sa. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 8				0E	
ldfpd.sa. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				0F	
ldfp8.sa. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				0D	
ldfps.c.clr. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 8				22	
ldfpd.c.clr. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				23	
ldfp8.c.clr. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				21	
ldfps.c.nc. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 8				26	
ldfpd.c.nc. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				27	
ldfp8.c.nc. <i>ldhint</i>	f <sub>1</sub> , f <sub>2</sub> = [r <sub>3</sub> ], 16				25	

## 4.4.2 Line Prefetch

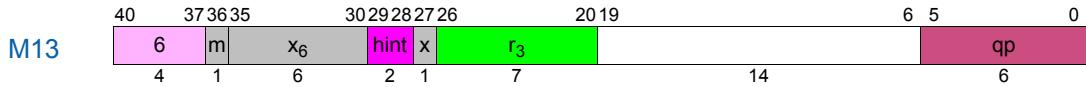
The line prefetch instructions are encoded in major opcodes 6 and 7 along with the floating-point load/store instructions. See “Loads and Stores” on page 3:323 for a summary of the opcode extensions.

The line prefetch instructions all have a 2-bit cache locality opcode hint extension field in bits 29:28 (hint) as shown in Table 4-44.

**Table 4-41. Line Prefetch Hint Completer**

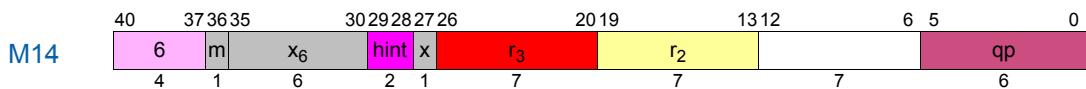
hint Bits 29:28	lfhint
0	none
1	.nt1
2	.nt2
3	.nta

#### 4.4.2.1 Line Prefetch



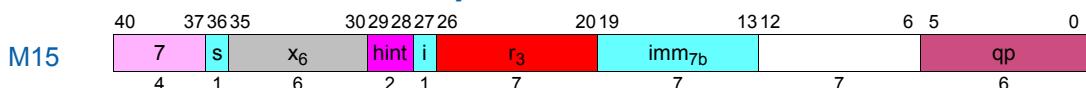
Instruction	Operands	Opcode	Extension			
			m	x	$x_6$	hint
Ifetch.excl.Ifhint					2D	
Ifetch.fault.Ifhint	[ $r_3$ ]	6	0	0	2E	See Table 4-41 on page 3:337
Ifetch.fault.excl.Ifhint					2F	

#### 4.4.2.2 Line Prefetch – Increment by Register



Instruction	Operands	Opcode	Extension			
			m	x	$x_6$	hint
Ifetch.Ifhint					2C	
Ifetch.excl.Ifhint	[ $r_3$ , $r_2$ ]	6	1	0	2D	See Table 4-41 on page 3:337
Ifetch.fault.Ifhint					2E	
Ifetch.fault.excl.Ifhint					2F	

#### 4.4.2.3 Line Prefetch – Increment by Immediate

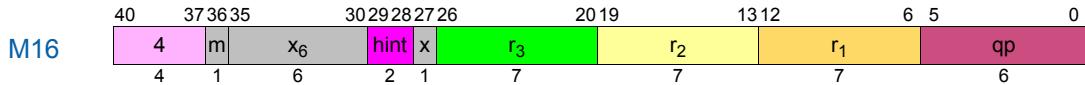


Instruction	Operands	Opcode	Extension	
			$x_6$	hint
Ifetch.Ifhint			2C	
Ifetch.excl.Ifhint	[ $r_3$ , imm <sub>9</sub> ]	7	2D	See Table 4-41 on page 3:337
Ifetch.fault.Ifhint			2E	
Ifetch.fault.excl.Ifhint			2F	

## 4.4.3 Semaphores

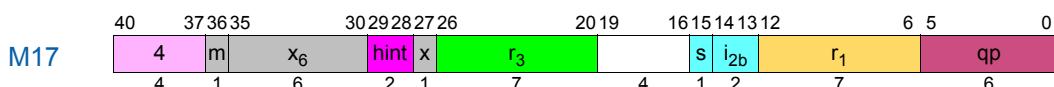
The semaphore instructions are encoded in major opcode 4 along with the integer load/store instructions. See “[Loads and Stores](#)” on page 3:323 for a summary of the opcode extensions. These instructions have the same cache locality opcode hint extension field in bits 29:28 (hint) as load instructions. See [Table 4-39, “Load Hint Completer”](#) on page 3:328.

#### 4.4.3.1 Exchange/Compare and Exchange



Instruction	Operands	Opcode	Extension			
			m	x	x <sub>6</sub>	hint
cmpxchg1.acq. <i>ldhint</i>	<i>r</i> <sub>1</sub> = [r <sub>3</sub> ], <i>r</i> <sub>2</sub> , ar.ccv	4	0	1	00	See Table 4-39 on page 3:328
cmpxchg2.acq. <i>ldhint</i>					01	
cmpxchg4.acq. <i>ldhint</i>					02	
cmpxchg8.acq. <i>ldhint</i>					03	
cmpxchg1.rel. <i>ldhint</i>					04	
cmpxchg2.rel. <i>ldhint</i>					05	
cmpxchg4.rel. <i>ldhint</i>					06	
cmpxchg8.rel. <i>ldhint</i>					07	
cmp8xchg16.acq. <i>ldhint</i>	<i>r</i> <sub>1</sub> = [r <sub>3</sub> ], <i>r</i> <sub>2</sub> , ar.csd, ar.ccv	4	0	1	20	See Table 4-39 on page 3:328
cmp8xchg16.rel. <i>ldhint</i>					24	
xchg1. <i>ldhint</i>	<i>r</i> <sub>1</sub> = [r <sub>3</sub> ], <i>r</i> <sub>2</sub>	4	0	1	08	See Table 4-39 on page 3:328
xchg2. <i>ldhint</i>					09	
xchg4. <i>ldhint</i>					0A	
xchg8. <i>ldhint</i>					0B	

#### 4.4.3.2 Fetch and Add – Immediate



Instruction	Operands	Opcode	Extension			
			m	x	x <sub>6</sub>	hint
fetchadd4.acq. <i>ldhint</i>	<i>r</i> <sub>1</sub> = [r <sub>3</sub> ], inc <sub>3</sub>	4	0	1	12	See Table 4-39 on page 3:328
fetchadd8.acq. <i>ldhint</i>					13	
fetchadd4.rel. <i>ldhint</i>					16	
fetchadd8.rel. <i>ldhint</i>					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 3:323 for a summary of the opcode extensions.

#### 4.4.4.1 Set FR

M18	40 373635 3029282726	2019	1312	6 5	0
	6 m x <sub>6</sub> x	r <sub>2</sub>	f <sub>1</sub>	qp	

Instruction	Operands	Opcode	Extension		
			m	x	x <sub>6</sub>
setf.sig					1C
setf.exp					1D
setf.s					1E
setf.d					1F

#### 4.4.4.2 Get FR

M19	40 373635 3029282726	2019	1312	6 5	0
	4 m x <sub>6</sub> x r <sub>2</sub> r <sub>1</sub> qp				

Instruction	Operands	Opcode	Extension		
			m	x	x <sub>6</sub>
getf.sig					1C
getf.exp					1D
getf.s					1E
getf.d	r <sub>1</sub> = f <sub>2</sub>	4	0	1	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 3:345 for a summary of the opcode extensions.

#### 4.4.5.1 Integer Speculation Check (M-Unit)

M20	40 373635 3332	2019	1312	6 5	0
	1 s x <sub>3</sub> imm <sub>13c</sub> r <sub>2</sub> imm <sub>7a</sub> qp				

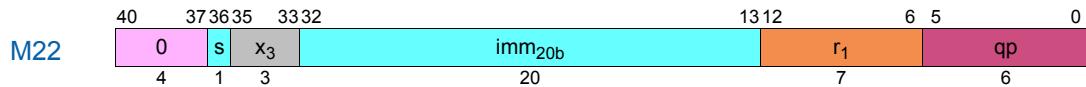
Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	
chk.s.m	r <sub>2</sub> , target <sub>25</sub>	1	1	

#### 4.4.5.2 Floating-point Speculation Check

M21	40 373635 3332	2019	1312	6 5	0
	1 s x <sub>3</sub> imm <sub>13c</sub> f <sub>2</sub> imm <sub>7a</sub> qp				

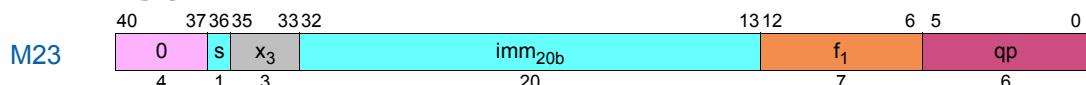
Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	
chk.s	f <sub>2</sub> , target <sub>25</sub>	1	3	

#### 4.4.5.3 Integer Advanced Load Check



Instruction	Operands	Opcode	Extension		
			x <sub>3</sub>	4	5
chk.a.nc	r <sub>1</sub> , target <sub>25</sub>	0			
chk.a.clr					

#### 4.4.5.4 Floating-point Advanced Load Check

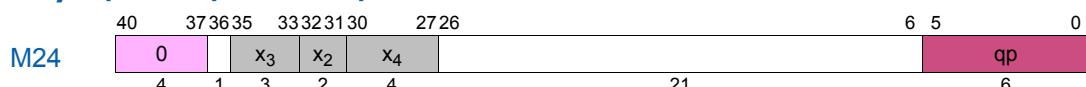


Instruction	Operands	Opcode	Extension		
			x <sub>3</sub>	6	7
chk.a.nc	f <sub>1</sub> , target <sub>25</sub>	0			
chk.a.clr					

### 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 3:345 for a summary of the opcode extensions.

#### 4.4.6.1 Sync/Fence/Serialize/ALAT Control



Instruction	Opcode	Extension		
		x <sub>3</sub>	x <sub>4</sub>	x <sub>2</sub>
invala	0	0	1	
fwb	0	0		
mf	0	2		2
mf.a	0	3		
srlz.d	0	0		
srlz.i	0	1		3
sync.i	0	3		

#### 4.4.6.2 RSE Control

M25	40 373635 33323130 2726	6 5 0		
	0   x <sub>3</sub>   x <sub>2</sub>   x <sub>4</sub>	0		
	4 1 3 2 4	21 6		
	Instruction	Opcode	Extension	
		x <sub>3</sub>	x <sub>4</sub>	x <sub>2</sub>
flushrs <sup>f</sup>	0	0	C	0
loadrs <sup>f</sup>			A	

#### 4.4.6.3 Integer ALAT Entry Invalidate

M26	40 373635 33323130 2726	13 12 6 5 0			
	0   x <sub>3</sub>   x <sub>2</sub>   x <sub>4</sub>	r <sub>1</sub> qp			
	4 1 3 2 4	14 7 6			
	Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>4</sub>	x <sub>2</sub>
invala.e	r <sub>1</sub>	0	0	2	1

#### 4.4.6.4 Floating-point ALAT Entry Invalidate

M27	40 373635 33323130 2726	13 12 6 5 0			
	0   x <sub>3</sub>   x <sub>2</sub>   x <sub>4</sub>	f <sub>1</sub> qp			
	4 1 3 2 4	14 7 6			
	Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>4</sub>	x <sub>2</sub>
invala.e	f <sub>1</sub>	0	0	3	1

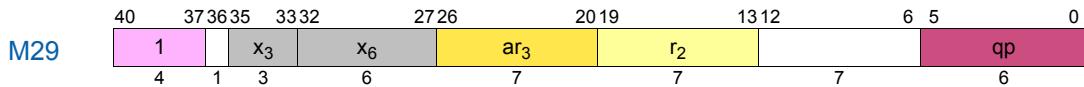
#### 4.4.6.5 Flush Cache

M28	40 373635 3332 2726 2019	6 5 0			
	1   x   x <sub>3</sub>   x <sub>6</sub>   r <sub>3</sub>	qp			
	4 1 3 6 7	14 6			
	Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>	x
fc	r <sub>3</sub>	1	0	30	0
fc.i					1

#### 4.4.7 GR/AR Moves (M-Unit)

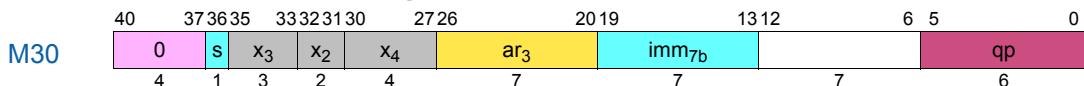
The M-Unit GR/AR move instructions are encoded in major opcode 0 along with the system/memory management instructions. (Some ARs are accessed using system control instructions on the I-unit. See “GR/AR Moves (I-Unit)” on page 3:321.) See “System/Memory Management” on page 3:345 for a summary of the M-Unit GR/AR opcode extensions.

#### 4.4.7.1 Move to AR – Register (M-Unit)



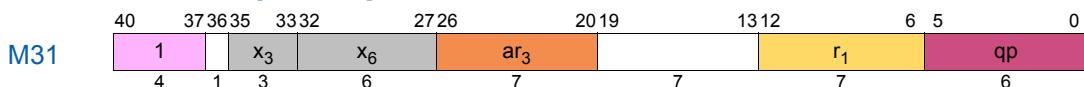
Instruction	Operands	Opcode	Extension	
			$x_3$	$x_6$
mov.m	$ar_3 = r_2$	1	0	2A

#### 4.4.7.2 Move to AR – Immediate (M-Unit)



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)

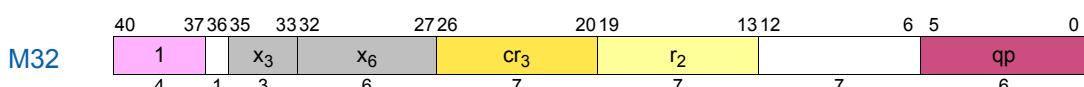


Instruction	Operands	Opcode	Extension	
			$x_3$	$x_6$
mov.m	$r_1 = ar_3$	1	0	22

### 4.4.8 GR/CR Moves

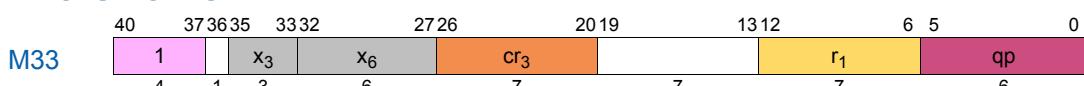
The GR/CR move instructions are encoded in major opcode 0 along with the system/memory management instructions. See “System/Memory Management” on page 3:345 for a summary of the opcode extensions.

#### 4.4.8.1 Move to CR



Instruction	Operands	Opcode	Extension	
			$x_3$	$x_6$
mov <sup>P</sup>	$cr_3 = r_2$	1	0	2C

#### 4.4.8.2 Move from CR



Instruction	Operands	Opcode	Extension	
			$x_3$	$x_6$
mov <sup>P</sup>	$r_1 = cr_3$	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 [3:345](#) for a summary of the opcode extensions.

### 4.4.9.1 Allocate Register Stack Frame

M34	40	373635	33323130	2726	2019	1312	6 5	0
	1	x <sub>3</sub>		sor	sol	sof	r <sub>1</sub>	qp

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	
alloc <sup>f</sup>	r <sub>1</sub> = ar.pfs, i, l, o, r	1		6

**Note:** The three immediates in the instruction encoding are formed from the operands as follows:

$$\begin{aligned} \text{sof} &= i + l + o \\ \text{sol} &= i + l \\ \text{sor} &= r >> 3 \end{aligned}$$

### 4.4.9.2 Move to PSR

M35	40	373635	3332	2726	2019	1312	6 5	0
	1	x <sub>3</sub>	x <sub>6</sub>			r <sub>2</sub>		qp

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
mov <sup>P</sup>	psr.l = r <sub>2</sub>	1		2D
mov	psr.um = r <sub>2</sub>		0	29

### 4.4.9.3 Move from PSR

M36	40	373635	3332	2726	1312	6 5	0
	1	x <sub>3</sub>	x <sub>6</sub>			r <sub>1</sub>	qp

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
mov <sup>P</sup>	r <sub>1</sub> = psr	1		25
mov	r <sub>1</sub> = psr.um		0	21

### 4.4.9.4 Break (M-Unit)

M37	40	373635	33323130	272625	imm <sub>20a</sub>	6 5	0
	0	i	x <sub>3</sub>	x <sub>2</sub>	x <sub>4</sub>		qp

Instruction	Operands	Opcode	Extension		
			x <sub>3</sub>	x <sub>4</sub>	x <sub>2</sub>
break.m	imm <sub>21</sub>	0	0	0	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 4-42](#) shows the 3-bit assignments for opcode 0, [Table 4-43](#) summarizes the 4-bit+2-bit assignments for opcode 0, [Table 4-44](#) shows the 3-bit assignments for opcode 1, and [Table 4-45](#) summarizes the 6-bit assignments for opcode 1.

**Table 4-42. 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 ( <a href="#">Table 4-43</a> )
	1	
	2	
	3	
	4	chk.a.nc – int <a href="#">M22</a>
	5	chk.a.clr – int <a href="#">M22</a>
	6	chk.a.nc – fp <a href="#">M23</a>
	7	chk.a.clr – fp <a href="#">M23</a>

**Table 4-43. 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 <a href="#">M37</a>	invala <a href="#">M24</a>	fwb <a href="#">M24</a>	srlz.d <a href="#">M24</a>
		1	1-bit Ext ( <a href="#">Table 4-46</a> )			srlz.i <a href="#">M24</a>
		2		invala.e – int <a href="#">M26</a>	mf <a href="#">M24</a>	
		3		invala.e – fp <a href="#">M27</a>	mf.a <a href="#">M24</a>	sync.i <a href="#">M24</a>
		4		sum <a href="#">M44</a>		
		5		rum <a href="#">M44</a>		
		6		ssm <a href="#">M44</a>		
		7		rsm <a href="#">M44</a>		
		8			mov.m to ar – imm <sub>8</sub> <a href="#">M30</a>	
		9				
		A	loadrs <a href="#">M25</a>			
		B				
		C	flushrs <a href="#">M25</a>			
		D				
		E				
		F				

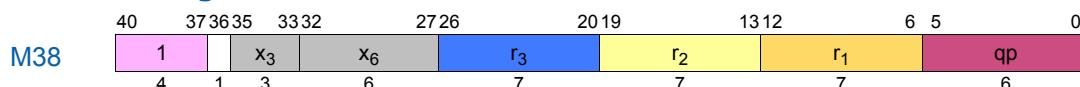
**Table 4-44. Opcode 1 System/Memory Management 3-bit Opcode Extensions**

Opcode Bits 40:37	x <sub>3</sub> Bits 35:33	
1	0	System/Memory Management 6-bit Ext (Table 4-45)
	1	chk.s.m – int M20
	2	
	3	chk.s – fp M21
	4	
	5	
	6	alloc M34
	7	

**Table 4-45. Opcode 1 System/Memory Management 6-bit Opcode Extensions**

Opcode Bits 40:37	x <sub>3</sub> Bits 35:33	x <sub>6</sub>				
		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 <sub>2</sub> M40
		2	mov to ibr M42	mov from ibr M43	mov.m from ar M31	probe.r.fault – imm <sub>2</sub> M40
		3	mov to pkr M42	mov from pkr M43		probe.w.fault – imm <sub>2</sub> M40
		4	mov to pmc M42	mov from pmc M43	mov from cr M33	ptc.e M47
		5	mov to pmd M42	mov from pmd M43	mov from psr M36	
		6				
		7		mov from cpuid M43		
		8		probe.r – imm <sub>2</sub> M39		probe.r M38
		9	ptc.l M45	probe.w – imm <sub>2</sub> 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



Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
probe.r		1	0	38
probe.w	r <sub>1</sub> = r <sub>3</sub> , r <sub>2</sub>			39

#### 4.4.10.2 Probe – Immediate<sub>2</sub>

M39	40 37 36 35 33 32 27 26 20 19 15 14 13 12 6 5 0
	4 1 3 6 7 5 2 7 6 5 6

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
probe.r	r <sub>1</sub> = r <sub>3</sub> , imm <sub>2</sub>	1	0	18
probe.w				19

#### 4.4.10.3 Probe Fault – Immediate<sub>2</sub>

M40	40 37 36 35 33 32 27 26 20 19 15 14 13 12 6 5 0
	4 1 3 6 7 5 2 7 6 5 6

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
probe.rw.fault				31
probe.r.fault	r <sub>3</sub> , imm <sub>2</sub>	1	0	32
probe.w.fault				33

#### 4.4.10.4 Translation Cache Insert

M41	40 37 36 35 33 32 27 26 20 19 13 12 6 5 0
	4 1 3 6 7 7 7 7 6

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
itc.d <sup>lp</sup>	r <sub>2</sub>	1	0	2E
itc.i <sup>lp</sup>				2F

#### 4.4.10.5 Move to Indirect Register/Translation Register Insert

M42	40 37 36 35 33 32 27 26 20 19 13 12 6 5 0
	4 1 3 6 7 7 7 7 6

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
mov <sup>p</sup>	rr[r <sub>3</sub> ] = r <sub>2</sub> dbr[r <sub>3</sub> ] = r <sub>2</sub> ibr[r <sub>3</sub> ] = r <sub>2</sub> pkr[r <sub>3</sub> ] = r <sub>2</sub> pmc[r <sub>3</sub> ] = r <sub>2</sub> pmd[r <sub>3</sub> ] = r <sub>2</sub>	1	0	00 01 02 03 04 05
itr.d <sup>p</sup> itr.i <sup>p</sup>	dtr[r <sub>3</sub> ] = r <sub>2</sub> itr[r <sub>3</sub> ] = r <sub>2</sub>			0E 0F

#### 4.4.10.6 Move from Indirect Register

M43	40 37 36 35 33 32 27 26 20 19 13 12 6 5 0
	1   x <sub>3</sub>   x <sub>6</sub>   r <sub>3</sub>   r <sub>1</sub>   qp

4 1 3 6 7 7 7 6

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
mov <sup>P</sup>	r <sub>1</sub> = rr[r <sub>3</sub> ] r <sub>1</sub> = dbr[r <sub>3</sub> ] r <sub>1</sub> = ibr[r <sub>3</sub> ] r <sub>1</sub> = pkr[r <sub>3</sub> ] r <sub>1</sub> = pmc[r <sub>3</sub> ]	1	0	10 11 12 13 14
	r <sub>1</sub> = pmd[r <sub>3</sub> ] r <sub>1</sub> = cpuid[r <sub>3</sub> ]			15 17

#### 4.4.10.7 Set/Reset User/System Mask

M44	40 37 36 35 33 32 31 30 27 26 6 5 0
	0   i   x <sub>3</sub>   i <sub>2d</sub>   x <sub>4</sub>   imm <sub>21a</sub>   qp

4 1 3 2 4 21 6

Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>4</sub>
sum				4
rum				5
ssm <sup>P</sup>	imm <sub>24</sub>	0	0	6
rsm <sup>P</sup>				7

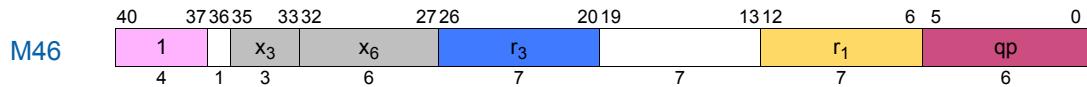
#### 4.4.10.8 Translation Purge

M45	40 37 36 35 33 32 27 26 20 19 13 12 6 5 0
	1   x <sub>3</sub>   x <sub>6</sub>   r <sub>3</sub>   r <sub>2</sub>   qp

4 1 3 6 7 7 7 6

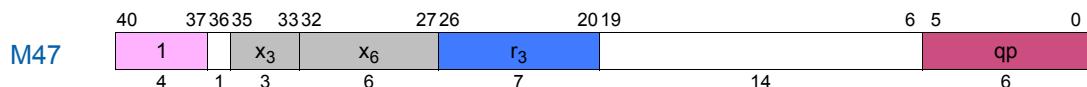
Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
ptc.l <sup>P</sup>				09
ptc.g <sup>I P</sup>				0A
ptc.ga <sup>I P</sup>	r <sub>3</sub> , r <sub>2</sub>	1	0	0B
ptr.d <sup>P</sup>				0C
ptr.i <sup>P</sup>				0D

#### 4.4.10.9 Translation Access



Instruction	Operands	Opcode	Extension	
			$x_3$	$x_6$
thash				1A
ttag				1B
tpa <sup>P</sup>	$r_1 = r_3$	1	0	1E
tak <sup>P</sup>				1F

#### 4.4.10.10 Purge Translation Cache Entry



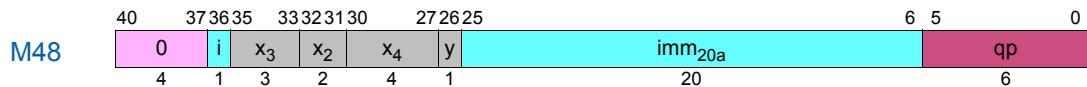
Instruction	Operands	Opcode	Extension	
			$x_3$	$x_6$
ptc.e <sup>P</sup>	$r_3$	1	0	34

#### 4.4.11 Nop/Hint (M-Unit)

M-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 ( $x_3$ ), a 2-bit opcode extension field in bits 32:31 ( $x_2$ ), a 4-bit opcode extension field in bits 30:27 ( $x_4$ ), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-46.

Table 4-46. Misc M-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	$x_3$ Bits 35:33	$x_4$ Bits 30:27	$x_2$ Bits 32:31	y Bit 26	
0	0	1	0	0	nop.m
				1	hint.m



Instruction	Operands	Opcode	Extension			
			$x_3$	$x_4$	$x_2$	y
nop.m		0	0	1	0	0
hint.m	imm <sub>21</sub>				1	1

### 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 4-47.

**Table 4-47. 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 4-48 summarizes these assignments.

**Table 4-48. 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 4-49)	e
	1		e	Indirect Return (Table 4-50)	e
	2	cover B8	e	e	e
	3	e	e	e	e
	4	clrrb B8	e	e	e
	5	clrrb.pr B8	e	e	e
	6	e	e	e	e
	7	e	e	e	e
	8	rfi B8	vmsw.0 B8	e	e
	9		vmsw.1 B8	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 4-49](#).

**Table 4-49. Indirect Branch Types**

Opcode Bits 40:37	x <sub>6</sub> Bits 32:27	btype Bits 8:6
0	20	0
		1
		2
		3
		4
		5
		6
		7

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 4-50](#).

**Table 4-50. Indirect Return Branch Types**

Opcode Bits 40:37	x <sub>6</sub> Bits 32:27	btype Bits 8:6
0	21	0
		1
		2
		3
		4
		5
		6
		7

All of the branch instructions have a 1-bit sequential prefetch opcode hint extension field, p, in bit 12. [Table 4-51](#) summarizes these assignments.

**Table 4-51. Sequential Prefetch Hint Completer**

p Bit 12	ph
0	.few
1	.many

The IP-relative and indirect branch instructions all have a 2-bit branch prediction “whether” opcode hint extension field in bits 34:33 (wh) as shown in [Table 4-52](#). Indirect call instructions have a 3-bit “whether” opcode hint extension field in bits 34:32 (wh) as shown in [Table 4-53](#).

**Table 4-52. Branch Whether Hint Completer**

wh Bits 34:33	bwh
0	.sptk
1	.spnt
2	.dptk
3	.dpnt

**Table 4-53. Indirect Call Whether Hint Completer**

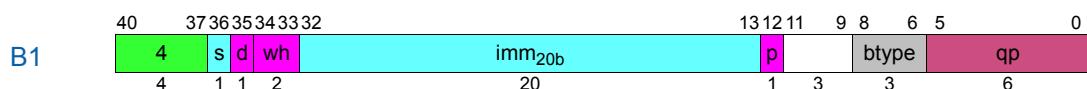
wh Bits 34:32	bwh
0	
1	.sptk
2	
3	.spnt
4	
5	.dptk
6	
7	.dpnt

The branch instructions also have a 1-bit branch cache deallocation opcode hint extension field in bit 35 (d) as shown in [Table 4-54](#).

**Table 4-54. Branch Cache Deallocation Hint Completer**

d Bit 35	dh
0	none
1	.clr

#### 4.5.1.1 IP-Relative Branch



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cond.bwh.ph.dh <sup>e</sup>			0	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352
br.wexit.bwh.ph.dh <sup>e t</sup>	target <sub>25</sub>	4	2			
br.wtop.bwh.ph.dh <sup>e t</sup>			3			

#### 4.5.1.2 IP-Relative Counted Branch

	40	37	36	35	34	33	32		13	12	11	9	8	6	5	0	
B2	4	s	d	wh	imm <sub>20b</sub>								p	btype	0		
	4	1	1	2									1	3	3	6	

Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cloop.bwh.ph.dh <sup>e t</sup>	target <sub>25</sub>	4	5	See	See	See
br.cexit.bwh.ph.dh <sup>e t</sup>			6	Table 4-51 on page 3:351	Table 4-52 on page 3:352	Table 4-54 on page 3:352
br.ctop.bwh.ph.dh <sup>e t</sup>			7			

### 4.5.1.3 IP-Relative Call

Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.bwh.ph.dh <sup>e</sup>	$b_1 = target_{25}$	5	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

#### **4.5.1.4 Indirect Branch**

B4	40	37	36	35	34	33	32	27	26	16	15	13	12	11	9	8	6	5	0
	0	d	wh	x <sub>6</sub>				b <sub>2</sub>	p	btype					qp				
	4	1	1	2	6			11		3	1	3	3	3	6				

Instruction	Operands	Opcode	Extension				
			x <sub>6</sub>	btype	p	wh	d
br.cond.bwh.ph.dh <sup>e</sup>	<i>b</i> <sub>2</sub>	0	20	0	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352
br.ia.bwh.ph.dh <sup>e</sup>			21	1			
br.ret.bwh.ph.dh <sup>e</sup>				4			

#### 4.5.1.5 Indirect Call

Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.bwh.ph.dh <sup>e</sup>	$b_1 = b_2$	1	See Table 4-51 on page 3:351	See Table 4-53 on page 3:352	See Table 4-54 on page 3:352

## 4.5.2 Branch Predict/Nop/Hint

The branch predict, nop, and hint instructions are encoded in major opcodes 2 (Indirect Predict/Nop/Hint) and 7 (IP-relative Predict). The indirect predict, nop, and hint instructions in major opcode 2 use a 6-bit opcode extension field in bits 32:27 ( $x_6$ ).

Table 4-55 summarizes these assignments.

**Table 4-55. Indirect Predict/Nop/Hint Opcode Extensions**

Opcode Bits 40:37	x <sub>6</sub>				
	Bits 30:27	Bits 32:31			
		0	1	2	3
2	0	nop.b B9	brp B7		
	1	hint.b B9	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 branch importance opcode hint extension field in bit 35 (ih). The mov to BR instruction ([page 3:320](#)) also has this hint in bit 23. [Table 4-56](#) shows these assignments.

**Table 4-56. 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 branch prediction “whether” opcode hint extension field in bits 4:3 (wh) as shown in [Table 4-57](#). Note that the combination of the .loop or .exit whether hint completer with the *none* importance hint completer is undefined.

**Table 4-57. IP-Relative Predict Whether Hint Completer**

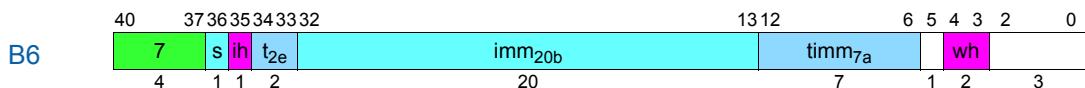
wh Bits 4:3	ipwh
0	.sptk
1	.loop
2	.dptk
3	.exit

The indirect branch predict instructions have a 2-bit branch prediction “whether” opcode hint extension field in bits 4:3 (wh) as shown in [Table 4-58](#).

**Table 4-58. Indirect Predict Whether Hint Completer**

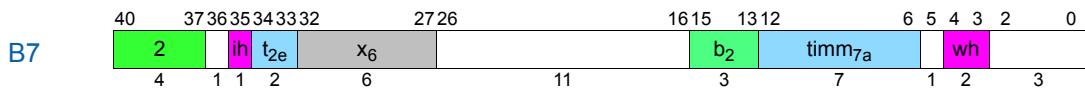
wh Bits 4:3	indwh
0	.sptk
1	
2	.dptk
3	

#### 4.5.2.1 IP-Relative Predict



Instruction	Operands	Opcode	Extension		
			ih	wh	
brp.ipwh.ih	target <sub>25</sub> , tag <sub>13</sub>	7	See Table 4-56 on page 3:354	See Table 4-57 on page 3:354	

#### 4.5.2.2 Indirect Predict

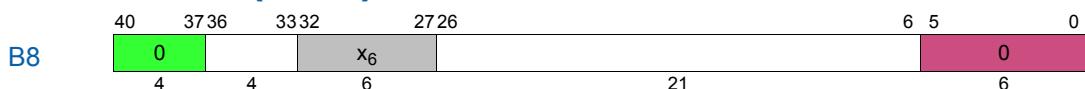


Instruction	Operands	Opcode	Extension		
			x <sub>6</sub>	ih	wh
brp.indwh.ih	b <sub>2</sub> , tag <sub>13</sub>	2	10	See Table 4-56 on page 3:354	See Table 4-58 on page 3:355
brp.ret.indwh.ih			11		

### 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<sub>6</sub>) as described in Table 4-48 on page 3:350.

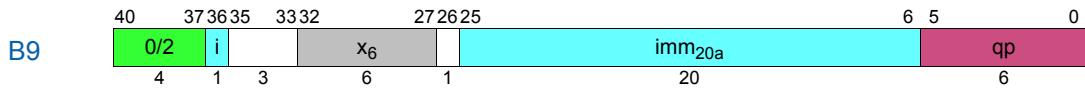
#### 4.5.3.1 Miscellaneous (B-Unit)



Instruction	Opcode	Extension	
		x <sub>6</sub>	
cover <sup>1</sup>		02	
clrrrb <sup>1</sup>		04	
clrrrb.pr <sup>1</sup>		05	
rfeelp	0	08	
bsw.0lp		0C	
bsw.1lp		0D	
epc		10	

Instruction	Opcode	Extension
		$x_6$
vmsw.0 P	0	18
vmsw.1 P		19

#### 4.5.3.2 Break/Nop/Hint (B-Unit)



Instruction	Operands	Opcode	Extension
			$x_6$
break.b <sup>e</sup>	$imm_{21}$	0	
			00
		2	01

## 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 4-59](#) shows the 1-bit x assignments, [Table 4-62](#) shows the additional 1-bit q assignments for the reciprocal approximation instructions; [Table 4-60](#) and [Table 4-61](#) summarize the 6-bit  $x_6$  assignments.

**Table 4-59. Miscellaneous Floating-point 1-bit Opcode Extensions**

Opcode Bits 40:37	x Bit 33	
0	0	6-bit Ext ( <a href="#">Table 4-60</a> )
	1	Reciprocal Approximation ( <a href="#">Table 4-62</a> )
1	0	6-bit Ext ( <a href="#">Table 4-61</a> )
	1	Reciprocal Approximation ( <a href="#">Table 4-62</a> )

**Table 4-60. Opcode 0 Miscellaneous Floating-point 6-bit Opcode Extensions**

Opcode Bits 40:37	x Bit 33	x <sub>6</sub>				
		Bits 30:27	Bits 32:31			
			0	1	2	3
0	0	0	break.f F15	fmerge.s F9		
		1	1-bit Ext (Table 4-68)	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.f xu F10		fmix.lr F9
		A		fcvt.fx.trunc F10		fmix.r F9
		B		fcvt.f xu.trunc F10		fmix.I F9
		C		fcvt.xf F11	fand F9	fsxt.r F9
		D			fandcm F9	fsxt.I F9
		E			for F9	
		F			fxor F9	

**Table 4-61. Opcode 1 Miscellaneous Floating-point 6-bit Opcode Extensions**

Opcode Bits 40:37	x Bit 33	x <sub>6</sub>				
		Bits 30:27	Bits 32:31			
			0	1	2	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.f xu F10		
		A		fpcvt.fx.trunc F10		
		B		fpcvt.f xu.trunc F10		
		C				
		D				
		E				
		F				

**Table 4-62. 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

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 4-63](#) summarizes these assignments.

**Table 4-63. 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 4-64](#).

**Table 4-64. 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<sub>2</sub>) in bits 35:34. These assignments are shown in [Table 4-65](#).

**Table 4-65. Fixed-point Multiply Add and Select Opcode Extensions**

Opcode Bits 40:37	x Bit 36	x <sub>2</sub> 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	40 37 36 35 34 33	27 26	20 19	13 12	6 5	0
	8 - D   x   sf   f <sub>4</sub>   f <sub>3</sub>   f <sub>2</sub>   f <sub>1</sub>   qp	4 1 2 7 7 7 7 6				

Instruction	Operands	Opcode	Extension	
			x	sf
fma.sf		8	0	
fma.s.sf		8	1	
fma.d.sf		9	0	
fpma.sf		9	1	
fms.sf	$f_1 = f_3, f_4, f_2$	A	0	See Table 4-63 on page 3:358
fms.s.sf		A	1	
fms.d.sf		B	0	
fpms.sf		B	1	
fnma.sf	$f_1 = f_3, f_4, f_2$	C	0	
fnma.s.sf		C	1	
fnma.d.sf		D	0	
fpnma.sf		D	1	

#### 4.6.1.2 Fixed-point Multiply Add

F2	40 37 36 35 34 33	27 26	20 19	13 12	6 5	0
	8 - D   x   x <sub>2</sub>   f <sub>4</sub>   f <sub>3</sub>   f <sub>2</sub>   f <sub>1</sub>   qp	4 1 2 7 7 7 7 6				

Instruction	Operands	Opcode	Extension	
			x	x <sub>2</sub>
xma.l		E		0
xma.h	$f_1 = f_3, f_4, f_2$	E	1	3
xma.hu		E		2

#### 4.6.2 Parallel Floating-point Select

F3	40 37 36 35 34 33	27 26	20 19	13 12	6 5	0
	8 - D   x     f <sub>4</sub>   f <sub>3</sub>   f <sub>2</sub>   f <sub>1</sub>   qp	4 1 2 7 7 7 7 6				

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 4-66. The  $sf$  assignments are shown in Table 4-63 on page 3:358.

The parallel floating-point compare instructions are described on page 3:362.

**Table 4-66. Floating-point Compare Opcode Extensions**

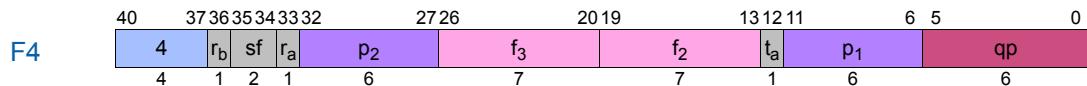
Opcode Bits 40:37	$r_a$ Bit 33	$r_b$ Bit 36	$t_a$ Bit 12	
			0	1
4	0	0	fcmp.eq F4	fcmp.eq.unc F4
		1	fcmp.lt F4	fcmp.lt.unc F4
	1	0	fcmp.le F4	fcmp.le.unc F4
		1	fcmp.unord F4	fcmp.unord.unc F4

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 4-67](#).

**Table 4-67. Floating-point Class 1-bit Opcode Extensions**

Opcode Bits 40:37	$t_a$ Bit 12
5	0
	1

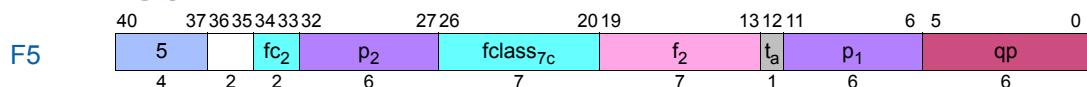
#### 4.6.3.1 Floating-point Compare



Instruction	Operands	Opcode	Extension			
			$r_a$	$r_b$	$t_a$	$sf$
fcmp.eq.sf			0	0		
fcmp.lt.sf			1	1	0	
fcmp.le.sf			0	0		
fcmp.unord.sf			1	1		
fcmp.eq.unc.sf	$p_1, p_2 = f_2, f_3$	4	0	0		
fcmp.lt.unc.sf			1	1		
fcmp.le.unc.sf			0	0	1	
fcmp.unord.unc.sf			1	1		

See Table 4-63 on page 3:358

#### 4.6.3.2 Floating-point Class



Instruction	Operands	Opcode	Extension	
			$t_a$	
fclass.m			0	
fclass.m.unc	$p_1, p_2 = f_2, fclass_9$	5	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	40 373635343332	2726	2019	1312	6 5	0
	0 - 1   q   sf   x   p <sub>2</sub>   f <sub>3</sub>   f <sub>2</sub>   f <sub>1</sub>   qp	4 1 2 1 6 7 7 7 6				

Instruction	Operands	Opcode	Extension		
			x	q	sf
frcpa.sf		0			
fprcpa.sf	f <sub>1</sub> , p <sub>2</sub> = f <sub>2</sub> , f <sub>3</sub>	1	1	0	See Table 4-63 on page 3:358

### 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	40 373635343332	2726	2019	1312	6 5	0
	0 - 1   q   sf   x   p <sub>2</sub>   f <sub>3</sub>   f <sub>1</sub>   qp	4 1 2 1 6 7 7 6				

Instruction	Operands	Opcode	Extension		
			x	q	sf
frsqrta.sf		0			
fprsqrta.sf	f <sub>1</sub> , p <sub>2</sub> = f <sub>3</sub>	1	1	1	See Table 4-63 on page 3:358

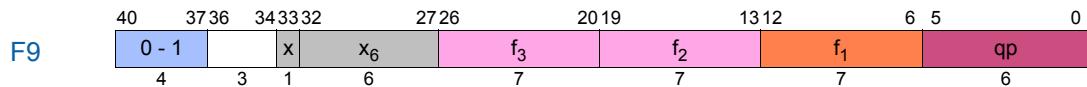
## 4.6.5 Minimum/Maximum and Parallel Compare

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.

	40	37	36	35	34	33	32	27	26	20	19	13	12	6	5	0
F8	0 - 1			sf	x		x <sub>6</sub>		f <sub>3</sub>		f <sub>2</sub>		f <sub>1</sub>		qp	
	4	1	2	1			6		7		7		7		6	

Instruction	Operands	Opcode	Extension		
			x	x <sub>6</sub>	sf
fmin.sf	$f_1 = f_2, f_3$	0		14	
fmax.sf				15	
famin.sf				16	
famax.sf				17	
fpmin.sf	$f_1 = f_2, f_3$	0		14	
fpmax.sf				15	
fpamin.sf				16	
fpamax.sf				17	
fpcmp.eq.sf		1		30	See Table 4-63 on page 3:358
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	

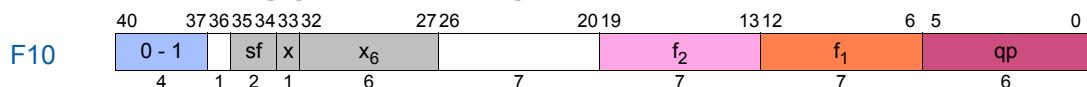
## 4.6.6 Merge and Logical



Instruction	Operands	Opcode	Extension	
			x	x <sub>6</sub>
fmerge.s				10
fmerge.ns				11
fmerge.se				12
fmix.lr				39
fmix.r				3A
fmix.l				3B
fsxt.r				3C
fsxt.l				3D
fpack		0		28
fswap	f <sub>1</sub> = f <sub>2</sub> , f <sub>3</sub>		0	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



Instruction	Operands	Opcode	Extension		
			x	x <sub>6</sub>	sf
fcvt.fx.sf				18	
fcvt.fxu.sf				19	
fcvt.fx.trunc.sf		0		1A	
fcvt.fxu.trunc.sf				1B	
fpcvt.fx.sf	f <sub>1</sub> = f <sub>2</sub>	0		18	See Table 4-63 on page 3:358
fpcvt.fxu.sf				19	
fpcvt.fx.trunc.sf		1		1A	
fpcvt.fxu.trunc.sf				1B	

#### 4.6.7.2 Convert Fixed-point to Floating-point

F11	40 3736 343332 2726 2019 1312 6 5 0																		
	<table border="1"> <tr> <td>0</td><td>x</td><td>x<sub>6</sub></td><td></td><td>f<sub>2</sub></td><td>f<sub>1</sub></td><td>qp</td><td></td><td></td></tr> <tr> <td>4</td><td>3</td><td>1</td><td>6</td><td>7</td><td>7</td><td>6</td><td></td><td></td></tr> </table>	0	x	x <sub>6</sub>		f <sub>2</sub>	f <sub>1</sub>	qp			4	3	1	6	7	7	6		
0	x	x <sub>6</sub>		f <sub>2</sub>	f <sub>1</sub>	qp													
4	3	1	6	7	7	6													

Instruction	Operands	Opcode	Extension	
			x	x <sub>6</sub>
fcvt.xf	f <sub>1</sub> = f <sub>2</sub>	0	0	1C

#### 4.6.8 Status Field Manipulation

##### 4.6.8.1 Floating-point Set Controls

F12	40 373635343332 2726 2019 1312 6 5 0																		
	<table border="1"> <tr> <td>0</td><td>sf</td><td>x</td><td>x<sub>6</sub></td><td>omask<sub>7c</sub></td><td>amask<sub>7b</sub></td><td></td><td>qp</td><td></td></tr> <tr> <td>4</td><td>1</td><td>2</td><td>1</td><td>6</td><td>7</td><td>7</td><td>6</td><td></td></tr> </table>	0	sf	x	x <sub>6</sub>	omask <sub>7c</sub>	amask <sub>7b</sub>		qp		4	1	2	1	6	7	7	6	
0	sf	x	x <sub>6</sub>	omask <sub>7c</sub>	amask <sub>7b</sub>		qp												
4	1	2	1	6	7	7	6												

Instruction	Operands	Opcode	Extension		
			x	x <sub>6</sub>	sf
fsetc.sf	amask <sub>7</sub> , omask <sub>7</sub>	0	0	04	See Table 4-63 on page 3:358

##### 4.6.8.2 Floating-point Clear Flags

F13	40 373635343332 2726 6 5 0																		
	<table border="1"> <tr> <td>0</td><td>sf</td><td>x</td><td>x<sub>6</sub></td><td></td><td>qp</td></tr> <tr> <td>4</td><td>1</td><td>2</td><td>1</td><td>6</td><td>6</td></tr> <tr> <td></td><td></td><td></td><td></td><td>21</td><td></td></tr> </table>	0	sf	x	x <sub>6</sub>		qp	4	1	2	1	6	6					21	
0	sf	x	x <sub>6</sub>		qp														
4	1	2	1	6	6														
				21															

Instruction	Opcode	Extension		
		x	x <sub>6</sub>	sf
fclr.ssf	0	0	05	See Table 4-63 on page 3:358

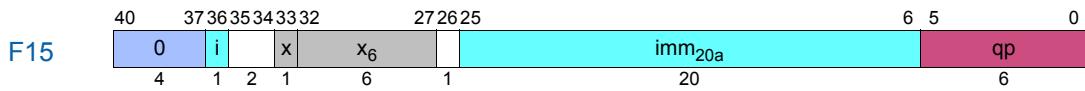
##### 4.6.8.3 Floating-point Check Flags

F14	40 373635343332 272625 6 5 0																
	<table border="1"> <tr> <td>0</td><td>s</td><td>sf</td><td>x</td><td>x<sub>6</sub></td><td>imm<sub>20a</sub></td><td>qp</td><td></td></tr> <tr> <td>4</td><td>1</td><td>2</td><td>1</td><td>6</td><td>1</td><td>20</td><td>6</td></tr> </table>	0	s	sf	x	x <sub>6</sub>	imm <sub>20a</sub>	qp		4	1	2	1	6	1	20	6
0	s	sf	x	x <sub>6</sub>	imm <sub>20a</sub>	qp											
4	1	2	1	6	1	20	6										

Instruction	Operands	Opcode	Extension		
			x	x <sub>6</sub>	sf
fchkf.ssf	target <sub>25</sub>	0	0	08	See Table 4-63 on page 3:358

## 4.6.9 Miscellaneous F-Unit Instructions

### 4.6.9.1 Break (F-Unit)



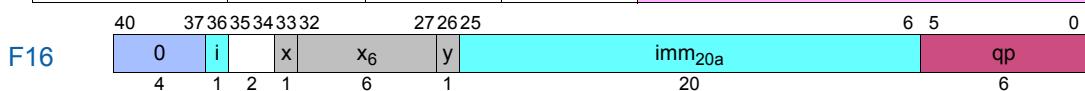
Instruction	Operands	Opcode	Extension	
			x	x <sub>6</sub>
break.f	imm <sub>21</sub>	0	0	00

### 4.6.9.2 Nop/Hint (F-Unit)

F-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x<sub>3</sub>), a 6-bit opcode extension field in bits 32:27 (x<sub>6</sub>), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-46.

**Table 4-68. Misc F-Unit 1-bit Opcode Extensions**

Opcode Bits 40:37	x Bit :33	x <sub>6</sub> Bits 32:27	y Bit 26	
0	0	01	0	nop.f
			1	hint.f



Instruction	Operands	Opcode	Extension		
			x	x <sub>6</sub>	y
nop.f hint.f	imm <sub>21</sub>	0	0	01	0 1

## 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<sub>41</sub> field occupies the L instruction slot. For brl, the imm<sub>39</sub> 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<sub>3</sub>) in bits 35:33 and a 6-bit opcode extension field (x<sub>6</sub>) in bits 32:27. Table 4-69 shows the 3-bit assignments and Table 4-70 summarizes the 6-bit assignments. These instructions are executed by an I-unit.

**Table 4-69. Misc X-Unit 3-bit Opcode Extensions**

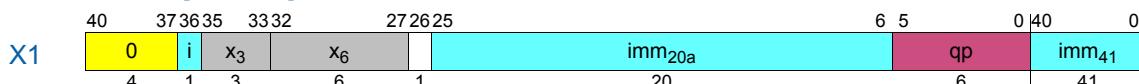
Opcode Bits 40:37		x <sub>3</sub> Bits 35:33	
0	0	0	6-bit Ext (Table 4-70)
	1	1	
	2	2	
	3	3	
	4	4	
	5	5	
	6	6	
	7	7	

**Table 4-70. Misc X-Unit 6-bit Opcode Extensions**

Opcode Bits 40:37	x <sub>3</sub> Bits 35:33	x <sub>6</sub>				
		Bits 30:27	Bits 32:31			
			0	1	2	3
0	0	0	break.x X1			
		1	1-bit Ext (Table 4-73)			
		2				
		3				
		4				
		5				
		6				
		7				
		8				
		9				
		A				
		B				
		C				
		D				
		E				
		F				

#### 4.7.1.1

#### Break (X-Unit)



Instruction	Operands	Opcode	Extension	
			x <sub>3</sub>	x <sub>6</sub>
break.x	imm <sub>62</sub>	0	0	00

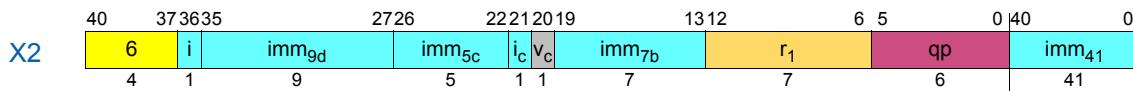
#### 4.7.2

#### Move Long Immediate<sub>64</sub>

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 4-71. This instruction is executed by an I-unit.

**Table 4-71. Move Long 1-bit Opcode Extensions**

Opcode Bits 40:37	V <sub>c</sub> Bit 20
6	0
	1



Instruction	Operands	Opcode	Extension
			V <sub>c</sub>
movl <sup>i</sup>	$r_1 = imm_{64}$	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 4-72](#).

**Table 4-72. 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 4-51 on page 3:351](#), [Table 4-52 on page 3:352](#), and [Table 4-54 on page 3:352](#).

#### 4.7.3.1 Long Branch



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
brl.cond.bwh.ph.dh e1	$target_{64}$	C	0	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

### 4.7.3.2 Long Call

X4

Instruction		Operands		Opcode	Extension					
					p		wh		d	
bri.call.bwh.ph.dh e1		$b_1 = \text{target}_{64}$		D	See Table 4-51 on page 3:351		See Table 4-52 on page 3:352		See Table 4-54 on page 3:352	

### 4.7.4 Nop/Hint (X-Unit)

X-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 ( $x_3$ ), a 6-bit opcode extension field in bits 32:27 ( $x_6$ ), and a 1-bit opcode extension field in bit 26 (y), as shown in [Table 4-73](#). These instructions are executed by an I-unit.

**Table 4-73. Misc X-Unit 1-bit Opcode Extensions**

X5

Opcode Bits 40:37		$x_3$ Bits 35:33		$x_6$ Bits 32:27		y Bit 26	Extension				
0		0		01		0	nop.x				
						1	hint.x				
0		37 36 35 33 32		27 26 25			6	5	0	40	0
		4 1 3		6		1	imm20a		qp	imm41	
									6		41

Instruction		Operands		Opcode	Extension		
					$x_3$	$x_6$	y
nop.x		$imm_{62}$		0	0	01	0
hint.x							1

## 4.8 Immediate Formation

[Table 4-74](#) 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 4-74. Immediate Formation**

Instruction Format	Immediate Formation
A2	$\text{count}_2 = ct_{2d} + 1$
A3 A8 I27 M30	$\text{imm}_8 = \text{sign\_ext}(s << 7   \text{imm}_{7b}, 8)$
A4	$\text{imm}_{14} = \text{sign\_ext}(s << 13   \text{imm}_{6d} << 7   \text{imm}_{7b}, 14)$
A5	$\text{imm}_{22} = \text{sign\_ext}(s << 21   \text{imm}_{5c} << 16   \text{imm}_{9d} << 7   \text{imm}_{7b}, 22)$
A10	$\text{count}_2 = (ct_{2d} > 2) ? \text{reservedQP}^a : ct_{2d} + 1$
I1	$\text{count}_2 = (ct_{2d} == 0) ? 0 : (ct_{2d} == 1) ? 7 : (ct_{2d} == 2) ? 15 : 16$

**Table 4-74. Immediate Formation (Continued)**

Instruction Format	Immediate Formation
I3	$\text{mbtype}_4 = (\text{mbt}_{4c} == 0) ? @\text{br cst} : (\text{mbt}_{4c} == 8) ? @\text{mix} : (\text{mbt}_{4c} == 9) ? @\text{shuf} : (\text{mbt}_{4c} == 0xA) ? @\text{alt} : (\text{mbt}_{4c} == 0xB) ? @\text{rev} : \text{reservedQP}^a$
I4	$\text{mhtype}_8 = \text{mht}_{8c}$
I6	$\text{count}_5 = \text{count}_{5b}$
I8	$\text{count}_5 = 31 - \text{ccount}_{5c}$
I10	$\text{count}_6 = \text{count}_{6d}$
I11	$\text{len}_6 = \text{len}_{6d} + 1$ $\text{pos}_6 = \text{pos}_{6b}$
I12	$\text{len}_6 = \text{len}_{6d} + 1$ $\text{pos}_6 = 63 - \text{cpos}_{6c}$
I13	$\text{len}_6 = \text{len}_{6d} + 1$ $\text{pos}_6 = 63 - \text{cpos}_{6c}$ $\text{imm}_8 = \text{sign\_ext}(\text{s} \ll 7 \mid \text{imm}_{7b}, 8)$
I14	$\text{len}_6 = \text{len}_{6d} + 1$ $\text{pos}_6 = 63 - \text{cpos}_{6b}$ $\text{imm}_1 = \text{sign\_ext}(\text{s}, 1)$
I15	$\text{len}_4 = \text{len}_{4d} + 1$ $\text{pos}_6 = 63 - \text{cpos}_{6d}$
I16	$\text{pos}_6 = \text{pos}_{6b}$
I18 I19 M37 M48	$\text{imm}_{21} = \text{i} \ll 20 \mid \text{imm}_{20a}$
I21	$\text{tag}_{13} = \text{IP} + (\text{sign\_ext}(\text{timm}_{9c}, 9) \ll 4)$
I23	$\text{mask}_{17} = \text{sign\_ext}(\text{s} \ll 16 \mid \text{mask}_{8c} \ll 8 \mid \text{mask}_{7a} \ll 1, 17)$
I24	$\text{imm}_{44} = \text{sign\_ext}(\text{s} \ll 43 \mid \text{imm}_{27a} \ll 16, 44)$
I30	$\text{imm}_5 = \text{imm}_{5b} + 32$
M3 M8 M22	$\text{imm}_9 = \text{sign\_ext}(\text{s} \ll 8 \mid \text{i} \ll 7 \mid \text{imm}_{7b}, 9)$
M5 M10	$\text{imm}_9 = \text{sign\_ext}(\text{s} \ll 8 \mid \text{i} \ll 7 \mid \text{imm}_{7a}, 9)$
M17	$\text{inc}_3 = \text{sign\_ext}((\text{s}) ? -1 : 1) * ((\text{i}_{2b} == 3) ? 1 : 1 \ll (4 - \text{i}_{2b}), 6)$
I20 M20 M21	$\text{target}_{25} = \text{IP} + (\text{sign\_ext}(\text{s} \ll 20 \mid \text{imm}_{13c} \ll 7 \mid \text{imm}_{7a}, 21) \ll 4)$
M22 M23	$\text{target}_{25} = \text{IP} + (\text{sign\_ext}(\text{s} \ll 20 \mid \text{imm}_{20b}, 21) \ll 4)$
M34	$\text{il} = \text{sol}$ $\text{o} = \text{sof} - \text{sol}$ $\text{r} = \text{sol} \ll 3$
M39 M40	$\text{imm}_2 = \text{i}_{2b}$
M44	$\text{imm}_{24} = \text{i} \ll 23 \mid \text{i}_{2d} \ll 21 \mid \text{imm}_{21a}$
B1 B2 B3	$\text{target}_{25} = \text{IP} + (\text{sign\_ext}(\text{s} \ll 20 \mid \text{imm}_{20b}, 21) \ll 4)$
B6	$\text{target}_{25} = \text{IP} + (\text{sign\_ext}(\text{s} \ll 20 \mid \text{imm}_{20b}, 21) \ll 4)$ $\text{tag}_{13} = \text{IP} + (\text{sign\_ext}(\text{t}_{2e} \ll 7 \mid \text{timm}_{7a}, 9) \ll 4)$
B7	$\text{tag}_{13} = \text{IP} + (\text{sign\_ext}(\text{t}_{2e} \ll 7 \mid \text{timm}_{7a}, 9) \ll 4)$
B9	$\text{imm}_{21} = \text{i} \ll 20 \mid \text{imm}_{20a}$
F5	$\text{fclass}_9 = \text{fclass}_{7c} \ll 2 \mid \text{fc}_2$
F12	$\text{amask}_7 = \text{amask}_{7b}$ $\text{omask}_7 = \text{omask}_{7c}$
F14	$\text{target}_{25} = \text{IP} + (\text{sign\_ext}(\text{s} \ll 20 \mid \text{imm}_{20a}, 21) \ll 4)$
F15 F16	$\text{imm}_{21} = \text{i} \ll 20 \mid \text{imm}_{20a}$
X1 X5	$\text{imm}_{62} = \text{imm}_{41} \ll 21 \mid \text{i} \ll 20 \mid \text{imm}_{20a}$
X2	$\text{imm}_{64} = \text{i} \ll 63 \mid \text{imm}_{41} \ll 22 \mid \text{i}_c \ll 21 \mid \text{imm}_{5c} \ll 16 \mid \text{imm}_{9d} \ll 7 \mid \text{imm}_{7b}$
X3 X4	$\text{target}_{64} = \text{IP} + ((\text{i} \ll 59 \mid \text{imm}_{39} \ll 20 \mid \text{imm}_{20b}) \ll 4)$

- a. This encoding causes an Illegal Operation fault if the value of the qualifying predicate is 1.

§

## 5.1 Reading and Writing Resources

An Itanium instruction is said to be a **reader** of a resource if the instruction's qualifying predicate is 1 or it has no qualifying predicate or is one of the instructions that reads a resource even when its qualifying predicate is 0, and the execution of the instruction depends on that resource.

An Itanium instruction is said to be an **writer** of a resource if the instruction's qualifying predicate is 1 or it has no qualifying predicate or writes the resource even when the qualifying predicate is 0, and the execution of the instruction writes that resource.

An Itanium instruction is said to be a reader or writer of a resource even if it only sometimes depends on that resource and it cannot be determined statically whether the resource will be read or written. For example, `cover` only writes CR[IFS] when PSR.ic is 0, but for purposes of dependency, it is treated as if it always writes the resource since this condition cannot be determined statically. On the other hand, `rsm` conditionally writes several bits in the PSR depending on a mask which is encoded as an immediate in the instruction. Since the PSR bits to be written can be determined by examining the encoded instruction, the instruction is treated as only writing those bits which have a corresponding mask bit set. All exceptions to these general rules are described in this appendix.

## 5.2 Dependencies and Serialization

A **RAW** (Read-After-Write) dependency is a sequence of two events where the first is a writer of a resource and the second is a reader of the same resource. Events may be instructions, interruptions, or other 'uses' of the resource such as instruction stream fetches and VHPT walks. [Table 5-2](#) covers only dependencies based on instruction readers and writers.

A **WAW** (Write-After-Write) dependency is a sequence of two events where both events write the resource in question. Events may be instructions, interruptions, or other 'updates' of the resource. [Table 5-3](#) covers only dependencies based on instruction writers.

A **WAR** (Write-After-Read) dependency is a sequence of two instructions, where the first is a reader of a resource and the second is a writer of the same resource. Such dependencies are always allowed except as indicated in [Table 5-4](#) and only those related to instruction readers and writers are included.

A **RAR** (Read-After-Read) dependency is a sequence of two instructions where both are readers of the same resource. Such dependencies are always allowed.

RAW and WAW dependencies are generally not allowed without some type of serialization event (an implied, data, or instruction serialization after the first writing instruction. (See [Section 3.2, "Serialization" on page 2:17](#) for details on serialization.) The tables and associated rules in this appendix provide a comprehensive list of readers and writers of resources and describe the serialization required for the dependency to be observed and possible outcomes if the required serialization is not met. Even when targeting code for machines which do not check for particular disallowed dependencies, such code sequences are considered architecturally undefined and may cause code to behave differently across processors, operating systems, or even separate executions of the code sequence during the same program run. In some cases, different serializations may yield different, but well-defined results.

The serialization of application level (non-privileged) resources is always implied. This means that if a writer of that resource and a subsequent read of that same resource are in different instruction groups, then the reader will see the value written. In addition, for dependencies on PRs and BRs, where the writer is a non-branch instruction and the reader is a branch instruction, the writer and reader may be in the same instruction group.

System resources generally require explicit serialization, i.e., the use of a `srlz.i` or `srlz.d` instruction, between the writing and the reading of that resource. Note that RAW accesses to CRs are not exceptional – they require explicit data or instruction serialization. However, in some cases (other than CRs) where pairs of instructions explicitly encode the same resource, serialization is implied.

There are cases where it is architecturally allowed to omit a serialization, and that the response from the CPU must be atomic (act as if either the old or the new state were fully in place). The tables in this appendix indicate dependency requirements under the assumption that the desired result is for the dependency to always be observed. In some such cases, the programmer may not care if the old or new state is used; such situations are allowed, but the value seen is not deterministic.

On the other hand, if an *impliedF* dependency is violated, then the program is incorrectly coded and the processor's behavior is undefined.

## 5.3 Resource and Dependency Table Format Notes

- The “Writers” and “Readers” columns of the dependency tables contain instruction class names and instruction mnemonic prefixes as given in the format section of each instruction page. To avoid ambiguity, instruction classes are shown in bold, while instruction mnemonic prefixes are in regular font. For instruction mnemonic prefixes, all instructions that exactly match the name specified or those that begin with the specified text and are followed by a ‘.’ and then followed by any other text will match.
- The dependency on a listed instruction is in effect no matter what values are encoded in the instruction or what dynamic values occur in operands, unless a superscript is present or one of the special case instruction rules in [Section 5.3.1](#) applies. Instructions listed are still subject to rules regarding qualifying predicates.
- Instruction classes are groups of related instructions. Such names appear in boldface for clarity. The list of all instruction classes is contained in [Table 5-5](#). Note that an instruction may appear in multiple instruction classes, instruction classes

may expand to contain other classes, and that when fully expanded, a set of classes (e.g., the readers of some resource) may contain the same instruction multiple times.

- The syntax '**x\y**' where **x** and **y** are both instruction classes, indicates an unnamed instruction class that includes all instructions in instruction class **x** but that are not in instruction class **y**. Similarly, the notation '**x\y\z**' means all instructions in instruction class **x**, but that are not in either instruction class **y** or instruction class **z**.
- Resources on separate rows of a table are independent resources. This means that there are no serialization requirements for an event which references one of them followed by an event which uses a different resource. In cases where resources are broken into subrows, dependencies only apply between instructions within a subrow. Instructions that do not appear in a subrow together have no dependencies (reader/writer or writer/writer dependencies) for the resource in question, although they may still have dependencies on some other resource.
- The dependencies listed for pairs of instructions on each resource are not unique – the same pair of instructions might also have a dependency on some other resource with a different semantics of dependency. In cases where there are multiple resource dependencies for the same pair of instructions, the most stringent semantics are assumed: *instr* overrides *data* which overrides *impliedF* which overrides *implied* which overrides *none*.
- Arrays of numbered resources are represented in a single row of a table using the % notation as a substitute for the number of the resource. In such cases, the semantics of the table are as if each numbered resource had its own row in that table and is thus an independent resource. The range of values that the % can take are given in the "Resource Name" column.
- An asterisk '\*' in the "Resource Name" column indicates that this resource may not have a physical resource associated with it, but is added to enforce special dependencies.
- A pound sign '#' in the "Resource Name" column indicates that this resource is an array of resources that are indexed by a value in a GR. The number of individual elements in the array is described in the detailed description of each resource.
- The "Semantics of Dependency" column describes the outcome given various serialization and instruction group boundary conditions. The exact definition for each keyword is given in [Table 5-1](#).

**Table 5-1. Semantics of Dependency Codes**

Semantics of Dependency Code	Serialization Type Required	Effects of Serialization Violation
instr	Instruction Serialization (See " <a href="#">Instruction Serialization</a> " on page 2:18).	Atomic: Any attempt to read a resource after one or more insufficiently serialized writes is either the value previously in the register (before any of the unserialized writes) or the value of one of any unserialized writes. Which value is returned is unpredictable and multiple insufficiently serialized reads may see different results. No fault will be caused by the insufficient serialization.
data	Data Serialization (See " <a href="#">Data Serialization</a> " on page 2:18)	
implied	Instruction Group Break. Writer and reader must be in separate instruction groups. (See " <a href="#">Instruction Sequencing Considerations</a> " on page 1:39).	

**Table 5-1. Semantics of Dependency Codes (Continued)**

Semantics of Dependency Code	Serialization Type Required	Effects of Serialization Violation
impliedF	Instruction Group Break (same as above).	
stop	Stop. Writer and reader must be separated by a stop.	An undefined value is returned, or an Illegal Operation fault may be taken. If no fault is taken, the value returned is unpredictable, and may be unrelated to past writes, but will not be data which could not be accessed by the current process (e.g., if PSR.cpl != 0, the undefined value to return cannot be read from some control register).
none	None	N/A
specific	Implementation Specific	
SC	Special Case	Described elsewhere in book, see referenced section in the entry.

### 5.3.1 Special Case Instruction Rules

The following rules apply to the specified instructions when they appear in [Table 5-2](#), [Table 5-3](#), [Table 5-4](#), or [Table 5-5](#):

- An instruction always reads a given resource if its qualifying predicate is 1 and it appears in the “Reader” column of the table (except as noted). An instruction always writes a given resource if its qualifying predicate is 1 and it appears in the “Writer” column of the table (except as noted). An instruction never reads or writes the specified resource if its qualifying predicate is 0 (except as noted). These rules include branches and their qualifying predicate. Instructions in the **unpredictable-instructions** class have no qualifying predicate and thus always read or write their resources (except as noted).
- An instruction of type **mov-from-PR** reads all PRs if its PR[qp] is true. If the PR[qp] is false, then only the PR[qp] is read.
- An instruction of type **mov-to-PR** writes only those PRs as indicated by the immediate mask encoded in the instruction.
- A **st8.spill** only writes AR[UNAT]{X} where X equals the value in bits 8:3 of the store’s data address. A **ld8.fill** instruction only reads AR[UNAT]{Y} where Y equals the value in bits 8:3 of the load’s data address.
- Instructions of type **mod-sched-brs** always read AR[EC] and the rotating register base registers in CFM, and always write AR[EC], the rotating register bases in CFM, and PR[63] even if they do not change their values or if their PR[qp] is false.
- Instructions of type **mod-sched-brs-counted** always read and write AR[LC], even if they do not change its value.
- For instructions of type **pr-or-writers** or **pr-and-writers**, if their completer is **or.andcm**, then only the first target predicate is an or-compare and the second target predicate is an and-compare. Similarly, if their completer is **and.orcm**, then only the second target predicate is an or-compare and the first target predicate is an and-compare.
- **rum** and **sum** only read PSR.sp when the bit corresponding to PSR.up (bit 2) is set in the immediate field of the instruction.

### 5.3.2 RAW Dependency Table

[Table 5-2](#) architecturally defines the following information:

- A list of all architecturally-defined, independently-writable resources in the Itanium architecture. Each row represents an ‘atomic’ resource. Thus, for each row in the table, hardware will probably require a separate write-enable control signal.
- For each resource, a complete list of readers and writers.
- For each instruction, a complete list of all resources read and written. Such a list can be obtained by taking the union of all the rows in which each instruction appears.

**Table 5-2. RAW Dependencies Organized by Resource**

Resource Name	Writers	Readers	Semantics of Dependency
ALAT	chk.a.clr, <b>mem-readers-alat,</b> <b>mem-writers, invala-all</b>	<b>mem-readers-alat,</b> <b>mem-writers, chk-a,</b> invala.e	none
AR[BSP]	br.call, brl.call, br.ret, cover, <b>mov-to-AR-BSPSTORE</b> , rfi	br.call, brl.call, br.ia, br.ret, cover, flushrs, loadrs, <b>mov-from-AR-BSP</b> , rfi	impliedF
AR[BSPSTORE]	alloc, loadrs, flushrs, <b>mov-to-AR-BSPSTORE</b>	alloc, br.ia, flushrs, <b>mov-from-AR-BSPSTORE</b>	impliedF
AR[CCV]	<b>mov-to-AR-CCV</b>	br.ia, <b>cpxchq,</b> <b>mov-from-AR-CCV</b>	impliedF
AR[CFLG]	<b>mov-to-AR-CFLG</b>	br.ia, <b>mov-from-AR-CFLG</b>	impliedF
AR[CSD]	ld16, <b>mov-to-AR-CSD</b>	br.ia, cmp8xchg16, <b>mov-from-AR-CSD</b> , st16	impliedF
AR[EC]	<b>mod-sched-brs</b> , br.ret, <b>mov-to-AR-EC</b>	br.call, brl.call, br.ia, <b>mod-sched-brs</b> , <b>mov-from-AR-EC</b>	impliedF
AR[EFLAG]	<b>mov-to-AR-EFLAG</b>	br.ia, <b>mov-from-AR-EFLAG</b>	impliedF
AR[FCR]	<b>mov-to-AR-FCR</b>	br.ia, <b>mov-from-AR-FCR</b>	impliedF
AR[FDR]	<b>mov-to-AR-FDR</b>	br.ia, <b>mov-from-AR-FDR</b>	impliedF
AR[FIR]	<b>mov-to-AR-FIR</b>	br.ia, <b>mov-from-AR-FIR</b>	impliedF
AR[FPSR].sf0.controls	<b>mov-to-AR-FPSR</b> , fsetc.s0	br.ia, <b>fp-arith-s0</b> , <b>fcmp-s0</b> , <b>fpcmp-s0</b> , fsetc, <b>mov-from-AR-FPSR</b>	impliedF
AR[FPSR].sf1.controls	<b>mov-to-AR-FPSR</b> , fsetc.s1	br.ia, <b>fp-arith-s1</b> , <b>fcmp-s1</b> , <b>fpcmp-s1</b> , <b>mov-from-AR-FPSR</b>	
AR[FPSR].sf2.controls	<b>mov-to-AR-FPSR</b> , fsetc.s2	br.ia, <b>fp-arith-s2</b> , <b>fcmp-s2</b> , <b>fpcmp-s2</b> , <b>mov-from-AR-FPSR</b>	
AR[FPSR].sf3.controls	<b>mov-to-AR-FPSR</b> , fsetc.s3	br.ia, <b>fp-arith-s3</b> , <b>fcmp-s3</b> , <b>fpcmp-s3</b> , <b>mov-from-AR-FPSR</b>	
AR[FPSR].sf0.flags	<b>fp-arith-s0</b> , fclrf.s0, <b>fcmp-s0</b> , <b>fpcmp-s0</b> , <b>mov-to-AR-FPSR</b>	br.ia, fchkf, <b>mov-from-AR-FPSR</b>	impliedF
AR[FPSR].sf1.flags	<b>fp-arith-s1</b> , fclrf.s1, <b>fcmp-s1</b> , <b>fpcmp-s1</b> , <b>mov-to-AR-FPSR</b>	br.ia, fchkf.s1, <b>mov-from-AR-FPSR</b>	
AR[FPSR].sf2.flags	<b>fp-arith-s2</b> , fclrf.s2, <b>fcmp-s2</b> , <b>fpcmp-s2</b> , <b>mov-to-AR-FPSR</b>	br.ia, fchkf.s2, <b>mov-from-AR-FPSR</b>	
AR[FPSR].sf3.flags	<b>fp-arith-s3</b> , fclrf.s3, <b>fcmp-s3</b> , <b>fpcmp-s3</b> , <b>mov-to-AR-FPSR</b>	br.ia, fchkf.s3, <b>mov-from-AR-FPSR</b>	
AR[FPSR].traps	<b>mov-to-AR-FPSR</b>	br.ia, <b>fp-arith</b> , fchkf, fcmp, fpcmp, <b>mov-from-AR-FPSR</b>	impliedF
AR[FPSR].rv	<b>mov-to-AR-FPSR</b>	br.ia, <b>fp-arith</b> , fchkf, fcmp, fpcmp, <b>mov-from-AR-FPSR</b>	impliedF
AR[FSR]	<b>mov-to-AR-FSR</b>	br.ia, <b>mov-from-AR-FSR</b>	impliedF

**Table 5-2. RAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers	Readers	Semantics of Dependency
AR[ITC]	<b>mov-to-AR-ITC</b>	br.ia, <b>mov-from-AR-ITC</b>	impliedF
AR[K%], % in 0 - 7	<b>mov-to-AR-K<sup>1</sup></b>	br.ia, <b>mov-from-AR-K<sup>1</sup></b>	impliedF
AR[LC]	<b>mod-sched-brs-counted, mov-to-AR-LC</b>	br.ia, <b>mod-sched-brs-counted, mov-from-AR-LC</b>	impliedF
AR[PFS]	br.call, brl.call	alloc, br.ia, br.ret, epc, <b>mov-from-AR-PFS</b>	impliedF
	<b>mov-to-AR-PFS</b>	alloc, br.ia, epc, <b>mov-from-AR-PFS</b>	impliedF
		br.ret	none
AR[RNAT]	alloc, flushrs, loadrs, <b>mov-to-AR-RNAT, mov-to-AR-BSPSTORE</b>	alloc, br.ia, flushrs, loadrs, <b>mov-from-AR-RNAT</b>	impliedF
AR[RSC]	<b>mov-to-AR-RSC</b>	alloc, br.ia, flushrs, loadrs, <b>mov-from-AR-RSC, mov-from-AR-BSPSTORE, mov-to-AR-RNAT, mov-from-AR-RNAT, mov-to-AR-BSPSTORE</b>	impliedF
AR[RUC]	<b>mov-to-AR-RUC</b>	br.ia, <b>mov-from-AR-RUC</b>	impliedF
AR[SSD]	<b>mov-to-AR-SSD</b>	br.ia, <b>mov-from-AR-SSD</b>	impliedF
AR[UNAT]{%}, % in 0 - 63	<b>mov-to-AR-UNAT, st8.spill</b>	br.ia, ld8.fill, <b>mov-from-AR-UNAT</b>	impliedF
AR%, % in 8-15, 20, 22-23, 31, 33-35, 37-39, 41-43, 46-47, 67-111	<b>none</b>	br.ia, <b>mov-from-AR-rv<sup>1</sup></b>	none
AR%, % in 48-63, 112-127	<b>mov-to-AR-ig<sup>1</sup></b>	br.ia, <b>mov-from-AR-ig<sup>1</sup></b>	impliedF
BR%, % in 0 - 7	br.call <sup>1</sup> , brl.call <sup>1</sup>	<b>indirect-brs<sup>1</sup>, indirect-brp<sup>1</sup>, mov-from-BR<sup>1</sup></b>	impliedF
	<b>mov-to-BR<sup>1</sup></b>	<b>indirect-brs<sup>1</sup></b>	none
		<b>indirect-brp<sup>1</sup>, mov-from-BR<sup>1</sup></b>	impliedF
CFM	<b>mod-sched-brs</b>	<b>mod-sched-brs</b>	impliedF
		cover, alloc, rfi, loadrs, br.ret, br.call, brl.call	impliedF
		<b>cfm-readers<sup>2</sup></b>	impliedF
	br.call, brl.call, br.ret, clrrb, cover, rfi	<b>cfm-readers</b>	impliedF
CPUID#	<b>alloc</b>	<b>cfm-readers</b>	none
	<b>none</b>	<b>mov-from-IND-CPUID<sup>3</sup></b>	specific
	<b>mov-to-CR-CMCV</b>	<b>mov-from-CR-CMCV</b>	data
CR[DCR]	<b>mov-to-CR-DCR</b>	<b>mov-from-CR-DCR, mem-readers-spec</b>	data

**Table 5-2. RAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers	Readers	Semantics of Dependency
CR[EOI]	<b>mov-to-CR-EOI</b>	<b>none</b>	SC Section 5.8.3.4, “End of External Interrupt Register (EOI – CR67)” on <a href="#">page 2:124</a>
CR[IFA]	<b>mov-to-CR-IFA</b>	itc.i, itc.d, itr.i, itr.d <b>mov-from-CR-IFA</b>	implied data
CR[IFS]	<b>mov-to-CR-IFS</b>	<b>mov-from-CR-IFS</b> rfi	data implied
	cover	<b>rfi, mov-from-CR-IFS</b>	implied
CR[IHA]	<b>mov-to-CR-IHA</b>	<b>mov-from-CR-IHA</b>	data
CR[IIB%], % in 0 - 1	<b>mov-to-CR-IIB</b>	<b>mov-from-CR-IIB</b>	data
CR[IIM]	<b>mov-to-CR-IIM</b>	<b>mov-from-CR-IIM</b>	data
CR[IIP]	<b>mov-to-CR-IIP</b>	<b>mov-from-CR-IIP</b> rfi	data implied
CR[IIPA]	<b>mov-to-CR-IIPA</b>	<b>mov-from-CR-IIPA</b>	data
CR[IPSR]	<b>mov-to-CR-IPSR</b>	<b>mov-from-CR-IPSR</b> rfi	data implied
CR[IRR%], % in 0 - 3	<b>mov-from-CR-IVR</b>	<b>mov-from-CR-IRR</b> <sup>1</sup>	data
CR[ISR]	<b>mov-to-CR-ISR</b>	<b>mov-from-CR-ISR</b>	data
CR[ITIR]	<b>mov-to-CR-ITIR</b>	<b>mov-from-CR-ITIR</b> itc.i, itc.d, itr.i, itr.d	data implied
CR[ITM]	<b>mov-to-CR-ITM</b>	<b>mov-from-CR-ITM</b>	data
CR[ITO]	<b>mov-to-CR-ITO</b>	<b>mov-from-AR-ITC, mov-from-CR-ITO</b>	data
CR[ITV]	<b>mov-to-CR-ITV</b>	<b>mov-from-CR-ITV</b>	data
CR[IVA]	<b>mov-to-CR-IVA</b>	<b>mov-from-CR-IVA</b>	instr
CR[IVR]	<b>none</b>	<b>mov-from-CR-IVR</b>	SC Section 5.8.3.2, “External Interrupt Vector Register (IVR – CR65)” on <a href="#">page 2:123</a>
CR[LID]	<b>mov-to-CR-LID</b>	<b>mov-from-CR-LID</b>	SC Section 5.8.3.1, “Local ID (LID – CR64)” on <a href="#">page 2:122</a>
CR[LRR%], % in 0 - 1	<b>mov-to-CR-LRR</b> <sup>1</sup>	<b>mov-from-CR-LRR</b> <sup>1</sup>	data
CR[PMV]	<b>mov-to-CR-PMV</b>	<b>mov-from-CR-PMV</b>	data
CR[PTA]	<b>mov-to-CR-PTA</b>	<b>mov-from-CR-PTA, mem-readers, mem-writers, non-access, thash</b>	data

**Table 5-2. RAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers	Readers	Semantics of Dependency
CR[TPR]	<b>mov-to-CR-TPR</b>	<b>mov-from-CR-TPR,</b> <b>mov-from-CR-IVR</b>	data
		<b>mov-to-PSR-I</b> <sup>17</sup> , <b>ssm</b> <sup>17</sup>	SC Section 5.8.3.3, "Task Priority Register (TPR – CR66)" on page 2:123
		rfi	implied
CR%, % in 3, 5-7, 10-15, 18, 28-63, 75-79, 82-127	<b>none</b>	<b>mov-from-CR-rv</b> <sup>1</sup>	none
DBR#	<b>mov-to-IND-DBR</b> <sup>3</sup>	<b>mov-from-IND-DBR</b> <sup>3</sup>	impliedF
		<b>probe-all, Ifetch-all, mem-readers, mem-writers</b>	data
DTC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	<b>mem-readers, mem-writers, non-access</b>	data
	itc.i, itc.d, itr.i, itr.d	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	impliedF
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	none
		itc.i, itc.d, itr.i, itr.d	impliedF
DTC_LIMIT*	ptc.g, ptc.ga	ptc.g, ptc.ga	impliedF
DTR	itr.d	<b>mem-readers, mem-writers, non-access</b>	data
		ptc.g, ptc.ga, ptc.l, ptr.d, itr.d	impliedF
	ptr.d	<b>mem-readers, mem-writers, non-access</b>	data
		ptc.g, ptc.ga, ptc.l, ptr.d	none
		itr.d, itc.d	impliedF
FR%, % in 0 - 1	<b>none</b>	<b>fr-readers</b> <sup>1</sup>	none
FR%, % in 2 - 127	<b>fr-writers</b> <sup>1</sup> \ <b>ldf-c</b> <sup>1</sup> \ <b>ldfp-c</b> <sup>1</sup>	<b>fr-readers</b> <sup>1</sup>	impliedF
	<b>ldf-c</b> <sup>1</sup> , <b>ldfp-c</b> <sup>1</sup>	<b>fr-readers</b> <sup>1</sup>	none
GR0	<b>none</b>	<b>gr-readers</b> <sup>1</sup>	none
GR%, % in 1 - 127	<b>ld-c</b> <sup>1,13</sup>	<b>gr-readers</b> <sup>1</sup>	none
	<b>gr-writers</b> <sup>1</sup> \ <b>ld-c</b> <sup>1,13</sup>	<b>gr-readers</b> <sup>1</sup>	impliedF
IBR#	<b>mov-to-IND-IBR</b> <sup>3</sup>	<b>mov-from-IND-IBR</b> <sup>3</sup>	impliedF
InService*	<b>mov-to-CR-EOI</b>	<b>mov-from-CR-IVR</b>	data
	<b>mov-from-CR-IVR</b>	<b>mov-from-CR-IVR</b>	impliedF
	<b>mov-to-CR-EOI</b>	<b>mov-to-CR-EOI</b>	impliedF
IP	<b>all</b>	<b>all</b>	none
ITC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	epc, vmsw	instr
		itc.i, itc.d, itr.i, itr.d	impliedF
		ptr.i, ptr.d, ptc.e, ptc.g, ptc.ga, ptc.l	none
	itc.i, itc.d, itr.i, itr.d	epc, vmsw	instr
		itc.d, itc.i, itr.d, itr.i, ptr.d, ptr.i, ptc.g, ptc.ga, ptc.l	impliedF
ITC_LIMIT*	ptc.g, ptc.ga	ptc.g, ptc.ga	impliedF

**Table 5-2. RAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers	Readers	Semantics of Dependency
ITR	itr.i	itr.i, itc.i, ptc.g, ptc.ga, ptc.l, ptr.i	impliedF
		epc, vmsw	instr
	ptr.i	itc.i, itr.i	impliedF
		ptc.g, ptc.ga, ptc.l, ptr.i	none
		epc, vmsw	instr
		<b>mem-readers</b>	none
memory	<b>mem-writers</b>		
PKR#	<b>mov-to-IND-PKR<sup>3</sup></b>	<b>mem-readers, mem-writers, mov-from-IND-PKR<sup>4</sup>, probe-all</b>	data
		<b>mov-to-IND-PKR<sup>4</sup></b>	none
		<b>mov-from-IND-PKR<sup>3</sup></b>	impliedF
		<b>mov-to-IND-PKR<sup>3</sup></b>	impliedF
PMC#	<b>mov-to-IND-PMC<sup>3</sup></b>	<b>mov-from-IND-PMC<sup>3</sup></b>	impliedF
		<b>mov-from-IND-PMD<sup>3</sup></b>	SC Section 7.2.1, "Generic Performance Counter Registers" for PMC[0].fr on <a href="#">page 2:156</a>
PMD#	<b>mov-to-IND-PMD<sup>3</sup></b>	<b>mov-from-IND-PMD<sup>3</sup></b>	impliedF
PR0	<b>pr-writers<sup>1</sup></b>	<b>pr-readers-br<sup>1</sup>, pr-readers-nobr-nomovpr<sup>1</sup>, mov-from-PR<sup>12</sup>, mov-to-PR<sup>12</sup></b>	none
PR%, % in 1 - 15	<b>pr-writers<sup>1</sup>, mov-to-PR-allreg<sup>7</sup></b>	<b>pr-readers-nobr-nomovpr<sup>1</sup>, mov-from-PR, mov-to-PR<sup>12</sup></b>	impliedF
	<b>pr-writers-fp<sup>1</sup></b>	<b>pr-readers-br<sup>1</sup></b>	impliedF
	<b>pr-writers-int<sup>1</sup>, mov-to-PR-allreg<sup>7</sup></b>	<b>pr-readers-br<sup>1</sup></b>	none
PR%, % in 16 - 62	<b>pr-writers<sup>1</sup>, mov-to-PR-allreg<sup>7</sup>, mov-to-PR-rotreg</b>	<b>pr-readers-nobr-nomovpr<sup>1</sup>, mov-from-PR, mov-to-PR<sup>12</sup></b>	impliedF
	<b>pr-writers-fp<sup>1</sup></b>	<b>pr-readers-br<sup>1</sup></b>	impliedF
	<b>pr-writers-int<sup>1</sup>, mov-to-PR-allreg<sup>7</sup>, mov-to-PR-rotreg</b>	<b>pr-readers-br<sup>1</sup></b>	none
PR63	<b>mod-sched-brs, pr-writers<sup>1</sup>, mov-to-PR-allreg<sup>7</sup>, mov-to-PR-rotreg</b>	<b>pr-readers-nobr-nomovpr<sup>1</sup>, mov-from-PR, mov-to-PR<sup>12</sup></b>	impliedF
	<b>pr-writers-fp<sup>1</sup>, mod-sched-brs</b>	<b>pr-readers-br<sup>1</sup></b>	impliedF
	<b>pr-writers-int<sup>1</sup>, mov-to-PR-allreg<sup>7</sup>, mov-to-PR-rotreg</b>	<b>pr-readers-br<sup>1</sup></b>	none

**Table 5-2. RAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers	Readers	Semantics of Dependency
PSR.ac	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um	mem-readers, mem-writers	implied
	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I	mem-readers, mem-writers	data
	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um, sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I	mov-from-PSR, mov-from-PSR-um	impliedF
	rfi	mem-readers, mem-writers, mov-from-PSR, mov-from-PSR-um	impliedF
PSR.be	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um	mem-readers, mem-writers	implied
	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I	mem-readers, mem-writers	data
	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um, sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I	mov-from-PSR, mov-from-PSR-um	impliedF
	rfi	mem-readers, mem-writers, mov-from-PSR, mov-from-PSR-um	impliedF
PSR.bn	bsw, rfi	gr-readers <sup>10</sup> , gr-writers <sup>10</sup>	impliedF
PSR.cpl	epc, br.ret	priv-ops, br.call, bri.call, epc, mov-from-AR-ITC, mov-from-AR-RUC, mov-to-AR-ITC, mov-to-AR-RSC, mov-to-AR-RUC, mov-to-AR-K, mov-from-IND-PMD, probe-all, mem-readers, mem-writers, Ifetch-all	implied
	rfi	priv-ops, br.call, bri.call, epc, mov-from-AR-ITC, mov-from-AR-RUC, mov-to-AR-ITC, mov-to-AR-RSC, mov-to-AR-RUC, mov-to-AR-K, mov-from-IND-PMD, probe-all, mem-readers, mem-writers, Ifetch-all	impliedF
PSR.da	rfi	mem-readers, Ifetch-all, mem-writers, probe-fault	impliedF
PSR.db	mov-to-PSR-I	Ifetch-all, mem-readers, mem-writers, probe-fault	data
		mov-from-PSR	impliedF
	rfi	Ifetch-all, mem-readers, mem-writers, mov-from-PSR, probe-fault	impliedF
PSR.dd	rfi	Ifetch-all, mem-readers, probe-fault, mem-writers	impliedF

**Table 5-2. RAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers	Readers	Semantics of Dependency
PSR.dfh	<b>sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I</b>	<b>fr-readers<sup>8</sup>, fr-writers<sup>8</sup></b>	data
	rfi	<b>mov-from-PSR</b>	impliedF
		<b>fr-readers<sup>8</sup>, fr-writers<sup>8</sup>, mov-from-PSR</b>	impliedF
PSR.dfl	<b>sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I</b>	<b>fr-writers<sup>8</sup>, fr-readers<sup>8</sup></b>	data
	rfi	<b>mov-from-PSR</b>	impliedF
		<b>fr-writers<sup>8</sup>, fr-readers<sup>8</sup>, mov-from-PSR</b>	impliedF
PSR.di	<b>sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I</b>	br.ia	data
	rfi	<b>mov-from-PSR</b>	impliedF
		<b>br.ia, mov-from-PSR</b>	impliedF
PSR.dt	<b>sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I</b>	<b>mem-readers, mem-writers, non-access</b>	data
	rfi	<b>mov-from-PSR</b>	impliedF
		<b>mem-readers, mem-writers, non-access, mov-from-PSR</b>	impliedF
PSR.ed	rfi	<b>lfetch-all, mem-readers-spec</b>	impliedF
PSR.i	<b>sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I, rfi</b>	<b>mov-from-PSR</b>	impliedF
PSR.ia	rfi	<b>all</b>	none
PSR.ic	<b>sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I</b>	<b>mov-from-PSR</b>	impliedF
		<b>cover, itc.i, itc.d, itr.i, itr.d, mov-from-interruption-CR, mov-to-interruption-CR</b>	data
	rfi	<b>mov-from-PSR, cover, itc.i, itc.d, itr.i, itr.d, mov-from-interruption-CR, mov-to-interruption-CR</b>	impliedF
PSR.id	rfi	<b>all</b>	none
PSR.is	br.ia, rfi	<b>none</b>	none
PSR.it	rfi	<b>branches, mov-from-PSR, chk, epc, fchkf, vmsw</b>	impliedF
PSR.ip	<b>mov-to-PSR-I</b>	<b>mov-from-PSR</b>	impliedF
		<b>br.ret</b>	data
	rfi	<b>mov-from-PSR, br.ret</b>	impliedF
PSR.mc	rfi	<b>mov-from-PSR</b>	impliedF
PSR.mfh	<b>fr-writers<sup>9</sup>, user-mask-writers-partial<sup>7</sup>, mov-to-PSR-um, sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I, rfi</b>	<b>mov-from-PSR-um, mov-from-PSR</b>	impliedF
PSR.mfl	<b>fr-writers<sup>9</sup>, user-mask-writers-partial<sup>7</sup>, mov-to-PSR-um, sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I, rfi</b>	<b>mov-from-PSR-um, mov-from-PSR</b>	impliedF

**Table 5-2. RAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers	Readers	Semantics of Dependency
PSR.pk	<b>sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I</b>	<b>Ifetch-all, mem-readers, mem-writers, probe-all</b>	data
		<b>mov-from-PSR</b>	impliedF
	rfi	<b>Ifetch-all, mem-readers, mem-writers, mov-from-PSR, probe-all</b>	impliedF
PSR.pp	<b>sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I, rfi</b>	<b>mov-from-PSR</b>	impliedF
PSR.ri	rfi	<b>all</b>	none
PSR.rt	<b>mov-to-PSR-I</b>	<b>mov-from-PSR</b>	impliedF
		alloc, flushrs, loadrs	data
	rfi	<b>mov-from-PSR, alloc, flushrs, loadrs</b>	impliedF
PSR.si	<b>sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I</b>	<b>mov-from-PSR</b>	impliedF
		<b>mov-from-AR-ITC, mov-from-AR-RUC</b>	data
	rfi	<b>mov-from-AR-ITC, mov-from-AR-RUC, mov-from-PSR</b>	impliedF
PSR.sp	<b>sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I</b>	<b>mov-from-PSR</b>	impliedF
		<b>mov-from-IND-PMD, mov-to-PSR-um, rum, sum</b>	data
	rfi	<b>mov-from-IND-PMD, mov-from-PSR, mov-to-PSR-um, rum, sum</b>	impliedF
PSR.ss	rfi	<b>all</b>	impliedF
PSR.tb	<b>mov-to-PSR-I</b>	<b>branches, chk, fchkf</b>	data
		<b>mov-from-PSR</b>	impliedF
	rfi	<b>branches, chk, fchkf, mov-from-PSR</b>	impliedF
PSR.up	<b>user-mask-writers-partial<sup>7</sup>, mov-to-PSR-um, sys-mask-writers-partial<sup>7</sup>, mov-to-PSR-I, rfi</b>	<b>mov-from-PSR-um, mov-from-PSR</b>	impliedF
PSR.vm	vmsw	<b>mem-readers, mem-writers, mov-from-AR-ITC, mov-from-AR-RUC, mov-from-IND-CPUID, mov-to-AR-ITC, mov-to-AR-RUC, priv-ops\vmw, cover, thash, ttag</b>	implied
	rfi	<b>mem-readers, mem-writers, mov-from-AR-ITC, mov-from-AR-RUC, mov-from-IND-CPUID, mov-to-AR-ITC, mov-to-AR-RUC, priv-ops\vmw, cover, thash, ttag</b>	impliedF
RR#	<b>mov-to-IND-RR<sup>6</sup></b>	<b>mem-readers, mem-writers, itc.i, itc.d, itr.i, itr.d, non-access, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, thash, ttag</b>	data
		<b>mov-from-IND-RR<sup>6</sup></b>	impliedF
RSE	<b>rse-writers<sup>14</sup></b>	<b>rse-readers<sup>14</sup></b>	impliedF

### 5.3.3 WAW Dependency Table

General rules specific to the WAW table:

- All resources require at most an instruction group break to provide sequential behavior.
- Some resources require no instruction group break to provide sequential behavior.
- There are a few special cases that are described in greater detail elsewhere in the manual and are indicated with an SC (special case) result.
- Each sub-row of writers represents a group of instructions that when taken in pairs in any combination has the dependency result indicated. If the column is split in sub-columns, then the dependency semantics apply to any pair of instructions where one is chosen from left sub-column and one is chosen from the right sub-column.

**Table 5-3. WAW Dependencies Organized by Resource**

Resource Name	Writers		Semantics of Dependency
ALAT	<b>mem-readers-alat, mem-writers, chk.a.clr, invala-all</b>		none
AR[BSP]	br.call, brl.call, br.ret, cover, <b>mov-to-AR-BSPSTORE, rfi</b>		impliedF
AR[BSPSTORE]	alloc, loadrs, flushrs, <b>mov-to-AR-BSPSTORE</b>		impliedF
AR[CCV]	<b>mov-to-AR-CCV</b>		impliedF
AR[CFLG]	<b>mov-to-AR-CFLG</b>		impliedF
AR[CSD]	ld16, <b>mov-to-AR-CSD</b>		impliedF
AR[EC]	br.ret, <b>mod-sched-brs, mov-to-AR-EC</b>		impliedF
AR[EFLAG]	<b>mov-to-AR-EFLAG</b>		impliedF
AR[FCR]	<b>mov-to-AR-FCR</b>		impliedF
AR[FDR]	<b>mov-to-AR-FDR</b>		impliedF
AR[FIR]	<b>mov-to-AR-FIR</b>		impliedF
AR[FPSR].sf0.controls	<b>mov-to-AR-FPSR, fsetc.s0</b>		impliedF
AR[FPSR].sf1.controls	<b>mov-to-AR-FPSR, fsetc.s1</b>		impliedF
AR[FPSR].sf2.controls	<b>mov-to-AR-FPSR, fsetc.s2</b>		impliedF
AR[FPSR].sf3.controls	<b>mov-to-AR-FPSR, fsetc.s3</b>		impliedF
AR[FPSR].sf0.flags	<b>fp-arith-s0, fcmp-s0, fpcmp-s0</b>		none
	fclrf.s0, <b>fcmp-s0, fp-arith-s0, fpcmp-s0, mov-to-AR-FPSR</b>	fclrf.s0, <b>mov-to-AR-FPSR</b>	impliedF
AR[FPSR].sf1.flags	<b>fp-arith-s1, fcmp-s1, fpcmp-s1</b>		none
	fclrf.s1, <b>fcmp-s1, fp-arith-s1, fpcmp-s1, mov-to-AR-FPSR</b>	fclrf.s1, <b>mov-to-AR-FPSR</b>	impliedF
AR[FPSR].sf2.flags	<b>fp-arith-s2, fcmp-s2, fpcmp-s2</b>		none
	fclrf.s2, <b>fcmp-s2, fp-arith-s2, fpcmp-s2, mov-to-AR-FPSR</b>	fclrf.s2, <b>mov-to-AR-FPSR</b>	impliedF
AR[FPSR].sf3.flags	<b>fp-arith-s3, fcmp-s3, fpcmp-s3</b>		none
	fclrf.s3, <b>fcmp-s3, fp-arith-s3, fpcmp-s3, mov-to-AR-FPSR</b>	fclrf.s3, <b>mov-to-AR-FPSR</b>	impliedF
AR[FPSR].rv	<b>mov-to-AR-FPSR</b>		impliedF
AR[FPSR].traps	<b>mov-to-AR-FPSR</b>		impliedF
AR[FSR]	<b>mov-to-AR-FSR</b>		impliedF
AR[ITC]	<b>mov-to-AR-ITC</b>		impliedF

**Table 5-3. WAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers		Semantics of Dependency
AR[K%], % in 0 - 7	<b>mov-to-AR-K<sup>1</sup></b>		impliedF
AR[LC]	<b>mod-sched-brs-counted, mov-to-AR-LC</b>		impliedF
AR[PFS]	br.call, brl.call		none
	br.call, brl.call	<b>mov-to-AR-PFS</b>	impliedF
AR[RNAT]	alloc, flushrs, loadrs, <b>mov-to-AR-RNAT,</b> <b>mov-to-AR-BSPSTORE</b>		impliedF
AR[RSC]	<b>mov-to-AR-RSC</b>		impliedF
AR[RUC]	<b>mov-to-AR-RUC</b>		impliedF
AR[SSD]	<b>mov-to-AR-SSD</b>		impliedF
AR[UNAT]{%}, % in 0 - 63	<b>mov-to-AR-UNAT</b> , st8.spill		impliedF
AR%, % in 8-15, 20, 22-23, 31, 33-35, 37-39, 41-43, 46-47, 67-111	<b>none</b>		none
AR%, % in 48 - 63, 112-127	<b>mov-to-AR-ig<sup>1</sup></b>		impliedF
BR%, % in 0 - 7	br.call <sup>1</sup> , brl.call <sup>1</sup>	<b>mov-to-BR<sup>1</sup></b>	impliedF
	<b>mov-to-BR<sup>1</sup></b>		impliedF
	br.call <sup>1</sup> , brl.call <sup>1</sup>		none
CFM	<b>mod-sched-brs</b> , br.call, brl.call, br.ret, alloc, clrrrb, cover, rfi		impliedF
CPUID#	<b>none</b>		none
CR[CMCV]	<b>mov-to-CR-CMCV</b>		impliedF
CR[DCR]	<b>mov-to-CR-DCR</b>		impliedF
CR[EOI]	<b>mov-to-CR-EOI</b>		SC Section 5.8.3.4, "End of External Interrupt Register (EOI – CR67)" on <a href="#">page 2:124</a>
CR[IFA]	<b>mov-to-CR-IFA</b>		impliedF
CR[IFS]	<b>mov-to-CR-IFS</b> , cover		impliedF
CR[IHA]	<b>mov-to-CR-IHA</b>		impliedF
CR[IIB%], % in 0 - 1	<b>mov-to-CR-IIB</b>		impliedF
CR[IIM]	<b>mov-to-CR-IIM</b>		impliedF
CR[IIP]	<b>mov-to-CR-IIP</b>		impliedF
CR[IIPA]	<b>mov-to-CR-IIPA</b>		impliedF
CR[IPSR]	<b>mov-to-CR-IPSR</b>		impliedF
CR[IRR%], % in 0 - 3	<b>mov-from-CR-IVR</b>		impliedF
CR[ISR]	<b>mov-to-CR-ISR</b>		impliedF
CR[ITIR]	<b>mov-to-CR-ITIR</b>		impliedF
CR[ITM]	<b>mov-to-CR-ITM</b>		impliedF
CR[ITO]	<b>mov-to-CR-ITO</b>		impliedF

**Table 5-3. WAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers	Semantics of Dependency
CR[ITV]	<b>mov-to-CR-ITV</b>	impliedF
CR[IVA]	<b>mov-to-CR-IVA</b>	impliedF
CR[IVR]	<b>none</b>	SC
CR[LID]	<b>mov-to-CR-LID</b>	SC
CR[LRR%], % in 0 - 1	<b>mov-to-CR-LRR<sup>1</sup></b>	impliedF
CR[PMV]	<b>mov-to-CR-PMV</b>	impliedF
CR[PTA]	<b>mov-to-CR-PTA</b>	impliedF
CR[TPR]	<b>mov-to-CR-TPR</b>	impliedF
CR%, % in 3, 5-7, 10-15, 18, 28-63, 75-79, 82-127	<b>none</b>	none
DBR#	<b>mov-to-IND-DBR<sup>3</sup></b>	impliedF
DTC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	none
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	impliedF
DTC_LIMIT*	<b>ptc.g, ptc.ga</b>	impliedF
DTR	itr.d	impliedF
	itr.d   ptr.d	impliedF
	ptr.d	none
FR%, % in 0 - 1	<b>none</b>	none
FR%, % in 2 - 127	<b>fr-writers<sup>1</sup>, ldf-c<sup>1</sup>, ldfp-c<sup>1</sup></b>	impliedF
GR0	<b>none</b>	none
GR%, % in 1 - 127	<b>ld-c<sup>1</sup>, gr-writers<sup>1</sup></b>	impliedF
IBR#	<b>mov-to-IND-IBR<sup>3</sup></b>	impliedF
InService*	<b>mov-to-CR-EOI, mov-from-CR-IVR</b>	SC
IP	<b>all</b>	none
ITC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	none
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	impliedF
ITR	itr.i	impliedF
	ptr.i	none
memory	<b>mem-writers</b>	none
PKR#	<b>mov-to-IND-PKR<sup>3</sup></b>	<b>mov-to-IND-PKR<sup>4</sup></b>
	<b>mov-to-IND-PKR<sup>3</sup></b>	impliedF
PMC#	<b>mov-to-IND-PMC<sup>3</sup></b>	impliedF
PMD#	<b>mov-to-IND-PMD<sup>3</sup></b>	impliedF
PR0	<b>pr-writers<sup>1</sup></b>	none

**Table 5-3. WAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers		Semantics of Dependency
PR%, % in 1 - 15	pr-and-writers <sup>1</sup>		none
	pr-or-writers <sup>1</sup>		none
	pr-unc-writers-fp <sup>1</sup> , pr-unc-writers-int <sup>1</sup> , pr-norm-writers-fp <sup>1</sup> , pr-norm-writers-int <sup>1</sup> , pr-and-writers <sup>1</sup> , mov-to-PR-allreg <sup>7</sup>	pr-unc-writers-fp <sup>1</sup> , pr-unc-writers-int <sup>1</sup> , pr-norm-writers-fp <sup>1</sup> , pr-norm-writers-int <sup>1</sup> , pr-or-writers <sup>1</sup> , mov-to-PR-allreg <sup>7</sup>	impliedF
PR%, % in 16 - 62	pr-and-writers <sup>1</sup>		none
	pr-or-writers <sup>1</sup>		none
	pr-unc-writers-fp <sup>1</sup> , pr-unc-writers-int <sup>1</sup> , pr-norm-writers-fp <sup>1</sup> , pr-norm-writers-int <sup>1</sup> , pr-and-writers <sup>1</sup> , mov-to-PR-allreg <sup>7</sup> , mov-to-PR-rotreg	pr-unc-writers-fp <sup>1</sup> , pr-unc-writers-int <sup>1</sup> , pr-norm-writers-fp <sup>1</sup> , pr-norm-writers-int <sup>1</sup> , pr-or-writers <sup>1</sup> , mov-to-PR-allreg <sup>7</sup> , mov-to-PR-rotreg	impliedF
PR63	pr-and-writers <sup>1</sup>		none
	pr-or-writers <sup>1</sup>		none
	mod-sched-brs, pr-unc-writers-fp <sup>1</sup> , pr-unc-writers-int <sup>1</sup> , pr-norm-writers-fp <sup>1</sup> , pr-norm-writers-int <sup>1</sup> , pr-and-writers <sup>1</sup> , mov-to-PR-allreg <sup>7</sup> , mov-to-PR-rotreg	mod-sched-brs, pr-unc-writers-fp <sup>1</sup> , pr-unc-writers-int <sup>1</sup> , pr-norm-writers-fp <sup>1</sup> , pr-norm-writers-int <sup>1</sup> , pr-or-writers <sup>1</sup> , mov-to-PR-allreg <sup>7</sup> , mov-to-PR-rotreg	impliedF
PSR.ac	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um, sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.be	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um, sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.bn	bsw, rfi		impliedF
PSR.cpl	epc, br.ret, rfi		impliedF
PSR.da	rfi		impliedF
PSR.db	mov-to-PSR-I, rfi		impliedF
PSR.dd	rfi		impliedF
PSR.dfh	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.dfl	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.di	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.dt	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.ed	rfi		impliedF
PSR.i	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.ia	rfi		impliedF
PSR.ic	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.id	rfi		impliedF
PSR.is	br.ia, rfi		impliedF
PSR.it	rfi		impliedF
PSR.ip	mov-to-PSR-I, rfi		impliedF
PSR.mc	rfi		impliedF

**Table 5-3. WAW Dependencies Organized by Resource (Continued)**

Resource Name	Writers		Semantics of Dependency
PSR.mfh	fr-writers <sup>9</sup>		none
	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um, fr-writers <sup>9</sup> , sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um, sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi	impliedF
PSR.mfl	fr-writers <sup>9</sup>		none
	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um, fr-writers <sup>9</sup> , sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um, sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi	impliedF
PSR.pk	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.pp	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.ri	rfi		impliedF
PSR.rt	mov-to-PSR-I, rfi		impliedF
PSR.si	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.sp	sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.ss	rfi		impliedF
PSR.tb	mov-to-PSR-I, rfi		impliedF
PSR.up	user-mask-writers-partial <sup>7</sup> , mov-to-PSR-um, sys-mask-writers-partial <sup>7</sup> , mov-to-PSR-I, rfi		impliedF
PSR.vm	rfi, vmsw		impliedF
RR#	mov-to-IND-RR <sup>6</sup>		impliedF
RSE	rse-writers <sup>14</sup>		impliedF

### 5.3.4 WAR Dependency Table

A general rule specific to the WAR table:

1. WAR dependencies are always allowed within instruction groups except for the entry in [Table 5-4](#) below. The readers and subsequent writers specified must be separated by a stop in order to have defined behavior.

**Table 5-4. WAR Dependencies Organized by Resource**

Resource Name	Readers	Writers	Semantics of Dependency
PR63	pr-readers-br <sup>1</sup>	mod-sched-brs	stop

### 5.3.5 Listing of Rules Referenced in Dependency Tables

The following rules restrict the specific instances in which some of the instructions in the tables cause a dependency and must be applied where referenced to correctly interpret those entries. Rules only apply to the instance of the instruction class, or instruction mnemonic prefix where the rule is referenced as a superscript. If the rule is referenced in [Table 5-5](#) where instruction classes are defined, then it applies to all instances of the instruction class.

- Rule 1. These instructions only write a register when that register's number is explicitly encoded as a target of the instruction and is only read when it is encoded as a source of the instruction (or encoded as its PR[qp]).

- Rule 2. These instructions only read CFM when they access a rotating GR, FR, or PR. **mov-to-PR** and **mov-from-PR** only access CFM when their qualifying predicate is in the rotating region.
- Rule 3. These instructions use a general register value to determine the specific indirect register accessed. These instructions only access the register resource specified by the value in bits {7:0} of the dynamic value of the index register.
- Rule 4. These instructions only read the given resource when bits {7:0} of the indirect index register value *does not* match the register number of the resource.
- Rule 5. All rules are implementation specific.
- Rule 6. There is a dependency only when both the index specified by the reader and the index specified by the writer have the same value in bits {63:61}.
- Rule 7. These instructions access the specified resource only when the corresponding mask bit is set.
- Rule 8. PSR.dfh is only read when these instructions reference FR32-127. PSR.dfl is only read when these instructions reference FR2-31.
- Rule 9. PSR.mfl is only written when these instructions write FR2-31. PSR.mfh is only written when these instructions write FR32-127.
- Rule 10. The PSR.bn bit is only accessed when one of GR16-31 is specified in the instruction.
- Rule 11. The target predicates are written independently of PR[qp], but source registers are only read if PR[qp] is true.
- Rule 12. This instruction only reads the specified predicate register when that register is the PR[qp].
- Rule 13. This reference to Id-c only applies to the GR whose value is loaded with data returned from memory, not the post-incremented address register. Thus, a stop is still required between a post-incrementing Id-c and a consumer that reads the post-incremented GR.
- Rule 14. The RSE resource includes implementation-specific internal state. At least one (and possibly more) of these resources are read by each instruction listed in the **rse-readers** class. At least one (and possibly more) of these resources are written by each instruction listed in the **rse-writers** class. To determine exactly which instructions read or write each individual resource, see the corresponding instruction pages.
- Rule 15. This class represents all instructions marked as Reserved if PR[qp] is 1 B-type instructions as described in "[Format Summary](#)" on page 3:294.
- Rule 16. This class represents all instructions marked as Reserved if PR[qp] is 1 instructions as described in "[Format Summary](#)" on page 3:294.
- Rule 17. CR[TPR] has a RAW dependency only between **mov-to-CR-TPR** and **mov-to-PSR-I** or ssm instructions that set PSR.i, PSR.pp or PSR.up.

## **5.4 Support Tables**

**Table 5-5. Instruction Classes**

Class	Events/Instructions
all	<b>predicatable-instructions, unpredictabile-instructions</b>
branches	<b>indirect-brs, ip-rel-brs</b>
cfm-readers	<b>fr-readers, fr-writers, gr-readers, gr-writers, mod-sched-brs, predicatable-instructions, pr-writers</b> , alloc, br.call, brl.call, br.ret, cover, loadrs, rfi, <b>chk-a, invala.e</b>
chk-a	chk.a.clr, chk.a.nc
cmpxchg	cmpxchg1, cmpxchg2, cmpxchg4, cmpxchg8, cmp8xchg16
czx	czx1, czx2
fcmp-s0	fcmp[Field(sf)==s0]
fcmp-s1	fcmp[Field(sf)==s1]
fcmp-s2	fcmp[Field(sf)==s2]
fcmp-s3	fcmp[Field(sf)==s3]
fetchadd	fetchadd4, fetchadd8
fp-arith	fadd, famax, famin, fcvt.fx, fcvt.fxu, fcvt.xuf, fma, fmax, fmin, fmpy, fms, fnma, fnmpy, fnorm, fpamax, fpamin, fpctlx, fpctlxu, fpma, fpmax, fpmin, fpmpy, fpms, fpnma, fpnmpy, fprrcpa, fprrsqta, frcpa, fsqrta, fsub
fp-arith-s0	<b>fp-arith[Field(sf)==s0]</b>
fp-arith-s1	<b>fp-arith[Field(sf)==s1]</b>
fp-arith-s2	<b>fp-arith[Field(sf)==s2]</b>
fp-arith-s3	<b>fp-arith[Field(sf)==s3]</b>
fp-non-arith	fabs, fand, fandcm, fclass, fcvt.xf, fmerge, fmix, fneg, fnegabs, for, fpabs, fpmerge, fpack, fpneg, fpnegabs, fselect, fswap, fsxt, fxor, xma, xmpy
fpcmp-s0	fpcmp[Field(sf)==s0]
fpcmp-s1	fpcmp[Field(sf)==s1]
fpcmp-s2	fpcmp[Field(sf)==s2]
fpcmp-s3	fpcmp[Field(sf)==s3]
fr-readers	<b>fp-arith, fp-non-arith, mem-writers-fp, pr-writers-fp, chk.s[Format in {M21}], getf</b>
fr-writers	<b>fp-arith, fp-non-arith\fclass, mem-readers-fp, setf</b>
gr-readers	<b>gr-readers-writers, mem-readers, mem-writers, chk.s, cmp, cmp4, fc, itc.i, itc.d, itr.i, itr.d, mov-to-AR-gr, mov-to-BR, mov-to-CR, mov-to-IND, mov-from-IND, mov-to-PR-allreg, mov-to-PSR-I, mov-to-PSR-um, probe-all, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, setf, tbit, tnat</b>
gr-readers-writers	<b>mov-from-IND, add, addl, addp4, adds, and, andcm, clz, czx, dep\dep[Format in {I13}], extr, mem-readers-int, Id-all-postinc, lfetch-postinc, mix, mux, or, pack, padd, pavg, pavgsub, pcmp, pmax, pmin, pmpy, pmpysh, popcnt, probe-regular, psad, pshl, pshladd, pshr, pshrad, psub, shl, shladd, shladdp4, shr, shrp, st-postinc, sub, sxt, tak, thash, tpa, tttag, unpack, xor, zxt</b>
gr-writers	alloc, dep, getf, <b>gr-readers-writers, mem-readers-int, mov-from-AR, mov-from-BR, mov-from-CR, mov-from-PR, mov-from-PSR, mov-from-PSR-um, mov-ip, movl</b>
indirect-brp	brp[Format in {B7}]
indirect-brs	br.call[Format in {B5}], br.cond[Format in {B4}], br.ia, br.ret
invala-all	invala[Format in {M24}], invala.e
ip-rel-brs	<b>mod-sched-brs, br.call[Format in {B3}], brl.call, brl.cond, br.cond[Format in {B1}], br.cloop</b>
ld	ld1, ld2, ld4, ld8, ld8.fill, ld16
ld-a	ld1.a, ld2.a, ld4.a, ld8.a

**Table 5-5. Instruction Classes (Continued)**

Class	Events/Instructions
ld-all-postinc	<b>ld</b> [Format in {M2 M3}], <b>ldfp</b> [Format in {M12}], <b>ldf</b> [Format in {M7 M8}]
ld-c	<b>ld-c-nc</b> , <b>ld-c-clr</b>
ld-c-clr	ld1.c.clr, ld2.c.clr, ld4.c.clr, ld8.c.clr, <b>ld-c-clr-acq</b>
ld-c-clr-acq	ld1.c.clr.acq, ld2.c.clr.acq, ld4.c.clr.acq, ld8.c.clr.acq
ld-c-nc	ld1.c.nc, ld2.c.nc, ld4.c.nc, ld8.c.nc
ld-s	ld1.s, ld2.s, ld4.s, ld8.s
ld-sa	ld1.sa, ld2.sa, ld4.sa, ld8.sa
ldf	ldfs, ldfd, ldfe, ldf8, ldf.fill
ldf-a	ldfs.a, ldfd.a, ldfe.a, ldf8.a
ldf-c	<b>ldf-c-nc</b> , <b>ldf-c-clr</b>
ldf-c-clr	ldfs.c.clr, ldfd.c.clr, ldfe.c.clr, ldf8.c.clr
ldf-c-nc	ldfs.c.nc, ldfd.c.nc, ldfe.c.nc, ldf8.c.nc
ldf-s	ldfs.s, ldfd.s, ldfe.s, ldf8.s
ldf-sa	ldfs.sa, ldfd.sa, ldfe.sa, ldf8.sa
ldfp	ldfps, ldfpd, ldfp8
ldfp-a	ldfps.a, ldfpd.a, ldfp8.a
ldfp-c	<b>ldfp-c-nc</b> , <b>ldfp-c-clr</b>
ldfp-c-clr	ldfps.c.clr, ldfpd.c.clr, ldfp8.c.clr
ldfp-c-nc	ldfps.c.nc, ldfpd.c.nc, ldfp8.c.nc
ldfp-s	ldfps.s, ldfpd.s, ldfp8.s
ldfp-sa	ldfps.sa, ldfpd.sa, ldfp8.sa
lfetch-all	lfetch
lfetch-fault	lfetch[Field(lftype)==fault]
lfetch-nofault	lfetch[Field(lftype)==]
lfetch-postinc	lfetch[Format in {M20 M22}]
mem-readers	<b>mem-readers-fp</b> , <b>mem-readers-int</b>
mem-readers-alat	<b>ld-a</b> , <b>ldf-a</b> , <b>ldfp-a</b> , <b>ld-sa</b> , <b>ldf-sa</b> , <b>ldfp-sa</b> , <b>ld-c</b> , <b>ldf-c</b> , <b>ldfp-c</b>
mem-readers-fp	<b>ldf</b> , <b>ldfp</b>
mem-readers-int	<b>cpxchg</b> , <b>fetchadd</b> , <b>xchg</b> , <b>ld</b>
mem-readers-spec	<b>ld-s</b> , <b>ld-sa</b> , <b>ldf-s</b> , <b>ldf-sa</b> , <b>ldfp-s</b> , <b>ldfp-sa</b>
mem-writers	<b>mem-writers-fp</b> , <b>mem-writers-int</b>
mem-writers-fp	<b>stf</b>
mem-writers-int	<b>cpxchg</b> , <b>fetchadd</b> , <b>xchg</b> , <b>st</b>
mix	mix1, mix2, mix4
mod-sched-brs	br.cexit, br.ctop, br.wexit, br.wtop
mod-sched-brs-counted	br.cexit, br.cloop, br.ctop
mov-from-AR	<b>mov-from-AR-M</b> , <b>mov-from-AR-I</b> , <b>mov-from-AR-IM</b>
mov-from-AR-BSP	<b>mov-from-AR-M</b> [Field(ar3) == BSP]
mov-from-AR-BSPSTORE	<b>mov-from-AR-M</b> [Field(ar3) == BSPSTORE]
mov-from-AR-CCV	<b>mov-from-AR-M</b> [Field(ar3) == CCV]
mov-from-AR-CFLG	<b>mov-from-AR-M</b> [Field(ar3) == CFLG]
mov-from-AR-CSD	<b>mov-from-AR-M</b> [Field(ar3) == CSD]
mov-from-AR-EC	<b>mov-from-AR-I</b> [Field(ar3) == EC]
mov-from-AR-EFLAG	<b>mov-from-AR-M</b> [Field(ar3) == EFLAG]
mov-from-AR-FCR	<b>mov-from-AR-M</b> [Field(ar3) == FCR]

**Table 5-5. Instruction Classes (Continued)**

Class	Events/Instructions
mov-from-AR-FDR	<b>mov-from-AR-M</b> [Field(ar3) == FDR]
mov-from-AR-FIR	<b>mov-from-AR-M</b> [Field(ar3) == FIR]
mov-from-AR-FPSR	<b>mov-from-AR-M</b> [Field(ar3) == FPSR]
mov-from-AR-FSR	<b>mov-from-AR-M</b> [Field(ar3) == FSR]
mov-from-AR-I	mov_ar[Format in {I28}]
mov-from-AR-ig	<b>mov-from-AR-IM</b> [Field(ar3) in {48-63 112-127}]
mov-from-AR-IM	mov_ar[Format in {I28 M31}]
mov-from-AR-ITC	<b>mov-from-AR-M</b> [Field(ar3) == ITC]
mov-from-AR-K	<b>mov-from-AR-M</b> [Field(ar3) in {K0 K1 K2 K3 K4 K5 K6 K7}]
mov-from-AR-LC	<b>mov-from-AR-I</b> [Field(ar3) == LC]
mov-from-AR-M	mov_ar[Format in {M31}]
mov-from-AR-PFS	<b>mov-from-AR-I</b> [Field(ar3) == PFS]
mov-from-AR-RNAT	<b>mov-from-AR-M</b> [Field(ar3) == RNAT]
mov-from-AR-RSC	<b>mov-from-AR-M</b> [Field(ar3) == RSC]
mov-from-AR-RUC	<b>mov-from-AR-M</b> [Field(ar3) == RUC]
mov-from-AR-rv	none
mov-from-AR-SSD	<b>mov-from-AR-M</b> [Field(ar3) == SSD]
mov-from-AR-UNAT	<b>mov-from-AR-M</b> [Field(ar3) == UNAT]
mov-from-BR	mov_br[Format in {I22}]
mov-from-CR	mov_cr[Format in {M33}]
mov-from-CR-CMCV	<b>mov-from-CR</b> [Field(cr3) == CMCV]
mov-from-CR-DCR	<b>mov-from-CR</b> [Field(cr3) == DCR]
mov-from-CR-EOI	<b>mov-from-CR</b> [Field(cr3) == EOI]
mov-from-CR-IFA	<b>mov-from-CR</b> [Field(cr3) == IFA]
mov-from-CR-IFS	<b>mov-from-CR</b> [Field(cr3) == IFS]
mov-from-CR-IHA	<b>mov-from-CR</b> [Field(cr3) == IHA]
mov-from-CR-IIB	<b>mov-from-CR</b> [Field(cr3) in {IIB0 IIB1}]
mov-from-CR-IIM	<b>mov-from-CR</b> [Field(cr3) == IIM]
mov-from-CR-IIP	<b>mov-from-CR</b> [Field(cr3) == IIP]
mov-from-CR-IIPA	<b>mov-from-CR</b> [Field(cr3) == IIPA]
mov-from-CR-IPSR	<b>mov-from-CR</b> [Field(cr3) == IPSR]
mov-from-CR-IRR	<b>mov-from-CR</b> [Field(cr3) in {IRR0 IRR1 IRR2 IRR3}]
mov-from-CR-ISR	<b>mov-from-CR</b> [Field(cr3) == ISR]
mov-from-CR-ITIR	<b>mov-from-CR</b> [Field(cr3) == ITIR]
mov-from-CR-ITM	<b>mov-from-CR</b> [Field(cr3) == ITM]
mov-from-CR-ITO	<b>mov-from-CR</b> [Field(cr3) == ITO]
mov-from-CR-ITV	<b>mov-from-CR</b> [Field(cr3) == ITV]
mov-from-CR-IVA	<b>mov-from-CR</b> [Field(cr3) == IVA]
mov-from-CR-IVR	<b>mov-from-CR</b> [Field(cr3) == IVR]
mov-from-CR-LID	<b>mov-from-CR</b> [Field(cr3) == LID]
mov-from-CR-LRR	<b>mov-from-CR</b> [Field(cr3) in {LRR0 LRR1}]
mov-from-CR-PMV	<b>mov-from-CR</b> [Field(cr3) == PMV]
mov-from-CR-PTA	<b>mov-from-CR</b> [Field(cr3) == PTA]
mov-from-CR-rv	none
mov-from-CR-TPR	<b>mov-from-CR</b> [Field(cr3) == TPR]

**Table 5-5. Instruction Classes (Continued)**

Class	Events/Instructions
mov-from-IND	mov_indirect[Format in {M43}]
mov-from-IND-CPUID	<b>mov-from-IND</b> [Field(ireg) == cpuid]
mov-from-IND-DBR	<b>mov-from-IND</b> [Field(ireg) == dbr]
mov-from-IND-IBR	<b>mov-from-IND</b> [Field(ireg) == ibr]
mov-from-IND-PKR	<b>mov-from-IND</b> [Field(ireg) == pkr]
mov-from-IND-PMC	<b>mov-from-IND</b> [Field(ireg) == pmc]
mov-from-IND-PMD	<b>mov-from-IND</b> [Field(ireg) == pmd]
mov-from-IND-priv	<b>mov-from-IND</b> [Field(ireg) in {dbr ibr pkr pmc rr}]
mov-from-IND-RR	<b>mov-from-IND</b> [Field(ireg) == rr]
mov-from-interruption-CR	<b>mov-from-CR-ITIR</b> , <b>mov-from-CR-IFS</b> , <b>mov-from-CR-IIB</b> , <b>mov-from-CR-IIM</b> , <b>mov-from-CR-IIP</b> , <b>mov-from-CR-IPSR</b> , <b>mov-from-CR-ISR</b> , <b>mov-from-CR-IFA</b> , <b>mov-from-CR-IHA</b> , <b>mov-from-CR-IIPA</b>
mov-from-PR	mov_pr[Format in {I25}]
mov-from-PSR	mov_psrf[Format in {M36}]
mov-from-PSR-um	mov_um[Format in {M36}]
mov-ip	mov_ip[Format in {I25}]
mov-to-AR	<b>mov-to-AR-M</b> , <b>mov-to-AR-I</b>
mov-to-AR-BSP	<b>mov-to-AR-M</b> [Field(ar3) == BSP]
mov-to-AR-BSPSTORE	<b>mov-to-AR-M</b> [Field(ar3) == BSPSTORE]
mov-to-AR-CCV	<b>mov-to-AR-M</b> [Field(ar3) == CCV]
mov-to-AR-CFLG	<b>mov-to-AR-M</b> [Field(ar3) == CFLG]
mov-to-AR-CSD	<b>mov-to-AR-M</b> [Field(ar3) == CSD]
mov-to-AR-EC	<b>mov-to-AR-I</b> [Field(ar3) == EC]
mov-to-AR-EFLAG	<b>mov-to-AR-M</b> [Field(ar3) == EFLAG]
mov-to-AR-FCR	<b>mov-to-AR-M</b> [Field(ar3) == FCR]
mov-to-AR-FDR	<b>mov-to-AR-M</b> [Field(ar3) == FDR]
mov-to-AR-FIR	<b>mov-to-AR-M</b> [Field(ar3) == FIR]
mov-to-AR-FPSR	<b>mov-to-AR-M</b> [Field(ar3) == FPSR]
mov-to-AR-FSR	<b>mov-to-AR-M</b> [Field(ar3) == FSR]
mov-to-AR-gr	<b>mov-to-AR-M</b> [Format in {M29}], <b>mov-to-AR-I</b> [Format in {I26}]
mov-to-AR-I	mov_ar[Format in {I26 I27}]
mov-to-AR-ig	<b>mov-to-AR-IM</b> [Field(ar3) in {48-63 112-127}]
mov-to-AR-IM	mov_ar[Format in {I26 I27 M29 M30}]
mov-to-AR-ITC	<b>mov-to-AR-M</b> [Field(ar3) == ITC]
mov-to-AR-K	<b>mov-to-AR-M</b> [Field(ar3) in {K0 K1 K2 K3 K4 K5 K6 K7}]
mov-to-AR-LC	<b>mov-to-AR-I</b> [Field(ar3) == LC]
mov-to-AR-M	mov_ar[Format in {M29 M30}]
mov-to-AR-PFS	<b>mov-to-AR-I</b> [Field(ar3) == PFS]
mov-to-AR-RNAT	<b>mov-to-AR-M</b> [Field(ar3) == RNAT]
mov-to-AR-RSC	<b>mov-to-AR-M</b> [Field(ar3) == RSC]
mov-to-AR-RUC	<b>mov-to-AR-M</b> [Field(ar3) == RUC]
mov-to-AR-SSD	<b>mov-to-AR-M</b> [Field(ar3) == SSD]
mov-to-AR-UNAT	<b>mov-to-AR-M</b> [Field(ar3) == UNAT]
mov-to-BR	mov_br[Format in {I21}]
mov-to-CR	mov_cr[Format in {M32}]
mov-to-CR-CMCV	<b>mov-to-CR</b> [Field(cr3) == CMCV]

**Table 5-5. Instruction Classes (Continued)**

Class	Events/Instructions
mov-to-CR-DCR	<b>mov-to-CR</b> [Field(cr3) == DCR]
mov-to-CR-EOI	<b>mov-to-CR</b> [Field(cr3) == EOI]
mov-to-CR-IFA	<b>mov-to-CR</b> [Field(cr3) == IFA]
mov-to-CR-IFS	<b>mov-to-CR</b> [Field(cr3) == IFS]
mov-to-CR-IHA	<b>mov-to-CR</b> [Field(cr3) == IHA]
mov-to-CR-IIB	<b>mov-to-CR</b> [Field(cr3) in {IIB0 IIB1}]
mov-to-CR-IIM	<b>mov-to-CR</b> [Field(cr3) == IIM]
mov-to-CR-IIP	<b>mov-to-CR</b> [Field(cr3) == IIP]
mov-to-CR-IIPA	<b>mov-to-CR</b> [Field(cr3) == IIPA]
mov-to-CR-IPSR	<b>mov-to-CR</b> [Field(cr3) == IPSR]
mov-to-CR-IRR	<b>mov-to-CR</b> [Field(cr3) in {IRR0 IRR1 IRR2 IRR3}]
mov-to-CR-ISR	<b>mov-to-CR</b> [Field(cr3) == ISR]
mov-to-CR-ITIR	<b>mov-to-CR</b> [Field(cr3) == ITIR]
mov-to-CR-ITM	<b>mov-to-CR</b> [Field(cr3) == ITM]
mov-to-CR-ITO	<b>mov-to-CR</b> [Field(cr3) == ITO]
mov-to-CR-ITV	<b>mov-to-CR</b> [Field(cr3) == ITV]
mov-to-CR-IVA	<b>mov-to-CR</b> [Field(cr3) == IVA]
mov-to-CR-IVR	<b>mov-to-CR</b> [Field(cr3) == IVR]
mov-to-CR-LID	<b>mov-to-CR</b> [Field(cr3) == LID]
mov-to-CR-LRR	<b>mov-to-CR</b> [Field(cr3) in {LRR0 LRR1}]
mov-to-CR-PMV	<b>mov-to-CR</b> [Field(cr3) == PMV]
mov-to-CR-PTA	<b>mov-to-CR</b> [Field(cr3) == PTA]
mov-to-CR-TPR	<b>mov-to-CR</b> [Field(cr3) == TPR]
mov-to-IND	mov_indirect[Format in {M42}]
mov-to-IND-CPUID	<b>mov-to-IND</b> [Field(ireg) == cpuid]
mov-to-IND-DBR	<b>mov-to-IND</b> [Field(ireg) == dbr]
mov-to-IND-IBR	<b>mov-to-IND</b> [Field(ireg) == ibr]
mov-to-IND-PKR	<b>mov-to-IND</b> [Field(ireg) == pkr]
mov-to-IND-PMC	<b>mov-to-IND</b> [Field(ireg) == pmc]
mov-to-IND-PMD	<b>mov-to-IND</b> [Field(ireg) == pmd]
mov-to-IND-priv	<b>mov-to-IND</b>
mov-to-IND-RR	<b>mov-to-IND</b> [Field(ireg) == rr]
mov-to-interruption-CR	<b>mov-to-CR-ITIR</b> , <b>mov-to-CR-IFS</b> , <b>mov-to-CR-IIB</b> , <b>mov-to-CR-IIM</b> , <b>mov-to-CR-IIP</b> , <b>mov-to-CR-IPSR</b> , <b>mov-to-CR-ISR</b> , <b>mov-to-CR-IFA</b> , <b>mov-to-CR-IHA</b> , <b>mov-to-CR-IIPA</b>
mov-to-PR	<b>mov-to-PR-allreg</b> , <b>mov-to-PR-rotreg</b>
mov-to-PR-allreg	mov_pr[Format in {I23}]
mov-to-PR-rotreg	mov_pr[Format in {I24}]
mov-to-PSR-I	mov_psr[Format in {M35}]
mov-to-PSR-um	mov_um[Format in {M35}]
mux	mux1, mux2
non-access	fc, lfetch, <b>probe-all</b> , tpa, tak
none	-
pack	pack2, pack4
padd	padd1, padd2, padd4
pavg	pavg1, pavg2

**Table 5-5. Instruction Classes (Continued)**

Class	Events/Instructions
pavgsub	pavgsub1, pavgsub2
pcmp	pcmp1, pcmp2, pcmp4
pmax	pmax1, pmax2
pmin	pmin1, pmin2
pmpy	pmpy2
pmpyshr	pmpyshr2
pr-and-writers	<b>pr-gen-writers-int</b> [Field(ctype) in {and andcm}], <b>pr-gen-writers-int</b> [Field(ctype) in {or.andcm and.orcm}]
pr-gen-writers-fp	fclass, fcmp
pr-gen-writers-int	cmp, cmp4, tbit, tf, tnat
pr-norm-writers-fp	<b>pr-gen-writers-fp</b> [Field(ctype)==]
pr-norm-writers-int	<b>pr-gen-writers-int</b> [Field(ctype)==]
pr-or-writers	<b>pr-gen-writers-int</b> [Field(ctype) in {or orcm}], <b>pr-gen-writers-int</b> [Field(ctype) in {or.andcm and.orcm}]
pr-readers-br	br.call, br.cond, brl.call, brl.cond, br.ret, br.wexit, br.wtop, break.b, hint.b, nop.b, <b>ReservedBQP</b>
pr-readers-nobr-nomovpr	add, addl, addp4, adds, and, andcm, break.f, break.i, break.m, break.x, chk.s, <b>chk-a</b> , cmp, cmp4, <b>cpxchg</b> , clz, <b>czx</b> , dep, extr, <b>fp-arith</b> , <b>fp-non-arith</b> , fc, fchkf, fclrf, fcmp, <b>fetchadd</b> , fpcmp, fsetc, fwb, getf, hint.f, hint.i, hint.m, hint.x, <b>invala-all</b> , itc.i, itc.d, itr.i, itr.d, <b>Id</b> , <b>ldf</b> , <b>ldfp</b> , <b>lfetch-all</b> , mf, mix, <b>mov-from-AR-M</b> , <b>mov-from-AR-IM</b> , <b>mov-from-AR-I</b> , <b>mov-to-AR-M</b> , <b>mov-to-AR-I</b> , <b>mov-to-AR-IM</b> , <b>mov-to-BR</b> , <b>mov-from-BR</b> , <b>mov-to-CR</b> , <b>mov-from-CR</b> , <b>mov-to-IND</b> , <b>mov-from-IND</b> , <b>mov-ip</b> , <b>mov-to-PSR-I</b> , <b>mov-to-PSR-um</b> , <b>mov-from-PSR</b> , <b>mov-from-PSR-um</b> , movl, <b>mux</b> , nop.f, nop.i, nop.m, nop.x, or, <b>pack</b> , <b>padd</b> , <b>pavg</b> , <b>pavgsub</b> , <b>pcmp</b> , <b>pmax</b> , <b>pmin</b> , <b>pmpy</b> , <b>pmpyshr</b> , popcnt, <b>probe-all</b> , psad, <b>pshl</b> , <b>pshladd</b> , <b>pshr</b> , <b>pshradd</b> , <b>psub</b> , ptc.e, ptc.g, ptc.ga, ptc.l, ptr.d, ptr.i, <b>ReservedQP</b> , rsm, self, shl, shladd, shladdp4, shr, shrp, srlz.i, srlz.d, ssm, <b>st</b> , <b>stf</b> , sub, sum, <b>sxt</b> , sync, tak, tbit, tf, thash, tnat, tpa, ttag, <b>unpack</b> , <b>xchg</b> , xma, xmpy, xor, <b>zxt</b>
pr-unc-writers-fp	<b>pr-gen-writers-fp</b> [Field(ctype)==unc] <sup>11</sup> , fprcpa <sup>11</sup> , fprsqrt <sup>11</sup> , frcpa <sup>11</sup> , frsqrt <sup>11</sup>
pr-unc-writers-int	<b>pr-gen-writers-int</b> [Field(ctype)==unc] <sup>11</sup>
pr-writers	<b>pr-writers-int</b> , <b>pr-writers-fp</b>
pr-writers-fp	<b>pr-norm-writers-fp</b> , <b>pr-unc-writers-fp</b>
pr-writers-int	<b>pr-norm-writers-int</b> , <b>pr-unc-writers-int</b> , <b>pr-and-writers</b> , <b>pr-or-writers</b>
predicatable-instructions	<b>mov-from-PR</b> , <b>mov-to-PR</b> , <b>pr-readers-br</b> , <b>pr-readers-nobr-nomovpr</b>
priv-ops	<b>mov-to-IND-priv</b> , bsw, itc.i, itc.d, itr.i, itr.d, <b>mov-to-CR</b> , <b>mov-from-CR</b> , <b>mov-to-PSR-I</b> , <b>mov-from-PSR</b> , <b>mov-from-IND-priv</b> , ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, rfi, rsm, ssm, tak, tpa, vmsw
probe-all	<b>probe-fault</b> , <b>probe-regular</b>
probe-fault	probe[Format in {M40}]
probe-regular	probe[Format in {M38 M39}]
psad	psad1
pshl	pshl2, pshl4
pshladd	pshladd2
pshr	pshr2, pshr4
pshradd	pshradd2
psub	psub1, psub2, psub4
ReservedBQP	- <sup>15</sup>
ReservedQP	- <sup>16</sup>

**Table 5-5. Instruction Classes (Continued)**

Class	Events/Instructions
rse-readers	alloc, br.call, br.ia, br.ret, brl.call, cover, flushrs, loadrs, <b>mov-from-AR-BSP</b> , <b>mov-from-AR-BSPSTORE</b> , <b>mov-to-AR-BSPSTORE</b> , <b>mov-from-AR-RNAT</b> , <b>mov-to-AR-RNAT</b> , rfi
rse-writers	alloc, br.call, br.ia, br.ret, brl.call, cover, flushrs, loadrs, <b>mov-to-AR-BSPSTORE</b> , rfi
st	st1, st2, st4, st8, st8.spill, st16
st-postinc	<b>stf</b> [Format in {M10}], <b>st</b> [Format in {M5}]
stf	stfs, std, stfe, stf8, stf.spill
sxt	sxt1, sxt2, sxt4
sys-mask-writers-partial	rsm, ssm
unpack	unpack1, unpack2, unpack4
unpredicatable-instructions	alloc, br.cloop, br.ctop, br.cexit, br.ia, brp, bsw, clrrb, cover, epc, flushrs, loadrs, rfi, vmsw
user-mask-writers-partial	rum, sum
xchg	xchg1, xchg2, xchg4, xchg8
zxt	zxt1, zxt2, zxt4

§



## ***Index***



# INDEX FOR VOLUMES 1, 2, 3 AND 4

## A

AAA Instruction 4:21  
AAD Instruction 4:22  
AAM Instruction 4:23  
AAS Instruction 4:24  
Aborts 2:95, 2:538  
ACPI 2:631  
    P-states 2:315, 2:637  
Acquire Semantics 2:507  
ADC Instruction 4:25, 4:26  
ADD Instruction 4:27, 4:28  
add Instruction 3:14  
addp4 Instruction 3:15  
ADDP5 Instruction 4:486  
Address Space Model 2:561  
ADDSS Instruction 4:487  
Advanced Load 1:153, 1:154  
Advanced Load Address Table (ALAT) 1:64  
Advanced Load Check 1:154  
ALAT (Advanced Load Address Table) 1:64  
    Coherency 2:554  
        Data Speculation 1:17  
alloc Instruction 3:16  
AND Instruction 4:29, 4:30  
and Instruction 3:18  
andcm Instruction 3:19  
ANDNPS Instruction 4:488  
ANDPS Instruction 4:489  
Application Architecture Guide 1:1  
Application Memory Addressing Model 1:36  
Application Register (AR) 1:23, 1:28, 1:140  
AR (Application Register) 1:28, 1:140  
Arithmetic Instructions 1:51  
ARPL Instruction 4:31, 4:32

## B

Backing Store 2:133  
Banked General Registers 2:42  
Bit Field and Shift Instructions 1:52  
Bit Strings 1:84  
Boot Sequence 2:13  
BOUND Instruction 4:33  
BR (Branch Register) 1:26, 1:140  
br Instruction 3:20  
    br.ia 1:112, 2:596  
Branch Hints 1:78, 1:176  
Branch Instructions 1:74, 1:145  
Branch Register (BR) 1:19, 1:26, 1:140  
break Instruction 2:556, 3:29  
Break Instruction Fault 2:151  
bri Instruction 3:30  
brp Instruction 3:32  
BSF Instruction 4:35  
BSP (RSE Backing Store Pointer Register) 1:29  
BSPSTORE (RSE Backing Store Pointer for Memory

Stores Register) 1:30  
BSR Instruction 4:37  
bsw Instruction 3:34  
BSWAP Instruction 4:39  
BT Instruction 4:40  
BTC Instruction 4:42  
BTR Instruction 4:44  
BTS Instruction 4:46  
Bundle Format 1:38  
Bundles 1:38, 1:141  
Byte Ordering 1:36

## C

CALL Instruction 4:48  
CBW Instruction 4:57  
CCV (Compare and Exchange Value Register) 1:30  
CDQ Instruction 4:85  
CFM (Current Frame Marker) 1:27  
Character Strings 1:83  
Check Code 1:161  
Check Load 1:154  
chk Instruction 3:35  
CLC Instruction 4:59  
CLD Instruction 4:60  
CLI Instruction 4:61  
clrrrb Instruction 3:37  
CLTS Instruction 4:63  
clz Instruction 3:38  
CMC (Corrected Machine Check) 2:350  
CMC Instruction 4:64  
CMCV (Corrected Machine Check Vector) 2:126  
CMP Instruction 4:69  
cmp Instruction 3:39  
cmp4 Instruction 3:43  
CMPPS Instruction 4:490  
CMPS Instruction 4:71  
CMPSB Instruction 4:71  
CMPSD Instruction 4:71  
CMPSS Instruction 4:493  
CMPSW Instruction 4:71  
CMPXCHG Instruction 4:74  
cmpxchg Instruction 2:508, 3:46  
CMPXCHG8B Instruction 4:76  
Coalescing Attribute 2:78  
COMISS Instruction 4:496  
Compare and Exchange Value Register (CCV) 1:30  
Compare and Store Data Register (CSD) 1:30  
Compare Types 1:55  
Context Management 2:549  
Context Switching 2:557  
    Operating System Kernel 2:558  
    User-Level 2:557  
Control Dependencies 1:148  
Control Registers 2:29  
Control Speculation 1:16, 1:60, 1:142, 1:151,

1:155, 2:579  
Control Speculative Load 1:156  
Corrected Error 2:350  
Corrected Machine Check Vector (CMCV) 2:126  
cover Instruction 3:48  
CPUID (Processor Identification Register) 1:34  
CPUID Instruction 4:78  
Cross-modifying Code 2:533  
CSD (Compare and Store Data Register) 1:30  
Current Frame Marker (CFM) 1:27  
CVTPI2PS Instruction 4:498  
CVTPS2PI Instruction 4:500  
CVTSI2SS Instruction 4:502  
CVTSS2SI Instruction 4:503  
CVTTPS2PI Instruction 4:504  
CVTTSS2SI Instruction 4:506  
CWD Instruction 4:85  
CWDE Instruction 4:57, 4:86  
czx Instruction 3:49

**D**

DAA Instruction 4:87  
DAS Instruction 4:88  
Data Arrangement 1:81  
Data Breakpoint Register (DBR) 2:151, 2:152  
Data Debug Faults 2:152  
Data Dependencies 1:149, 1:150, 3:371  
Data Poisoning 2:302  
Data Prefetch Hint 1:148  
Data Serialization 2:18  
Data Speculation 1:17, 1:63, 1:143, 1:151, 2:579  
Data Speculative Load 1:154  
DBR (Data Breakpoint Register) 2:151, 2:152  
DCR (Default Control Register) 2:31  
Debugging 2:151  
DEC Instruction 4:89  
Default Control Register (DCR) 2:31  
Dekker's Algorithm 2:529  
dep Instruction 3:51  
DIV Instruction 4:91  
DIVPS Instruction 4:507  
DIVSS Instruction 4:508

**E**

EC (Epilog Count Register) 1:33  
EFLAG (IA-32 EFLAG Register) 1:123  
EMMS Instruction 4:400  
End of External Interrupt Register (EOI) 2:124  
Endian 1:36  
ENTER Instruction 4:94  
EOI (End of External Interrupt Register) 2:124  
epc Instruction 2:555, 3:53  
Epilog Count Register (EC) 1:33  
Explicit Prefetch 1:70  
External Controller Interrupts 2:96

External Interrupt 2:96, 2:538  
External Interrupt Control Registers (CR64-81)  
2:42  
External Interrupt Request Registers (IRR0-3)  
2:125  
External Interrupt Vector Register (IVR) 2:123  
External Task Priority Cycle (XTP) 2:130  
External Task Priority Register (XTPR) 2:605  
ExtINT (External Controller Interrupt) 2:96  
extr Instruction 3:54

**F**

F2XM1 Instruction 4:97  
FABS Instruction 4:99  
fabs Instruction 3:55  
FADD Instruction 4:100  
fadd Instruction 3:56  
FADDP Instruction 4:100  
famax Instruction 3:57  
famin Instruction 3:58  
fand Instruction 3:59  
fandcm Instruction 3:60  
Fatal Error 2:350  
Fault Handlers 2:583  
Faults 2:96, 2:537  
FBLD Instruction 4:103  
FBSTP Instruction 4:105  
fc Instruction 3:61  
fchkf Instruction 3:63  
FCHS Instruction 4:108  
fclass Instruction 3:64  
FCLEX Instruction 4:109  
fclrf Instruction 3:66  
FCMOI Instruction 4:115  
FCMOVcc Instruction 4:110  
fcmp Instruction 3:67  
FCOM Instruction 4:112  
FCOMIP Instruction 4:115  
FCOMP Instruction 4:112  
FCOMPP Instruction 4:112  
FCOS Instruction 4:118  
FCR (IA-32 Floating-point Control Register) 1:126  
fcvt Instruction  
  fcvt.fx 3:70  
  fcvt.xf 3:72  
  fcvt.xuf 3:73  
FDECSTP Instruction 4:120  
FDIV Instruction 4:121  
FDIVP Instruction 4:121  
FDIVR Instruction 4:124  
FDIVRP Instruction 4:124  
Fence Semantics 2:508  
fetchadd Instruction 2:508, 3:74  
FFREE Instruction 4:127  
FIADD Instruction 4:100

FICOM Instruction 4:128  
 FICOMP Instruction 4:128  
 FIDIV Instruction 4:121  
 FIDIVR Instruction 4:124  
 FILD Instruction 4:130  
 FIMUL Instruction 4:145  
 FINCSTP Instruction 4:132  
 Firmware 1:7, 2:623  
 Firmware Address Space 2:283  
 Firmware Entrypoint 2:281, 2:350  
 Firmware Interface Table (FIT) 2:287  
 FIST Instruction 4:134  
 FISTP Instruction 4:134  
 FISUB Instruction 4:182, 4:183  
 FISUBR Instruction 4:185  
 FIT (Firmware Interface Table) 2:287  
 FLD Instruction 4:137  
 FLD1 Instruction 4:139  
 FLDCW Instruction 4:141  
 FLDENV Instruction 4:143  
 FLDL2E Instruction 4:139  
 FLDL2T Instruction 4:139  
 FLDLG2 Instruction 4:139  
 FLDLN2 Instruction 4:139  
 FLDPI Instruction 4:139  
 FLDZ Instruction 4:139  
 Floating-point Architecture 1:19, 1:85, 1:205  
 Floating-point Exception Fault 1:102  
 Floating-point Instructions 1:91  
 Floating-point Register (FR) 1:139  
 Floating-point Software Assistance Exception Handler (FPSWA) 2:587  
 Floating-point Status Register (FPSR) 1:31, 1:88  
 flushrs Instruction 3:76  
 fma Instruction 1:210, 3:77  
 fmax Instruction 3:79  
 fmerge Instruction 3:80  
 fmin Instruction 3:82  
 fmix Instruction 3:83  
 fmpy Instruction 3:85  
 fms Instruction 3:86  
 FMUL Instruction 4:145  
 FMULP Instruction 4:145  
 FNCLEX Instruction 4:109  
 fneg Instruction 3:88  
 fnegabs Instruction 3:89  
 FNINIT Instruction 4:133  
 fnma Instruction 3:90  
 fnmpy Instruction 3:92  
 FNOP Instruction 4:148  
 fnorm Instruction 3:93  
 FNSAVE Instruction 4:162  
 FNSTCW Instruction 4:176  
 FNSTENV Instruction 4:178  
 FNSTSW Instruction 4:180  
 for Instruction 3:94  
 fpabs Instruction 3:95  
 fpack Instruction 3:96  
 fpamax Instruction 3:97  
 fpamin Instruction 3:99  
 FPATAN Instruction 4:149  
 fpcmp Instruction 3:101  
 fpcvt Instruction 3:104  
 fpma Instruction 3:107  
 fpmax Instruction 3:109  
 fpmerge Instruction 3:111  
 fpmin Instruction 3:113  
 fpmpy Instruction 3:115  
 fpms Instruction 3:116  
 fpneg Instruction 3:118  
 fpnegabs Instruction 3:119  
 fpnma Instruction 3:120  
 fpnmpy Instruction 3:122  
 fprcpa Instruction 3:123  
 FPREM Instruction 4:151  
 FPREM1 Instruction 4:154  
 fprsqrta Instruction 3:126  
 FPSR (Floating-point Status Register) 1:31, 1:88  
 FPSWA (Floating-point Software Assistance Handler) 2:587  
 FPTAN Instruction 4:157  
 FR (Floating-point Register) 1:139  
 frcpa Instruction 3:128  
 FRNDINT Instruction 4:159  
 frsqrta Instruction 3:131  
 FRSTOR Instruction 4:160  
 FSAVE Instruction 4:162  
 FSACLE Instruction 4:165  
 fselect Instruction 3:134  
 fsetc Instruction 3:135  
 FSIN Instruction 4:167  
 FSINCOS Instruction 4:169  
 FSQRT Instruction 4:171  
 FSR (IA-32 Floating-point Status Register) 1:126  
 FST Instruction 4:173  
 FSTCW Instruction 4:176  
 FSTENV Instruction 4:178  
 FSTP Instruction 4:173  
 FSTSW Instruction 4:180  
 FSUB Instruction 4:182, 4:183  
 fsub Instruction 3:136  
 FSUBP Instruction 4:182, 4:183  
 FSUBR Instruction 4:185  
 FSUBRP Instruction 4:185  
 fswap Instruction 3:137  
 fsxt Instruction 3:139  
 FTST Instruction 4:188  
 FUCOM Instruction 4:190  
 FUCOMI Instruction 4:115  
 FUCOMIP Instruction 4:115  
 FUCOMP Instruction 4:190  
 FUCOMPP Instruction 4:190

FWAIT Instruction 4:386  
 fwb Instruction 3:141  
 FXAM Instruction 4:193  
 FXCH Instruction 4:195  
 fxor Instruction 3:142  
 FXRSTOR Instruction 4:509  
 FXSAVE Instruction 4:512, 4:515  
 FXTRACT Instruction 4:197  
 FYL2X Instruction 4:199  
 FYL2XP1 Instruction 4:201

**G**

General Register (GR) 1:25, 1:139  
 getf Instruction 3:143  
 GR (General Register) 1:139

**H**

hint Instruction 3:145  
 HLT Instruction 4:203

**I**

I/O Architecture 2:615  
 IA-32  
   IA-32 Application Execution 1:109  
   IA-32 Applications 2:239, 2:595  
   IA-32 Architecture 1:7, 1:21  
   IA-32 Current Privilege Level (PSR.cpl) 2:243  
   IA-32 EFLAG Register 1:123, 2:243  
   IA-32 Exception  
     Alignment Check Fault 2:229  
     Code Breakpoint Fault 2:215  
     Data Breakpoint, Single Step, Taken Branch Trap 2:216  
     Device Not Available Fault 2:221  
     Divide Fault 2:214  
     Double Fault 2:222  
     General Protection Fault 2:226  
     INT 3 Trap 2:217  
     Invalid Opcode Fault 2:220  
     Invalid TSS Fault 2:223  
     Machine Check 2:230  
     Overflow Trap 2:218  
     Page Fault 2:227  
     Pending Floating-point Error 2:228  
     Segment Not Present Fault 2:224  
     SSE Numeric Error Fault 2:231  
     Stack Fault 2:225  
   IA-32 Execution Layer 1:109  
   IA-32 Floating-point Control Registers 1:126  
   IA-32 Instruction Reference 4:11  
   IA-32 Instruction Set 2:253  
   IA-32 Intel® MMX™ Technology 1:129  
   IA-32 Intercept  
     Gate Intercept Trap 2:235  
     Instruction Intercept Fault 2:233

Locked Data Reference Fault 2:237  
 System Flag Trap 2:236  
 IA-32 Interrupt  
   Software Trap 2:232  
 IA-32 Interruption 2:111  
 IA-32 Interruption Vector Definitions 2:213  
 IA-32 Interruption Vector Descriptions 2:213  
 IA-32 Memory Ordering 2:265  
 IA-32 Physical Memory References 2:262  
 IA-32 SSE Extensions 1:20, 1:130  
 IA-32 System Registers 2:246  
 IA-32 System Segment Registers 2:241  
 IA-32 Trap Code 2:213  
 IA-32 Virtual Memory References 2:261  
 IBR (Index Breakpoint Register) 2:151, 2:152  
 IDIV Instruction 4:204  
 IFA (interuption Faulting Address) 2:541  
 IFS (Interruption Function State) 2:541  
 IHA (Interruption Hash Address) 2:41, 2:541  
 IIB0 (Interruption Instruction Bundle 0) 2:541  
 IIB1 (Interruption Instruction Bundle 1) 2:541  
 IIM (Interruption Immediate) 2:541  
 IIP (Interruption Instruction Pointer) 2:541  
 IIPA (Interruption Instruction Previous Address)  
   2:541  
 Implicit Prefetch 1:70  
 IMUL Instruction 4:207  
 IN Instruction 4:210  
 INC Instruction 4:212  
 In-flight Resources 2:19  
 INIT (Initialization Event) 2:96, 2:306, 2:635  
 Initialization Event (INIT) 2:96  
 INS Instruction 4:214  
 INSB Instruction 4:214  
 INSD Instruction 4:214  
 Instruction Breakpoint Register (IBR) 2:151,  
   2:152  
 Instruction Debug Faults 2:151  
 Instruction Dependencies 1:148  
 Instruction Encoding 1:38  
 Instruction Formats 3:293  
   SSE 4:483  
 Instruction Group 1:40  
 Instruction Level Parallelism 1:15  
 Instruction Pointer (IP) 1:27, 1:140  
 Instruction Scheduling 1:148, 1:150, 1:164  
 Instruction Serialization 2:18  
 Instruction Set Architecture (ISA) 1:7  
 Instruction Set Modes 1:110  
 Instruction Set Transition 1:14  
 Instruction Set Transitions 2:239, 2:596  
 Instruction Slot Mapping 1:38  
 Instruction Slots 1:38  
 INSW Instruction 4:214  
 INT (External Interrupt) 2:96  
 INT3 Instruction 4:217

INTA (Interrupt Acknowledge) 2:130  
 Inter-processor Interrupt (IPI) 2:127  
 Interrupt Acknowledge Cycle 2:130  
 Interruption Control Registers (CR16-27) 2:36  
 Interruption Handler 2:537  
 Interruption Handling 2:543  
 Interruption Hash Address 2:41  
 Interruption Instruction Bundle Registers (IIB0-1)  
     2:42  
 Interruption Processor Status Register (IPSR) 2:36  
 Interruption Register State 2:540  
 Interruption Registers 2:538  
 Interruption Status Register (ISR) 2:36  
 Interruption Vector 2:165  
     Alternate Data TLB 2:178  
     Alternate Instruction TLB 2:177  
     Break Instruction 2:185  
     Data Access Rights 2:191  
     Data Access-Bit 2:184  
     Data Key Miss 2:181  
     Data Nested TLB 2:179  
     Data TLB 2:176  
     Debug 2:200  
     Dirty-Bit 2:182  
     Disabled FP-Register 2:195  
     External Interrupt 2:186  
     Floating-point Fault 2:203  
     Floating-point Trap 2:204  
     General Exception 2:192  
     IA-32 Exception 2:210  
     IA-32 Intercept 2:211  
     IA-32 Interrupt 2:212  
     Instruction Access Rights 2:190  
     Instruction Access-Bit 2:183  
     Instruction Key Miss 2:180  
     Instruction TLB 2:175  
     Key Permission 2:189  
     Lower-Privilege Transfer Trap 2:205  
     NaT Consumption 2:196  
     Page Not Present 2:188  
     Single Step Trap 2:208  
     Speculation 2:198  
     Taken Branch Trap 2:207  
     Unaligned Reference 2:201  
     Unsupported Data Reference 2:202  
     Virtual External Interrupt 2:187  
     Virtualization 2:209  
 Interruption Vector Address 2:35, 2:538  
 Interruption Vector Table 2:538  
 Interruptions 2:95, 2:537  
 Interrupts 2:96, 2:114  
     External Interrupt Architecture 2:603  
 Interval Time Counter (ITC) 1:31  
 Interval Timer Match Register (ITM) 2:32  
 Interval Timer Offset (ITO) 2:34  
 Interval Timer Vector (ITV) 2:125  
 INTn Instruction 4:217  
 INTO Instruction 4:217  
 invala Instruction 3:146  
 INVD instructions 4:228  
 INVLPG Instruction 4:230  
 IP (Instruction Pointer) 1:27, 1:140  
 IPI (Inter-processor Interrupt) 2:127  
 IPSR (Interruption Processor Status Register)  
     2:36, 2:541  
 IRET Instruction 4:231  
 IRETD Instruction 4:231  
 IRR (External Interrupt Request Registers) 2:125  
 ISR (Interruption Status Register) 2:36, 2:165,  
     2:541  
 Itanium Architecture 1:7  
 Itanium Instruction Set 1:21  
 Itanium System Architecture 1:20  
 Itanium System Environment 1:7, 1:21  
 ITC (Interval Time Counter) 1:31, 2:32  
 itc Instruction 3:147  
 ITIR (Interruption TLB Insertion Register) 2:541  
 ITM (Interval Time Match Register) 2:32  
 ITO (Interval Timer Offset) 2:34  
 itr Instruction 3:149  
 ITV (Interval Timer Vector) 2:125  
 IVA (Interruption Vector Address) 2:35, 2:538  
 IVA-based interruptions 2:95, 2:537  
 IVR (External Interrupt Vector Register) 2:123

**J**

Jcc Instruction 4:239  
 JMP Instruction 4:243  
 JMPE Instruction 1:111, 2:597, 4:249

**K**

Kernel Register (KR) 1:29  
 KR (Kernel Register) 1:29

**L**

LAHF Instruction 4:251  
 Lamport's Algorithm 2:530  
 LAR Instruction 4:252  
 Large Constants 1:53  
 LC (Loop Count Register) 1:33  
 ld Instruction 3:151  
 ldf Instruction 3:157  
 ldip Instruction 3:161  
 LDMXCSR Instruction 4:516  
 LDS Instruction 4:255  
 LEA Instruction 4:258  
 LEAVE Instruction 4:260  
 LES Instruction 4:255  
 lfetech Instruction 3:164  
 LFS Instruction 4:255  
 LGDT Instruction 4:264

LGS Instruction 4:255  
 LIDT Instruction 4:264  
 LLDT Instruction 4:267  
 LMSW Instruction 4:270  
 Load Instructions 1:58  
 loadrs Instruction 3:167  
 Loads from Memory 1:147  
 Local Redirection Registers (LRR0-1) 2:126  
 Locality Hints 1:70  
 LOCK Instruction 4:272  
 LODS Instruction 4:274  
 LODSB Instruction 4:274  
 LODSD Instruction 4:274  
 LODSW Instruction 4:274  
 Logical Instructions 1:51  
 Loop Count Register (LC) 1:33  
 LOOP Instruction 4:276  
 Loop Optimization 1:160, 1:181  
 LOOPcc Instruction 4:276  
 Lower Privilege Transfer Trap 2:151  
 LRR (Local Redirection Registers) 2:126  
 LSL Instruction 4:278  
 LSS Instruction 4:255  
 LTR Instruction 4:282

**M**

Machine Check (MC) 2:95, 2:296, 2:351  
 Machine Check Abort (MCA) 2:632  
 MASKMOVQ Instruction 4:576  
 MAXPS Instruction 4:519  
 MAXSS Instruction 4:521  
 MC (Machine Check) 2:351  
 MCA (Machine Check Abort) 2:95, 2:296, 2:632  
 Memory 1:36  
   Cacheable Page 2:77  
   Memory Access 1:142  
   Memory Access Ordering 1:73  
   Memory Attribute Transition 2:88  
   Memory Attributes 2:75, 2:524  
   Memory Consistency 1:72  
   Memory Fences 2:510  
   Memory Instructions 1:57  
   Memory Management 2:561  
   Memory Ordering 2:507, 2:510  
     IA-32 2:525  
   Memory Reference 1:147  
   Memory Regions 2:561  
   Memory Synchronization 2:526  
 mf Instruction 2:510, 2:526, 3:168  
   mf.a 2:615  
 MINPS Instruction 4:523  
 MINSS Instruction 4:525  
 mix Instruction 3:169  
 MMX technology 1:20  
 MOV Instruction 4:284  
 mov Instruction 3:172

MOVAPS Instruction 4:527  
 MOVD Instruction 4:401  
 MOVHLPS Instruction 4:529  
 MOVHPS Instruction 4:530  
 movl Instruction 3:187  
 MOVLHPS Instruction 4:532  
 MOVLPS Instruction 4:533  
 MOVMSKPS Instruction 4:535  
 MOVNTPS Instruction 4:578  
 MOVNTQ Instruction 4:579  
 MOVQ Instruction 4:403  
 MOVS Instruction 4:292  
 MOVSB Instruction 4:292  
 MOVSD Instruction 4:292  
 MOVSS Instruction 4:536  
 MOVSW Instruction 4:292  
 MOVSX Instruction 4:294  
 MOVUPS Instruction 4:538  
 MOVZX Instruction 4:295  
 MP Coherence 2:507  
 mpy4 Instruction 3:188  
 mpyshl4 Instruction 3:189  
 MUL Instruction 4:297  
 MULPS Instruction 4:540  
 MULSS Instruction 4:541  
 Multimedia Instructions 1:79  
 Multimedia Support 1:20  
 Multi-threading 1:177  
 Multiway Branches 1:173  
 mux Instruction 3:190

**N**

NaT (Not a Thing) 1:155  
 NaTPage (Not a Thing Attribute) 2:86  
 NaTVal (Not a Thing Value) 1:26  
 NEG Instruction 4:299  
 NMI (Non-Maskable Interrupt) 2:96  
 Non-Maskable Interrupt (NMI) 2:96  
 NOP Instruction 4:301  
 nop Instruction 3:193  
 Not A Thing (NaT) 1:155  
 Not a Thing Attribute (NaTPage) 2:86  
 Not a Thing Value (NatVal) 1:26  
 NOT Instruction 4:302

**O**

OLR (On Line Replacement) 2:351  
 Operating Environments 1:14  
 Operating System - See OS (Operating System)  
 OR Instruction 4:304  
 or Instruction 3:194  
 ORPS Instruction 4:542  
 OS (Operating System)  
   Boot Flow Sample Code 2:639  
   Boot Sequence 2:625  
   FPSWA handler 2:587

Illegal Dependency Fault 2:584  
 Long Branch Emulation 2:585  
 Multiple Address Spaces 1:20, 2:562  
 OS\_BOOT Entrypoint 2:283  
 OS\_INIT Entrypoint 2:283  
 OS\_MCA Entrypoint 2:283  
 OS\_RENDEZ Entrypoint 2:283  
 Performance Monitoring Support 2:620  
 Single Address Space 1:20, 2:565  
 Unaligned Reference Handler 2:583  
 Unsupported Data Reference Handler 2:584  
 OUT Instruction 4:306  
 OUTS Instruction 4:308  
 OUTSB Instruction 4:308  
 OUTSD Instruction 4:308  
 OUTSW Instruction 4:308

## P

pack Instruction 3:195  
 PACKSSDW Instruction 4:405  
 PACKSSWB Instruction 4:405  
 PACKUSWB Instruction 4:408  
 padd Instruction 3:197  
 PADDB Instruction 4:410  
 PADDD Instruction 4:410  
 PADDSD Instruction 4:413  
 PADDSW Instruction 4:413  
 PADDUSB Instruction 4:416  
 PADDUSW Instruction 4:416  
 PADDW Instruction 4:410  
 Page Access Rights 2:56  
 Page Sizes 2:57  
 Page Table Address 2:35  
 PAL (Processor Abstraction Layer) 1:7, 1:21,  
     2:279, 2:351  
 PAL Entrypoints 2:282  
 PAL Initialization 2:306  
 PAL Intercepts 2:351  
 PAL Intercepts in Virtual Environment 2:332  
 PAL Procedure Calls 2:628  
 PAL Procedures 2:353  
 PAL Self-test Control Word 2:295  
 PAL Virtualization 2:324  
 PAL Virtualization Optimizations 2:335  
 PAL Virtualization Services 2:486  
 PAL Virtualization Disables 2:346  
 PAL\_A 2:283  
 PAL\_B 2:283  
 PAL\_BRAND\_INFO 2:366  
 PAL\_BUS\_GET\_FEATURES 2:367  
 PAL\_BUS\_SET\_FEATURES 2:369  
 PAL\_CACHE\_FLUSH 2:370  
 PAL\_CACHE\_INFO 2:374  
 PAL\_CACHE\_INIT 2:376  
 PAL\_CACHE\_LINE\_INIT 2:377  
 PAL\_CACHE\_PROT\_INFO 2:378

PAL\_CACHE\_READ 2:380  
 PAL\_CACHE\_SHARED\_INFO 2:382  
 PAL\_CACHE\_SUMMARY 2:384  
 PAL\_CACHE\_WRITE 2:385  
 PAL\_COPY\_INFO 2:388  
 PAL\_COPY\_PAL 2:389  
 PAL\_DEBUG\_INFO 2:390  
 PAL\_FIXED\_ADDR 2:391  
 PAL\_FREQ\_BASE 2:392  
 PAL\_FREQ RATIOS 2:393  
 PAL\_GET\_HW\_POLICY 2:394  
 PAL\_GET\_PSTATE 2:320, 2:396, 2:637  
 PAL\_HALT 2:314  
 PAL\_HALT\_INFO 2:401  
 PAL\_HALT\_LIGHT 2:314, 2:403  
 PAL\_LOGICAL\_TO\_PHYSICAL 2:404  
 PAL\_MC\_CLEAR\_LOG 2:407  
 PAL\_MC\_DRAIN 2:408  
 PAL\_MC\_DYNAMIC\_STATE 2:409  
 PAL\_MC\_ERROR\_INFO 2:410  
 PAL\_MC\_ERROR\_INJECT 2:421  
 PAL\_MC\_EXPECTED 2:434  
 PAL\_MC\_HW\_TRACKING 2:432  
 PAL\_MC\_RESUME 2:436  
 PAL\_MEM\_ATTRIB 2:437  
 PAL\_MEMORY\_BUFFER 2:438  
 PAL\_PERF\_MON\_INFO 2:440  
 PAL\_PLATFORM\_ADDR 2:442  
 PAL\_PMI\_ENTRYPOINT 2:443  
 PAL\_PREFETCH\_VISIBILITY 2:444  
 PAL\_PROC\_GET\_FEATURES 2:446  
 PAL\_PROC\_SET\_FEATURES 2:450  
 PAL\_PSTATE\_INFO 2:319, 2:451  
 PAL\_PTCE\_INFO 2:453  
 PAL\_REGISTER\_INFO 2:454  
 PAL\_RSE\_INFO 2:455  
 PAL\_SET\_HW\_POLICY 2:456  
 PAL\_SET\_PSTATE 2:319, 2:458, 2:637  
 PAL\_SHUTDOWN 2:460  
 PAL\_TEST\_INFO 2:461  
 PAL\_TEST\_PROC 2:462  
 PAL\_VERSION 2:465  
 PAL\_VM\_INFO 2:466  
 PAL\_VM\_PAGE\_SIZE 2:467  
 PAL\_VM\_SUMMARY 2:468  
 PAL\_VM\_TR\_READ 2:470  
 PAL\_VP\_CREATE 2:471  
 PAL\_VP\_ENV\_INFO 2:473  
 PAL\_VP\_EXIT\_ENV 2:475  
 PAL\_VP\_INFO 2:476  
 PAL\_VP\_INIT\_ENV 2:478  
 PAL\_VP\_REGISTER 2:481  
 PAL\_VP\_RESTORE 2:483  
 PAL\_VP\_SAVE 2:484  
 PAL\_VP\_TERMINATE 2:485  
 PAL\_VPS\_RESTORE 2:499

PAL\_VPS\_RESUME\_HANDLER 2:492  
 PAL\_VPS\_RESUME\_NORMAL 2:489  
 PAL\_VPS\_SAVE 2:500  
 PAL\_VPS\_SET\_PENDING\_INTERRUPT 2:495  
 PAL\_VPS\_SYNC\_READ 2:493  
 PAL\_VPS\_SYNC\_WRITE 2:494  
 PAL\_VPS\_THASH 2:497  
 PAL\_VPS\_TTAG 2:498  
 PAL-based Interruptions 2:95, 2:537  
 PALE\_CHECK 2:282, 2:296  
 PALE\_INIT 2:282, 2:306  
 PALE\_PMI 2:282, 2:310  
 PALE\_RESET 2:282, 2:289  
 PAND Instruction 4:419  
 PANDN Instruction 4:421  
 Parallel Arithmetic 1:79  
 Parallel Compares 1:172  
 Parallel Shifts 1:81  
 pavg Instruction 3:201  
 PAVGB Instruction 4:563  
 pavgsub Instruction 3:204  
 PAVGW Instruction 4:563  
 pcmpe Instruction 3:206  
 PCMPEQB Instruction 4:423  
 PCMPEQD Instruction 4:423  
 PCMPEQW Instruction 4:423  
 PCMPGTB Instruction 4:426  
 PCMPGTD Instruction 4:426  
 PCMPGTW Instruction 4:426  
 Performance Monitor Data Register (PMD) 1:33  
 Performance Monitor Events 2:162  
 Performance Monitoring 2:155, 2:619  
 Performance Monitoring Vector 2:126  
 PEEXTRW Instruction 4:565  
 PFS (Previous Function State Register) 1:32  
 Physical Addressing 2:73  
 PIB (Processor Interrupt Block) 2:127  
 PINSRW Instruction 4:566  
 PKR (Protection Key Register) 2:564  
 Platform Management Interrupt (PMI) 2:96,  
     2:310, 2:538, 2:637  
 PMADDWD Instruction 4:429  
 pmax Instruction 3:209  
 PMAXSW Instruction 4:567  
 PMAXUB Instruction 4:568  
 PMC (Performance Monitor Configuration) 2:155  
 PMD (Performance Monitor Data Register) 1:33  
 PMD (Performance Monitor Data) 2:155  
 PMI (Platform Management Interrupt) 2:96,  
     2:310, 2:538, 2:637  
 pmin Instruction 3:211  
 PMINSW Instruction 4:569  
 PMINUB Instruction 4:570  
 PMOVMSKB Instruction 4:571  
 pmpy Instruction 3:213  
 pmpyshr Instruction 3:214  
 PMULHUW Instruction 4:572  
 PMULHW Instruction 4:431  
 PMULLW Instruction 4:433  
 PMV (Performance Monitoring Vector) 2:126  
 POP Instruction 4:311  
 POPA Instruction 4:315  
 POPAD Instruction 4:315  
 popcnt Instruction 3:216  
 POPF Instruction 4:317  
 POPFD Instruction 4:317  
 POR Instruction 4:435  
 Power Management 2:313  
 Power-on Event 2:351  
 PR (Predicate Register) 1:26, 1:140  
 Predicate Register (PR) 1:26, 1:140  
 Predication 1:17, 1:54, 1:143, 1:163, 1:164  
 Prefetch Hints 1:176  
 PREFETCH Instruction 4:580  
 Preserved Values 2:351  
 Previous Function State (PFS) 1:32  
 Privilege Level Transfer 1:84  
 Privilege Levels 2:17  
 probe Instruction 3:217  
 Procedure Calls 2:549  
 Processor Abstraction Layer - See PAL (Processor  
     Abstraction Layer)  
 Processor Abstraction Layer (PAL) 2:279  
 Processor Boot Flow 2:623  
 Processor Identification Registers (CPUID) 1:34  
 Processor Interrupt Block (PIB) 2:127  
 Processor Min-state Save Area 2:302  
 Processor Reset 2:95  
 Processor State Parameter (PSP) 2:299, 2:308  
 Processor Status Register (PSR) 2:23  
 Programmed I/O 2:534  
 Protection Keys 2:59, 2:564  
 psad Instruction 3:220  
 PSADBW Instruction 4:573  
 Pseudo-Code Functions 3:281  
 pshl Instruction 3:222  
 pshladd Instruction 3:223  
 pshr Instruction 3:224  
 pshradd Instruction 3:226  
 PSHUFW Instruction 4:575  
 PSLLD Instruction 4:437  
 PSLLQ Instruction 4:437  
 PSLLW Instruction 4:437  
 PSP (Processor State Parameter) 2:308  
 PSR (Processor Status Register) 2:23  
 PSRAD Instruction 4:440  
 PSRAW Instruction 4:440  
 PSRLD Instruction 4:443  
 PSRLQ Instruction 4:443  
 PSRLW Instruction 4:443  
 psub Instruction 3:227  
 PSUBB Instruction 4:446

PSUBD Instruction 4:446  
 PSUBSB Instruction 4:449  
 PSUBSW Instruction 4:449  
 PSUBUSB Instruction 4:452  
 PSUBUSW Instruction 4:452  
 PSUBW Instruction 4:446  
 PTA (Page Table Address Register) 2:35  
 ptc Instruction  
     ptc.e 2:569, 3:230  
     ptc.g 2:570, 3:231  
     ptc.ga 2:570, 3:231  
     ptc.l 2:568, 3:233  
 ptr Instruction 3:234  
 PUNPCKHBW Instruction 4:455  
 PUNPCKHDQ Instruction 4:455  
 PUNPCKHWD Instruction 4:455  
 PUNPCKLBW Instruction 4:458  
 PUNPCKLDQ Instruction 4:458  
 PUNPCKLWD Instruction 4:458  
 PUSH Instruction 4:320  
 PUSHA Instruction 4:323  
 PUSHAD Instruction 4:323  
 PUSHF Instruction 4:325  
 PUSHFD Instruction 4:325  
 PXOR Instruction 4:461

## R

RAW Dependency 1:149  
 RCL Instruction 4:327  
 RCPPS Instruction 4:543  
 RCPSS Instruction 4:545  
 RCR Instruction 4:327  
 RDMSR Instruction 4:331  
 RDPMC Instruction 4:333  
 RDTSC Instruction 4:335  
 Read-after-write Dependency 1:149  
 Recoverable Error 2:351  
 Recovery Code 1:153, 1:154, 1:156  
 Region Identifier (RID) 2:561  
 Region Register (RR) 2:58, 2:561  
 Register File Transfers 1:82  
 Register Rotation 1:19, 1:185  
 Register Spill and Fill 1:62  
 Register Stack 1:18, 1:47  
 Register Stack Configuration Register (RSC) 1:29  
 Register Stack Engine (RSE) 1:144, 2:133  
 Register State 2:549  
 Release Semantics 2:507  
 Rendezvous 2:301  
 REP Instruction 4:337  
 REPE Instruction 4:337  
 REPNE Instruction 4:337  
 REPNZ Instruction 4:337  
 REPZ Instruction 4:337  
 Reserved Variables 2:351  
 Reset Event 2:95, 2:351

Resource Utilization Counter (RUC) 1:31, 2:33  
 RET Instruction 4:340  
 rfi Instruction 2:543, 3:236  
 RID (Region Identifier) 2:561  
 RNAT(RSE NaT Collection Register) 1:30  
 ROL Instruction 4:327  
 ROR Instruction 4:327  
 Rotating Registers 1:145  
 RR (Region Register) 2:58, 2:561  
 RSC (Register Stack Configuration Register) 1:29  
 RSE (Register Stack Engine) 2:133  
 RSE Backing Store Pointer (BSP) 1:29  
 RSE Backing Store Pointer for Memory Stores  
     (BSPSTORE) 1:30  
 RSE NaT Collection Register (RNAT) 1:30  
 RSM Instruction 4:346  
 rsm Instruction 3:239  
 RSQRTPS Instruction 4:547  
 RSQRTSS Instruction 4:548  
 RUC (Resource Utilization Counter) 1:31, 2:33  
 rum Instruction 3:241

## S

SAHF Instruction 4:347  
 SAL (System Abstraction Layer) 1:7, 1:21, 2:352,  
     2:630  
     SAL\_B 2:283  
     SALE\_ENTRY 2:282, 2:291, 2:305  
     SALE\_PMI 2:282, 2:310  
 SAL Instruction 4:348  
 SAR Instruction 4:348  
 SBB Instruction 4:352  
 SCAS Instruction 4:354  
 SCASB Instruction 4:354  
 SCASD Instruction 4:354  
 SCASW Instruction 4:354  
 Scratch Register 2:352  
 Self Test State Parameter 2:293  
 Self-modifying Code 2:532  
 Semaphore Instructions 1:59  
 Semaphores 2:508  
 Serialization 2:17, 2:537  
 SETcc Instruction 4:356  
 setf Instruction 3:242  
 SFENCE Instruction 4:581  
 SGDT Instruction 4:359  
 SHL Instruction 4:348  
 shl Instruction 3:244  
 shladd Instruction 3:245  
 shladdp4 Instruction 3:246  
 SHLD Instruction 4:362  
 SHR Instruction 4:348  
 shr Instruction 3:247  
 SHRD Instruction 4:364  
 shrp Instruction 3:248  
 SHUFPS Instruction 4:549

## INDEX

SIDT Instruction 4:359  
Single Step Trap 2:151  
SLDT Instruction 4:367  
SMSW Instruction 4:369  
Software Pipelining 1:19, 1:75, 1:145, 1:181  
Speculation 1:16, 1:142, 1:151  
    Control Speculation 1:16  
    Data Speculation 1:17  
    Recovery Code 1:17, 2:580  
    Speculation Check 1:156  
SQRTPS Instruction 4:551  
SQRTSS Instruction 4:552  
srlz Instruction 3:249  
SSE Instructions 4:463  
ssm Instruction 3:250  
st Instruction 3:251  
Stacked Calling Convention 2:352  
Stacked General Registers 2:550  
Stacked Registers 1:144  
Static Calling Convention 2:352  
Static General Registers 2:550  
STC Instruction 4:371  
STD Instruction 4:372  
stf Instruction 3:254  
STI Instruction 4:373  
STMXCSR Instruction 4:553  
Stops 1:38  
Store Instructions 1:59  
Stores to Memory 1:147  
STOS Instruction 4:376  
STOSB Instruction 4:376  
STOSD Instruction 4:376  
STOSW Instruction 4:376  
STR Instruction 4:378  
SUB Instruction 4:379  
sub Instruction 3:256  
SUBPS Instruction 4:554  
SUBSS Instruction 4:555  
sum Instruction 3:257  
sxt Instruction 3:258  
sync Instruction 3:259  
    sync.i 2:526  
System Abstraction Layer - See SAL (System Abstraction Layer)  
System Architecture 1:20  
System Environment 2:13  
System Programmer's Guide 2:501  
System State 2:20

## T

tak Instruction 3:260  
Taken Branch trap 2:151  
Task Priority Register (TPR) 2:123, 2:605  
tbit Instruction 3:261  
TC (Translation Cache) 2:49, 2:567

Template Field Encoding 1:38  
Templates 1:141  
TEST Instruction 4:381  
tf Instruction 3:263  
thash Instruction 3:265  
TLB (Translation Lookaside Buffer) 2:47, 2:565  
tnat Instruction 3:266  
tpa Instruction 3:268  
TPR (Task Priority Register) 2:123, 2:605  
TR (Translation Register) 2:48, 2:566  
Translation Cache (TC) 2:49, 2:567  
    purge 2:568  
Translation Instructions 2:60  
Translation Lookaside Buffer (TLB) 2:47, 2:565  
Translation Register (TR) 2:48, 2:566  
Traps 2:96, 2:537  
ttag Instruction 3:269

## U

UCOMISS Instruction 4:556  
UD2 Instruction 4:383  
UEFI (Unified Extensible Firmware Interface) 2:630  
UM (User Mask Register) 1:33  
UNAT (User NaT Collection Register) 1:31, 1:156  
Uncacheable Page 2:77  
Unchanged Register 2:352  
Unordered Semantics 2:507  
unpack Instruction 3:270  
UNPCKHPS Instruction 4:558  
UNPCKLPS Instruction 4:560  
User Mask (UM) 1:33  
User NaT Collection Register (UNAT) 1:31, 1:156

## V

VERR Instruction 4:384  
VERW Instruction 4:384  
VHPT (Virtual Hash Page Table) 2:61, 2:571  
VHPT Translation Vector 2:173  
Virtual Addressing 2:45  
Virtual Hash Page Table (VHPT) 2:61, 2:571  
Virtual Machine Monitor (VMM) 2:352  
Virtual Processor Descriptor (VPD) 2:325, 2:352  
Virtual Processor State 2:352  
Virtual Processor Status Register (VPSR) 2:327  
Virtual Region Number (VRN) 2:561  
Virtualization 2:44, 2:324  
Virtualization Acceleration Control (vac) 2:329  
Virtualization Disable Control (vdc) 2:329  
VMM (Virtual Machine Monitor) 2:352  
vmsw Instruction 3:273  
VPD (Virtual Processor Descriptor) 2:325, 2:352  
VPSR (Virtual Processor Status Register) 2:327  
VRN (Virtual Region Number) 2:561

**W**

WAIT Instruction 4:386  
WAR Dependency 1:149  
WAW Dependency 1:149  
WBINVD Instruction 4:387  
Write-after-read Dependency 1:149  
Write-after-write Dependency 1:149  
WRMSR Instruction 4:389

**X**

XADD Instruction 4:391  
XCHG Instruction 4:393  
xchg Instruction 2:508, 3:274  
XLAT Instruction 4:395  
XLATB Instruction 4:395  
xma Instruction 3:276  
xmpy Instruction 3:278  
XOR Instruction 4:397  
xor Instruction 3:279  
XORPS Instruction 4:562  
XTP (External Task Priority Cycle) 2:130  
XTPR (External Task Priority Register) 2:605

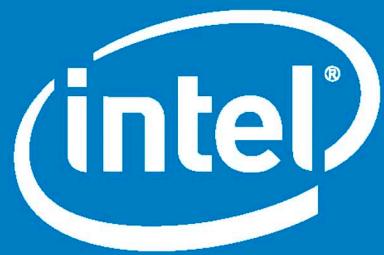
**Z**

zxt Instruction 3:280

## INDEX

*Index:12*

*Index for Volumes 1, 2, 3 and 4*



Copyright ©1999-2010 Intel Corporation. All rights reserved.  
Intel, the Intel logo, Intel Inside, and Itanium are trademarks or  
registered trademarks of Intel Corporation or its subsidiaries  
in the United States and other countries.

Other names and brands may be claimed as the property of others.  
0510/FL/DS/NOD/RRD/2K 323207-001US