Contents

1 About this Manual ................................................................. 3:1
   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

2 Instruction Reference .......................................................... 3:11
   2.1 Instruction Page Conventions ........................................... 3:11
   2.2 Instruction Descriptions ............................................... 3:13

3 Pseudo-Code Functions ....................................................... 3:281

4 Instruction Formats ............................................................ 3:293
   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
5 Resource and Dependency Semantics .................................................. 3:371
  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

Index ........................................................................................................... 3:397

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 | 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 | sz Completers | 3:151 |
| 2-33 | Load Types | 3:151 |
| 2-34 | Load Hints | 3:152 |
| 2-35 | fsz Completers | 3:157 |
| 2-36 | FP Load Types | 3:157 |
| 2-37 | iftype Mnemonic Values | 3:164 |
| 2-38 | lhint 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 | xsz 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 ntats | 3:266 |
| 2-59 | Test NaT Relations for Parallel ntats | 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 |
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 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 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, “Interruption 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 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 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.


**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:


- **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, 243191and 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

<table>
<thead>
<tr>
<th>Date of Revision</th>
<th>Revision Number</th>
<th>Description</th>
</tr>
</thead>
</table>
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.

Relaxed reserved and ignored fields checkings in IA-32 application registers in Vol 1 Ch 6 and Vol 2, Part I, Ch 10.

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 ptc.i instruction in Vol 3 for details.

Architecture extensions for processor Power/Performance states (P-states).

See Vol 2 PAL Chapter for details.

Introduced Unimplemented Instruction Address fault.

Relaxed ordering constraints for VHPT walks. See Vol 2, Part I, Ch 4 and 5 for details.

Architecture extensions for processor virtualization.

All instructions which must be last in an instruction group results in undefined behavior when this rule is violated.

Added architectural sequence that guarantees increasing ITC and PMD values on successive reads.

Addition of PAL_BRAND_INFO, PAL_GET_HW_POLICY, PAL_MC_ERROR_INJECT, PAL_MEMORY_BUFFER, PAL_SET_HW_POLICY and PAL_SHUTDOWN procedures.

Allows IPI-redirection feature to be optional.

Undefined behavior for 1-byte accesses to the non-architected regions in the IPI block.

Modified insertion behavior for TR overlaps. See Vol 2, Part I, Ch 4 for details.

"Bus parking" feature is now optional for PAL_BUS_GET_FEATURES.

Introduced low-power synchronization primitive using hint instruction.

FR32-127 is now preserved in PAL calling convention.

New return value from PAL_VM_SUMMARY procedure to indicate the number of multiple concurrent outstanding TLB purges.

Performance Monitor Data (PMD) registers are no longer sign-extended.

New memory attribute transition sequence for memory on-line delete. See Vol 2, Part I, Ch 4 for details.

Added 'shared error' (se) bit to the Processor State Parameter (PSP) in PAL_MC_ERROR_INFO procedure.

Clarified PMU interrupts as edge-triggered.

Modified 'proc_number' parameter in PAL_LOGICAL_TO_PHYSICAL procedure.

Modified pal_copy_info alignment requirements.

New bit in PAL_PROC_GET_FEATURES for variable P-state performance.

Clarified descriptions for check_target_register and check_target_register_sof.

Various fixes in dependency tables in Vol 3 Ch 5.

Clarified effect of sending IPIs to non-existent processor in Vol 2, Part I, Ch 5.

Clarified instruction serialization requirements for interruptions in Vol 2, Part II, Ch 3.

Updated performance monitor context switch routine in Vol 2, Part I, Ch 7.
### Date of Revision | Revision Number | Description
--- | --- | ---
August 2002 | 2.1 | Added Predicate Behavior of `alloc` Instruction Clarification (Section 4.1.2, Part I, Volume 1; Section 2.2, Part I, Volume 3).
Added New `fc.i` Instruction (Section 4.4.6.1, and 4.4.6.2, Part I, Volume 1; Section 4.4.7, 4.4.8, 4.4.9, 4.4.10, and 4.4.11, Part I, Volume 2; Section 2.5, 2.5.1, and 2.5.2, Part II, Volume 2; Section 2.2, 2.2.1, 2.3, 2.3.1, and 2.3.2, Part I, Volume 3).
Added Interval Time Counter (ITC) Fault Clarification (Section 3.3.2, Part I, Volume 2).
Added Interruption Control Registers Clarification (Section 3.3.5, Part I, Volume 2).
Added Spontaneous NaT Generation on Speculative Load (ld.s) (Section 5.5.5 and 11.9, Part I, Volume 2; Section 2.2 and 3, Part I, Volume 3).
Added Performance Counter Standardization (Sections 7.2.3 and 11.6, Part I, Volume 2).
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).
Added IA_32_Exception (Debug) IIPA Description Change (Section 9.2, Part I, Volume 2).
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).
Added BR1 to Min-state Save Area (Sections 11.3.2.3 and 11.3.3, Part I, Volume 2).
Added Fault Handling Semantics for `lfetch.fault` Instruction (Section 2.2, Part I, Volume 3).

December 2001 | 2.0 | Volume 1:
Faults in ld.c that hits ALAT clarification (Section 4.4.5.3.1).
IA-32 related changes (Section 6.2.5.4, Section 6.2.3, Section 6.2.4, Section 6.2.5.3).
Load instructions change (Section 4.4.1).
<table>
<thead>
<tr>
<th>Date of Revision</th>
<th>Revision Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume 2:</td>
<td></td>
<td>Class pr-writers-int clarification (Table A-5).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAL_MC_DRAIN clarification (Section 4.4.6.1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VHPT walk and forward progress change (Section 4.1.1.2).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA-32 IBR/DBR match clarification (Section 7.1.1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ISR figure changes (pp. 8-5, 8-26, 8-33 and 8-36).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAL_CACHE_FLUSH return argument change – added new status return argument (Section 11.8.3).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAL self-test Control and PAL_A procedure requirement change – added new arguments, figures, requirements (Section 11.2).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAL_CACHE_FLUSH clarifications (Chapter 11).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-speculative reference clarification (Section 4.4.6).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RID and Preferred Page Size usage clarification (Section 4.1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VHPT read atomicity clarification (Section 4.1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IIP and WC flush clarification (Section 4.4.5).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Revised RSE and PMC typographical errors (Section 6.4).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Revised DV table (Section A.4).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory attribute transitions – added new requirements (Section 4.4).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MCA for WC/UC aliasing change (Section 4.4.1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bus lock deprecation – changed behavior of DCR ‘ic’ bit (Section 3.3.4.1, Section 10.6.8, Section 11.8.3).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAL_PROC_GET/SET_FEATURES changes – extend calls to allow implementation-specific feature control (Section 11.8.3).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Split PAL_A architecture changes (Section 11.1.6).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Simple barrier synchronization clarification (Section 13.4.2).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Limited speculation clarification – added hardware-generated speculative references (Section 4.4.6).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAL memory accesses and restrictions clarification (Section 11.9).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSP validity on INITs from PAL_MC_ERROR_INFO clarification (Section 11.8.3).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speculation attributes clarification (Section 4.4.6).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAL_A FIT entry, PAL_VM_TR_READ, PSP, PAL_VERSION clarifications (Sections 11.8.3 and 11.3.2.1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TLB searching clarifications (Section 4.1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA-32 related changes (Section 10.3, Section 10.3.2, Section 10.3.2, Section 10.3.3.1, Section 10.10.1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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).</td>
</tr>
<tr>
<td>Volume 3:</td>
<td></td>
<td>IA-32 CPUID clarification (p. 5-71).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Revised figures for extract, deposit, and alloc instructions (Section 2.2).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RCPPS, RCPSS, RSQRTPS, and RSQRTSS clarification (Section 7.12).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA-32 related changes (Section 5.3).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tak, tpa change (Section 2.2).</td>
</tr>
<tr>
<td>July 2000</td>
<td>1.1</td>
<td>Volume 1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Processor Serial Number feature removed (Chapter 3).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarification on exceptions to instruction dependency (Section 3.4.3).</td>
</tr>
</tbody>
</table>
Volume 2:
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).
FCR/FDR mappings, and clarification to the value of PSR.ri after an RFI (Chapters 3 and 4).
Clarification regarding ordering data dependency.
Out-of-order IPI delivery is now allowed (Chapters 4 and 5).
Content of EFLAG field changed in IIM (p. 9-24).
PAL_CHECK and PAL_INIT calls – exit state changes (Chapter 11).
PAL_CHECK processor state parameter changes (Chapter 11).
PAL_BUS_GET/SET_FEATURES calls – added two new bits (Chapter 11).
PAL_MC_ERROR_INFO call – Changes made to enhance and simplify the call to provide more information regarding machine check (Chapter 11).
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).
PAL_CACHE_FLUSH – added a new cache_type argument (Chapter 11).
PAL_SHUTDOWN – removed from list of PAL calls (Chapter 11).
Clarified memory ordering changes (Chapter 13).
Clarification in dependence violation table (Appendix A).

Volume 3:
fmix instruction page figures corrected (Chapter 2).
Clarification of “reserved” fields in ITIR (Chapters 2 and 3).
Modified conditions for alloc/loadrs/flushrs instruction placement in bundle/instruction group (Chapters 2 and 4).
IA-32 JMPE instruction page typo fix (p. 5-238).
Processor Serial Number feature removed (Chapter 5).

<table>
<thead>
<tr>
<th>Date of Revision</th>
<th>Revision Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 2000</td>
<td>1.0</td>
<td>Initial release of document.</td>
</tr>
</tbody>
</table>

Volume 3: About this Manual

§
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

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

Table 2-2. Instruction Page Font Conventions

<table>
<thead>
<tr>
<th>Font</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular</td>
<td>(Format section) Required characters in an assembly language mnemonic</td>
</tr>
<tr>
<td>italic</td>
<td>(Format section) Assembly language field name that must be filled with one of a range of legal values listed in the Description section</td>
</tr>
<tr>
<td>code</td>
<td>(Operation section) C code specifying instruction behavior</td>
</tr>
<tr>
<td>code_italic</td>
<td>(Operation section) Assembly language field name corresponding to a italic field listed in the Format section</td>
</tr>
</tbody>
</table>

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

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

### Table 2-4. C Syntax Differences

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>{msb:lsb}, {bit}</td>
<td>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.</td>
</tr>
<tr>
<td>u&gt;, u&gt;=, u&lt;, u&lt;=</td>
<td>Unsigned inequality relations. Variables on either side of the operator are treated as unsigned.</td>
</tr>
<tr>
<td>u&gt;&gt;, u&gt;&gt;=</td>
<td>Unsigned right shift. Zeroes are shifted into the most significant bit position.</td>
</tr>
<tr>
<td>u+</td>
<td>Unsigned addition. Operands are treated as unsigned, and zero-extended.</td>
</tr>
<tr>
<td>u*</td>
<td>Unsigned multiplication. Operands are treated as unsigned.</td>
</tr>
</tbody>
</table>

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

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

2.2 Instruction Descriptions

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

Format:

- \((qp) add \ r_1 = r_2, r_3\)  \(\text{register\_form} \ \ A1\)
- \((qp) add \ r_1 = r_2, r_3, 1\)  \(\text{plus1\_form, register\_form} \ \ A1\)
- \((qp) add \ r_1 = imm, r_3\)  \(\text{pseudo-op} \ \ A1\)
- \((qp) adds \ r_1 = \text{imm14}, r_3\)  \(\text{imm14\_form} \ \ A4\)
- \((qp) addl \ r_1 = \text{imm22}, r_3\)  \(\text{imm22\_form} \ \ A5\)

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 imm14 form the first operand is taken from the sign-extended \(\text{imm14}\) encoding field; in the imm22 form the first operand is taken from the sign-extended \(\text{imm22}\) 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:

```c
if (FR[qp]) {
    check_target_register(r1);

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

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

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

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

Interruptions: Illegal Operation fault
addp4 — Add Pointer

Format:

(qp) addp4 r4 = r2, r3

(qp) addp4 r4 = imm14, r3

register_form A1

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 r3 are copied to bits {62:61} of the result. This result is placed in GR r4. In the register_form the first operand is GR r2; in the imm14_form the first operand is taken from the sign-extended imm14 encoding field.

Figure 2-1. Add Pointer

Operation:

if (PR[qp]) {
    check_target_register(r4);

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

    tmp_res = tmp_src + GR[r3];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[r3]{31:30};
    GR[r4] = tmp_res;
    GR[r4].nat = tmp_nat || GR[r3].nat;
}

Interruptions: Illegal Operation fault
alloc — Allocate Stack Frame

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

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

```c
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);
if (tmp_sor != CFM.sor &&
    (CFM.rrb.gr != 0 || CFM.rrb.fr != 0 || CFM.rrb.pr != 0))
    reserved_register_field_fault();
```

```c
CFM.sof = tmp_sof;
CFM.sol = tmp_sol;
CFM.sor = tmp_sor;
```

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

Interruptions:

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

**Format:**

- \((qp)\) and \(r_1 = r_2, r_3\) \hspace{1cm} \text{register_form} \hspace{1cm} A1
- \((qp)\) and \(r_1 = \text{imm}_8, r_3\) \hspace{1cm} \text{imm8_form} \hspace{1cm} 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 \(\text{imm}_8\) encoding field.

**Operation:**

```c
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:  

\[(qp) \text{ andcm } r_1 = r_2, r_3 \quad \text{register_form} \quad A1\]
\[(qp) \text{ andcm } r_1 = \text{imm8}, r_3 \quad \text{imm8_form} \quad A3\]

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 \textit{imm8} encoding field.

Operation:  

\[
\text{if (PR[qp])} \{
\quad \text{check_target_register}(r_1);
\quad \text{tmp_src} = (\text{register_form} ? \text{GR}[r_2] : \text{sign_ext(imm8, 8)});
\quad \text{tmp_nat} = (\text{register_form} ? \text{GR}[r_2].nat : 0);
\quad \text{GR}[r_1] = \text{tmp_src} \& \neg \text{GR}[r_3];
\quad \text{GR}[r_1].nat = \text{tmp_nat} || \text{GR}[r_3].nat;
\}
\]

Interruptions:  Illegal Operation fault
**br — Branch**

**Format:**
- 

```
(qp) br.btype.bwh.ph.dh target25  
ip_relative_form  B1
```
- 

```
(qp) br.btype.bwh.ph.dh b1 = target25  
call_form, ip_relative_form  B3
```
- 

```
br.btype.bwh.ph.dh target25  
counted_form, ip_relative_form  B2
```
- 

```
(qp) br.btype.bwh.ph.dh b2  
pseudo-op  B4
```
- 

```
(qp) br.btype.bwh.ph.dh b1 = b2  
call_form, indirect_form  B5
```
- 

```
br.ph.dh b2  
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 `target25` operand, in assembly, specifies a label to branch to. This is encoded in the branch instruction as a signed immediate displacement (`imm21`) between the target bundle and the bundle containing this instruction (`imm21 = target25 - IP >> 4`). For indirect branches, the target address is taken from BR `b2`.

**Table 2-6. Branch Types**

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

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

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

- **cond**: If the qualifying predicate is 1, the branch is taken. Otherwise it is not taken.
- **call**: If the qualifying predicate is 1, the branch is taken and several other actions occur:
  - The current values of the Current Frame Marker (CFM), the EC application register and the current privilege level are saved in the Previous Function State application register.
  - The caller’s stack frame is effectively saved and the callee is provided with a frame containing only the caller’s output region.
  - The rotation rename base registers in the CFM are reset to 0.
  - A return link value is placed in BR $b_1$.
- **return**: If the qualifying predicate is 1, the branch is taken and the following occurs:
  - CFM, EC, and the current privilege level are restored from PFS. (The privilege level is restored only if this does not increase privilege.)
  - The caller’s stack frame is restored.
  - If the return lowers the privilege, and PSR.lp is 1, then a Lower-Privilege Transfer trap is taken.
- **ia**: The branch is taken unconditionally, if it is not intercepted by the OS. The effect of the branch is to invoke the IA-32 instruction set (by setting PSR.is to 1) and begin processing IA-32 instructions at the virtual linear target address contained in BR $b_2 \{31:0\}$. If the qualifying predicate is not PR 0, an Illegal Operation fault is raised. If instruction set transitions are disabled (PSR.di is 1), then a Disabled Instruction Set Transition fault is raised.

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

Software must ensure the code segment descriptor (CSD) and selector (CS) are loaded before issuing the branch. If the target EIP value exceeds the code segment limit or has a code segment privilege violation, an IA_32_Exception(GPFault) is raised on the target IA-32 instruction. For entry into 16-bit IA-32 code, if BR $b_2$ is not within 64K-bytes of CSD.base a GPFault is raised on the target instruction. EFLAG.rf is unmodified until the successful completion of the first IA-32 instruction. PSR.da, PSR.id, PSR.ia, PSR.dd, and PSR.ed are cleared to zero after br.ia completes execution and before the first IA-32 instruction begins execution. EFLAG.rf is not cleared until the target IA-32 instruction successfully completes.

Software must 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.
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.
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[gp];
        break;

    case 'call':  // call saves a return link
        tmp_taken = PR[gp];
        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
        break;
}

Table 2-7.  Branch Whether Hint

<table>
<thead>
<tr>
<th>bwh Completer</th>
<th>Branch Whether Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>spnt</td>
<td>Static Not-Taken</td>
</tr>
<tr>
<td>sptk</td>
<td>Static Taken</td>
</tr>
<tr>
<td>dpnt</td>
<td>Dynamic Not-Taken</td>
</tr>
<tr>
<td>dpstk</td>
<td>Dynamic Taken</td>
</tr>
</tbody>
</table>

Table 2-8.  Sequential Prefetch Hint

<table>
<thead>
<tr>
<th>ph Completer</th>
<th>Sequential Prefetch Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>few or none</td>
<td>Few lines</td>
</tr>
<tr>
<td>many</td>
<td>Many lines</td>
</tr>
</tbody>
</table>

Table 2-9.  Branch Cache Deallocation Hint

<table>
<thead>
<tr>
<th>dh Completer</th>
<th>Branch Cache Deallocation Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Don't deallocate</td>
</tr>
<tr>
<td>clr</td>
<td>Deallocate branch information</td>
</tr>
</tbody>
</table>
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;
}
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]--;
b-zero;

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

case 'wtop':
  case 'wexit': // SW pipelined while loop
    if (slot != 2)
      illegal_operation_fault();
    if (btype == 'wtop') tmp_taken = (PR[gp] || (AR[EC] u> 1));
    if (btype == 'wexit') tmp_taken = !PR[gp] || (AR[EC] u> 1));
    if (PR[gp]) {
      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.th)
  taken_branch_trap();
}

**Interruptions:**
- Illegal Operation fault
- Disabled Instruction Set Transition fault
- Unimplemented Instruction Address trap

Additional Faults on IA-32 target instructions:
- IA_32_Exception(GPFault)
- Disabled FP Reg Fault if PSR.dfh is 1
break — Break

Format:

- \((qp)\) break \(imm_{21}\)
- \((qp)\) break.i \(imm_{21}\)
- \((qp)\) break.b \(imm_{21}\)
- \((qp)\) break.m \(imm_{21}\)
- \((qp)\) break.f \(imm_{21}\)
- \((qp)\) break.x \(imm_{62}\)

pseudo-op
i_unit_form I19
b_unit_form B9
m_unit_form M37
f_unit_form F15
x_unit_form X1

Description:

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

For the b_unit_form, \(imm_{21}\) is ignored and the value zero is placed in the Interruption Immediate control register (IIM).

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

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

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

Operation:

\[
\text{if (PR[qp])} \{ \\
\quad \text{if (b_unit_form)} \\
\quad \quad \text{immediate} = 0; \\
\quad \text{else if (x_unit_form)} \\
\quad \quad \text{immediate} = \text{zero_ext}(imm_{62}, \ 21); \\
\quad \text{else } // i_unit_form || m_unit_form || f_unit_form \\
\quad \quad \text{immediate} = \text{zero_ext}(imm_{21}, \ 21); \\
\quad \text{break_instruction_fault(immediate);} \\
\}
\]

Interruptions: Break Instruction fault
**brl — Branch Long**

**Format:**

(qp) \( \text{brl.btype.bwh.ph.dh target}_{64} \)  
(qp) \( \text{brl.btype.bwh.ph.dh b}_1 = \text{target}_{64} \)  
\( \text{brl.ph.dh target}_{64} \)

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

**Table 2-10. Long Branch Types**

<table>
<thead>
<tr>
<th>btype</th>
<th>Function</th>
<th>Branch Condition</th>
<th>Target Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond or none</td>
<td>Conditional branch</td>
<td>Qualifying predicate</td>
<td>IP-relative</td>
</tr>
<tr>
<td>call</td>
<td>Conditional procedure call</td>
<td>Qualifying predicate</td>
<td>IP-relative</td>
</tr>
</tbody>
</table>

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

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

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

Read after Write (RAW) and Write after Read (WAR) dependency requirements for long branch instructions are slightly different than for other instructions but are the same as for branch instructions. See page 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[gp];
        break;
    case 'call': // call saves a return link
        tmp_taken = PR[gp];
        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();
}
```

**Interruptions:**
- Illegal Operation fault
- Taken Branch trap
- Unimplemented Instruction Address trap
**brp — Branch Predict**

**Format:**
- \texttt{brp.ipwh.ih target_{25}, tag_{13}} \hspace{1cm} \text{ip\_relative\_form B6}
- \texttt{brp.indwh.ih b_2, tag_{13}} \hspace{1cm} \text{indirect\_form B7}
- \texttt{brp.ret.indwh.ih b_2, tag_{13}} \hspace{1cm} \text{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 \texttt{nop} instruction except for its performance effects.

The \textit{tag_{13}} operand, in assembly, specifies the address of the branch instruction to which this prediction information applies. This is encoded in the branch predict instruction as a signed immediate displacement (\textit{timm}_9) between the bundle containing the presaged branch and the bundle containing this instruction (\textit{timm}_9 = tag_{13} - IP >> 4).

The \textit{target}_{25} operand, in assembly, specifies the label that the presaged branch will have as its target. This is encoded in the branch predict instruction exactly as in branch instructions, with a signed immediate displacement (\textit{imm}_{21}) between the target bundle and the bundle containing this instruction (\textit{imm}_{21} = target_{25} - 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 \textit{b}_2.

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

Other hints can be given about the presaged branch. Values for various hint completers are shown in the following tables. For more details, refer to Section 4.5.2, “Branch Prediction Hints” on page 1:78.

The \textit{ipwh} and \textit{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**

<table>
<thead>
<tr>
<th>\textit{ipwh} Completer</th>
<th>IP-relative Branch Predict Whether Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>sptk</td>
<td>Presaged branch should be predicted Static Taken</td>
</tr>
<tr>
<td>loop</td>
<td>Presaged branch will be \texttt{br.cloop}, \texttt{br.ctop}, or \texttt{br.wtop}</td>
</tr>
<tr>
<td>exit</td>
<td>Presaged branch will be \texttt{br.cexit} or \texttt{br.wexit}</td>
</tr>
<tr>
<td>dptk</td>
<td>Presaged branch should be predicted Dynamically</td>
</tr>
</tbody>
</table>

**Table 2-12. Indirect Branch Predict Whether Hint**

<table>
<thead>
<tr>
<th>\textit{indwh} Completer</th>
<th>Indirect Branch Predict Whether Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>sptk</td>
<td>Presaged branch should be predicted Static Taken</td>
</tr>
<tr>
<td>dptk</td>
<td>Presaged branch should be predicted Dynamically</td>
</tr>
</tbody>
</table>

The \textit{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**

<table>
<thead>
<tr>
<th>\textit{ih} Completer</th>
<th>Branch Predict Importance Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Less important</td>
</tr>
<tr>
<td>imp</td>
<td>More important</td>
</tr>
</tbody>
</table>
Operation: 
\[
\text{tmp\_tag} = \text{IP} + \text{sign\_ext}((\text{imm}_9 \ll 4), 13);
\]
if (ip\_relative\_form) {
\[
\text{tmp\_target} = \text{IP} + \text{sign\_ext}((\text{imm}_{21} \ll 4), 25);
\]
\[
\text{tmp\_wh} = \text{ipwh};
\]
} else { // indirect\_form
\[
\text{tmp\_target} = BR[b_2];
\]
\[
\text{tmp\_wh} = \text{indwh};
\]
}
br\text{anch\_predict}(\text{tmp\_wh, } \text{i}h, \text{return\_form, tmp\_target, tmp\_tag});

Interruptions: None
bsw — Bank Switch

**Format:**

```
bsw.0  zero_form  B8
bsw.1  one_form  B8
```

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

A **bsw** instruction must be the last instruction in an instruction group; otherwise, 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:**

```cpp
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;
```

**Interruptions:**

Privileged Operation fault  Virtualization fault

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

Format:
- \((qp)\) \(\text{chk.s} \ r_2, \text{target}_{25}\)  
  \text{pseudo-op} \  
- \((qp)\) \(\text{chk.s.i} \ r_2, \text{target}_{25}\)  
  \text{control_form, i_unit_form, gr_form} \  \text{I20} \  
- \((qp)\) \(\text{chk.s.m} \ r_2, \text{target}_{25}\)  
  \text{control_form, m_unit_form, gr_form} \  \text{M20} \  
- \((qp)\) \(\text{chk.s} \ f_2, \text{target}_{25}\)  
  \text{control_form, fr_form} \  \text{M21} \  
- \((qp)\) \(\text{chk.a.clr} \ r_1, \text{target}_{25}\)  
  \text{data_form, gr_form} \  \text{M22} \  
- \((qp)\) \(\text{chk.a.clr} \ f_1, \text{target}_{25}\)  
  \text{data_form, fr_form} \  \text{M23} \  

Description: The result of a control- or data-speculative calculation is checked for success or failure. If the check fails, a branch to \(\text{target}_{25}\) 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 \(\text{chk.a}\) with general register specifier \(r_0\) or floating-point register specifiers \(f_0\) or \(f_1\) always fails.

The \(\text{target}_{25}\) 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 \((imm_{21} = \text{target}_{25} - \text{IP} >> 4)\).

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

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

For the data_form, if an ALAT entry matches, the matching ALAT entry can be optionally invalidated, based on the value of the aclr completer (See Table 2-14).

### Table 2-14. ALAT Clear Completer

<table>
<thead>
<tr>
<th>aclr Completer</th>
<th>Effect on ALAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>clr</td>
<td>Invalidate matching ALAT entry</td>
</tr>
<tr>
<td>nc</td>
<td>Don’t invalidate</td>
</tr>
</tbody>
</table>

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

```c
if (PR[gp]) {
    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 Unimplemented Instruction Address trap Speculative Operation fault Taken Branch trap
clrrrb — Clear RRB

Format:

- clrrrb
- clrrrb.pr

all_form  B8
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:

```c
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:

\[
\text{if (PR[qp])}
\]

\[
\text{if (!instruction_implemented(CLZ))}
\]

\[
\text{illegal_operation_fault();}
\]

\[
\text{check_target_register(r_1);}\]

\[
tmp\_val = 0;
\]

\[
do {
\text{if (GR[r_3]\{63 - tmp\_val\} != 0) break;}
} while (tmp\_val++ < 63);
\]

\[
\text{GR[r_1] = tmp\_val;}
\]

\[
\text{GR[r_1].nat = GR[r_3].nat;}
\]

Interruptions: Illegal Operation fault
cmp — Compare

Format:

- \((qp)\) cmp.crel.ctype \(p_1, p_2 = r_2, r_3\) register_form A6
- \((qp)\) cmp.crel.ctype \(p_1, p_2 = \text{imm}_8\) imm8_form A8
- \((qp)\) cmp.crel.ctype \(p_1, p_2 = r_0, r_3\) parallel_inequality_form A7
- \((qp)\) cmp.crel.ctype \(p_1, p_2 = r_3, r_0\) pseudo-op

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

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

The unc type is special in that it first initializes both predicate targets to 0, independent of the qualifying predicate. It then operates the same as the normal type. The behavior of the compare types is described in Table 2-15. A blank entry indicates the predicate target is left unchanged.

Table 2-15. Comparison Types

<table>
<thead>
<tr>
<th>ctype</th>
<th>Pseudo-op of</th>
<th>(PR[qp]=0)</th>
<th>(PR[qp]=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(PR[p_1])</td>
<td>(PR[p_2])</td>
</tr>
<tr>
<td>none</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>unc</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>or</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>and</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>or, and</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cm</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>and, cm</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

Of the ten relations, not all are directly implemented in hardware. Some are actually pseudo-ops. For these, the assembler simply switches the source operand specifiers and/or switches the predicate target specifiers and uses an implemented relation. For some of the pseudo-op compares in the imm8_form, the assembler subtracts 1 from the immediate value, making the allowed immediate range slightly different. Of the six parallel compare types, three of the types are actually pseudo-ops. The assembler
simply uses the negative relation with an implemented type. The implemented relations and how the pseudo-ops map onto them are shown in Table 2-16 (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

<table>
<thead>
<tr>
<th>crel</th>
<th>Compare Relation (a rel b)</th>
<th>Register Form is a pseudo-op of</th>
<th>Immediate Form is a pseudo-op of</th>
<th>Immediate Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>a == b</td>
<td>eq</td>
<td>p₁ ↔ p₂</td>
<td>-128 .. 127</td>
</tr>
<tr>
<td>ne</td>
<td>a != b</td>
<td>eq</td>
<td>p₁ ↔ p₂</td>
<td>-128 .. 127</td>
</tr>
<tr>
<td>lt</td>
<td>a &lt; b</td>
<td>signed</td>
<td>lt a ↔ b p₁ ↔ p₂</td>
<td>-128 .. 127</td>
</tr>
<tr>
<td>le</td>
<td>a &lt;= b</td>
<td>lt</td>
<td>a-1</td>
<td>-127 .. 128</td>
</tr>
<tr>
<td>gt</td>
<td>a &gt; b</td>
<td>lt</td>
<td>a-1</td>
<td>-127 .. 128</td>
</tr>
<tr>
<td>ge</td>
<td>a &gt;= b</td>
<td>lt</td>
<td>p₁ ↔ p₂</td>
<td>-128 .. 127</td>
</tr>
<tr>
<td>ltu</td>
<td>a &lt; b</td>
<td>unsiged</td>
<td>ltu a ↔ b p₁ ↔ p₂</td>
<td>0 .. 127, 2&lt;sup&gt;64&lt;/sup&gt;-128 .. 2&lt;sup&gt;64&lt;/sup&gt;-1</td>
</tr>
<tr>
<td>leu</td>
<td>a &lt;= b</td>
<td>ltu</td>
<td>a-1</td>
<td>1 .. 128, 2&lt;sup&gt;64&lt;/sup&gt;-127 .. 2&lt;sup&gt;64&lt;/sup&gt;</td>
</tr>
<tr>
<td>gtu</td>
<td>a &gt; b</td>
<td>ltu</td>
<td>a-1</td>
<td>1 .. 128, 2&lt;sup&gt;64&lt;/sup&gt;-127 .. 2&lt;sup&gt;64&lt;/sup&gt;</td>
</tr>
<tr>
<td>geu</td>
<td>a &gt;= b</td>
<td>ltu</td>
<td>p₁ ↔ p₂</td>
<td>0 .. 127, 2&lt;sup&gt;64&lt;/sup&gt;-128 .. 2&lt;sup&gt;64&lt;/sup&gt;-1</td>
</tr>
</tbody>
</table>

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. Unsinged 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 r₂) 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

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

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

**Format:**

- $(qp)$ cmp4.crel ctype $p_1, p_2 = r_2, r_3$
- $(qp)$ cmp4.crel ctype $p_1, p_2 = \text{imm}_8, r_3$
- $(qp)$ cmp4.crel ctype $p_1, p_2 = r_0, r_3$
- $(qp)$ cmp4.crel ctype $p_1, p_2 = r_3, r_0$

**Description:** The least significant 32 bits from each of two source operands are compared for one of ten relations specified by $crel$. This produces a boolean result which is 1 if the comparison condition is true, and 0 otherwise. This result is written to the two predicate register destinations, $p_1$ and $p_2$. The way the result is written to the destinations is determined by the compare type specified by $ctype$. See the Compare instruction and Table 2-15 on page 3:39.

In the register_form the first operand is GR $r_2$; in the imm8_form the first operand is taken from the sign-extended $imm_8$ encoding field; and in the parallel_inequality_form the first operand must be GR 0. The parallel_inequality_form is only used when the compare type is one of the parallel types, and the relation is an inequality ($>$, $>=$, $<$, $<=$). See the Compare instruction and Table 2-17 on page 3:40.

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

Of the ten relations, not all are directly implemented in hardware. Some are actually pseudo-ops. See the Compare instruction and Table 2-16 and Table 2-17 on page 3:40. The range for immediates is given below.

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

<table>
<thead>
<tr>
<th>$crel$</th>
<th>Compare Relation ($a \ rel b$)</th>
<th>Immediate Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>$a == b$</td>
<td>-128 .. 127</td>
</tr>
<tr>
<td>ne</td>
<td>$a != b$</td>
<td>-128 .. 127</td>
</tr>
<tr>
<td>lt</td>
<td>$a &lt; b$</td>
<td>signed</td>
</tr>
<tr>
<td>le</td>
<td>$a &lt;= b$</td>
<td>-128 .. 127</td>
</tr>
<tr>
<td>ge</td>
<td>$a &gt;= b$</td>
<td>-127 .. 128</td>
</tr>
<tr>
<td>gt</td>
<td>$a &gt; b$</td>
<td>-127 .. 128</td>
</tr>
<tr>
<td>ltu</td>
<td>$a &lt; b$</td>
<td>unsigned</td>
</tr>
<tr>
<td>leu</td>
<td>$a &lt;= b$</td>
<td>$0 .. 127, 2^{32} - 128$</td>
</tr>
<tr>
<td>glu</td>
<td>$a &gt; b$</td>
<td>$1 .. 128, 2^{32}$</td>
</tr>
<tr>
<td>geu</td>
<td>$a &gt;= b$</td>
<td>$1 .. 128, 2^{32} - 127$</td>
</tr>
</tbody>
</table>
Operation: if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

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

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

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

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

**Interruptions:** Illegal Operation fault
cmpxchg — Compare and Exchange

Format:

(qp) cmpxchg {sz, ldhint} r1 = [r3], r2, ar.cc

(qp) cmp8xchg16 {sem, ldhint} r1 = [r3], r2, ar.cs, ar.cc

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 r3. 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 r2 are written to memory starting at the address specified by the value in GR r3. For cmp8xchg16, if the two are equal, then 8-bytes from GR r2 are stored at the specified address ignoring bit 3 (GR r3 & ~0x8), and 8 bytes from the Compare and Store Data application register (AR[CSD]) are stored at that address + 8 ((GR r3 & ~0x8) + 8). The zero-extended value read from memory is placed in GR r1 and the NaT bit corresponding to GR r1 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>
<thead>
<tr>
<th>sz Completer</th>
<th>Bytes Accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 2-20. Compare and Exchange Semaphore Types

<table>
<thead>
<tr>
<th>sem Completer</th>
<th>Ordering Semantics</th>
<th>Semaphore Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>acq</td>
<td>Acquire</td>
<td>The memory read/write is made visible prior to all subsequent data memory accesses.</td>
</tr>
<tr>
<td>rel</td>
<td>Release</td>
<td>The memory read/write is made visible after all previous data memory accesses.</td>
</tr>
</tbody>
</table>

If the address specified by the value in GR r3 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:

```c
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;
}
```

Interruptions:

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

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

if (PSR.cpl == 0 && PSR.vm == 1)
    virtualization_fault();

alat_frame_update(CFM.sof, 0);
rsr_perserve_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**

**Format:**
- \((qp)\ czx1.l\ r_1 = r_3\)  
  - \(r_3\) in \(one\_byte\_form, left\_form\)  
- \((qp)\ czx1.r\ r_1 = r_3\)  
  - \(r_3\) in \(one\_byte\_form, right\_form\)  
- \((qp)\ czx2.l\ r_1 = r_3\)  
  - \(r_3\) in \(two\_byte\_form, left\_form\)  
- \((qp)\ czx2.r\ r_1 = r_3\)  
  - \(r_3\) in \(two\_byte\_form, right\_form\)

**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**

<table>
<thead>
<tr>
<th>Size</th>
<th>Element Width</th>
<th>Range of Result if Zero Element Found</th>
<th>Default Result if No Zero Element Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8 bit</td>
<td>0-7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>16 bit</td>
<td>0-3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Operation:**

```c
if (PR[qp]) {
    check_target_register(r_1);
    if (one_byte_form) {
        if (left_form) {
            // scan from most significant down
            if ((GR[r_3] & 0xff00000000000000) == 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] & 0xffff000000000000) == 0) GR[r_1] = 0;
            else if ((GR[r_3] & 0x0000ffff00000000) == 0) GR[r_1] = 1;
            else if ((GR[r_3] & 0x00000000ffff0000) == 0) GR[r_1] = 2;
            else if ((GR[r_3] & 0x0000000000ffff00) == 0) GR[r_1] = 3;
            else if ((GR[r_3] & 0x000000000000ffff) == 0) GR[r_1] = 4;
            else if ((GR[r_3] & 0x00000000000000ff) == 0) GR[r_1] = 5;
            else if ((GR[r_3] & 0x0000000000000000) == 0) GR[r_1] = 6;
            else if ((GR[r_3] & 0x000000000000000f) == 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] & 0x00000000ffff0000) == 0) GR[r_1] = 2;
            else if ((GR[r_3] & 0x0000000000ffff00) == 0) GR[r_1] = 3;
            else if ((GR[r_3] & 0x000000000000ffff) == 0) GR[r_1] = 4;
            else if ((GR[r_3] & 0x00000000000000ff) == 0) GR[r_1] = 5;
            else if ((GR[r_3] & 0x0000000000000000) == 0) GR[r_1] = 6;
            else if ((GR[r_3] & 0x000000000000000f) == 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] & 0x00000000000000ff) == 0) GR[r_1] = 1;
        }
    }
}
```
else if ((GR[r3] & 0x0000ffff00000000) == 0) GR[r1] = 2;
else if ((GR[r3] & 0xffff000000000000) == 0) GR[r1] = 3;
else GR[r1] = 4;
}

GR[r2].nat = GR[r3].nat;

Interruptions: Illegal Operation fault
**dep — Deposit**

**Format:**
- 
  \( (qp) \) dep \( r_1 = r_2, r_3, pos_6, len_4 \)  
  merge_form, register_form  
  115
- 
  \( (qp) \) dep \( r_1 = imm_1, r_3 \) pos_6, len_6  
  merge_form, imm_form  
  114
- 
  \( (qp) \) dep.z \( r_1 = r_2, pos_6, len_6 \)  
  zero_form, register_form  
  112
- 
  \( (qp) \) dep.z \( r_1 = imm_8, pos_6, len_6 \)  
  zero_form, imm_form  
  113

**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 \( len_6 \) immediate. Note that \( len_6 \) 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 \( len_6 \) immediate. The \( len_6 \) 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., \( len_6 + pos_6 > 64 \), the most significant \( len_6 + pos_6 - 64 \) bits of the deposited bit field are truncated. The \( len_6 \) immediate is encoded as \( len - 1 \) 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)**

![Figure 2-5. Deposit Example (merge_form)](image)

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)**

![Figure 2-6. Deposit Example (zero_form)](image)
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 u< 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) extr r1 = r3, pos6, len6
(qp) extr.u r1 = r3, pos6, len6

Description: A field is extracted from GR r3, either zero extended or sign extended, and placed right-justified in GR r1. The field begins at the bit position given by the second operand and extends len6 bits to the left. The bit position where the field begins is specified by the pos6 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 r3, the sign is taken from the most significant bit of GR r3. The immediate value len6 can be any number in the range 1 to 64, and is encoded as len6-1 in the instruction. The immediate value pos6 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 \( f_1 = f0, f_3 \)

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.

fadd — Floating-point Add

Format: \((qp)\ fadd.pc.sf\ f_1 = f_3, f_2\)

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

<table>
<thead>
<tr>
<th>(pc) Mnemonic</th>
<th>Precision Specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>.s</td>
<td>single</td>
</tr>
<tr>
<td>.d</td>
<td>double</td>
</tr>
<tr>
<td>none</td>
<td>dynamic</td>
</tr>
<tr>
<td></td>
<td>(i.e. use (pc) value in status field)</td>
</tr>
</tbody>
</table>

Table 2-23. \(sf\) Mnemonic Values

<table>
<thead>
<tr>
<th>(sf) Mnemonic</th>
<th>Status Field Accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>.s0 or none</td>
<td>sf0</td>
</tr>
<tr>
<td>.s1</td>
<td>sf1</td>
</tr>
<tr>
<td>.s2</td>
<td>sf2</td>
</tr>
<tr>
<td>.s3</td>
<td>sf3</td>
</tr>
</tbody>
</table>

Operation: See “fma — Floating-point Multiply Add” on page 3:77.
**famax — Floating-point Absolute Maximum**

**Format:**\( (qp) \text{famax.sf} \ f_1 = f_2, f_3 \)  

**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 \( \{ \text{PR}[qp] \} \) {
  \text{fp}._\text{check}._\text{target}._\text{register}(f_1);
  \text{if} \ (\text{tmp\_isrcode} = \text{fp\_reg\_disabled}(f_1, f_2, f_3, 0))
    \text{disabled\_fp\_register\_fault}(\text{tmp\_isrcode}, 0);

  \text{if} \ (\text{fp\_is\_natval}(\text{FR}[f_2]) \ | \ | \text{fp\_is\_natval}(\text{FR}[f_3])) \{
    \text{FR}[f_1] = \text{NATVAL};
  \} \text{ else } \{
    \text{fminmax\_exception\_fault\_check}(f_2, f_3, sf, &tmp\_fp\_env);
    \text{if} \ (\text{fp\_raise\_fault}(\text{tmp\_fp\_env}))
      \text{fp\_exception\_fault}(\text{fp\_decode\_fault}(\text{tmp\_fp\_env}));

    \text{tmp\_right} = \text{fp\_reg\_read}(\text{FR}[f_2]);
    \text{tmp\_left} = \text{fp\_reg\_read}(\text{FR}[f_3]);
    \text{tmp\_right}._\text{sign} = \text{FP\_SIGN\_POSITIVE};
    \text{tmp\_left}._\text{sign} = \text{FP\_SIGN\_POSITIVE};
    \text{tmp\_bool\_res} = \text{fp\_less\_than}(\text{tmp\_left}, \text{tmp\_right});
    \text{FR}[f_1] = \text{tmp\_bool\_res} \ ? \ \text{FR}[f_2] : \text{FR}[f_3];

    \text{fp\_update\_fpsr}(sf, \text{tmp\_fp\_env});
  \}

  \text{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:**

\[(qp) \text{famin.sf} \quad f_1 = f_2, f_3\]

**Description:**

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

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

If either \(f_2\) or \(f_3\) 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:**

```c
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);
    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));
        tmp_left = fp_reg_read(FR[f2]);
        tmp_right = fp_reg_read(FR[f3]);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        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
**fand — Floating-point Logical And**

**Format:**  
\((qp) \text{ fand } f_1 = f_2, f_3\)  

**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 \(2.063\) (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:**  

```c
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:**

\[(qp) \text{fandcm } f_1 = f_2, f_3\]

**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:**

\[
\text{if } (\text{PR}[qp]) \{
\begin{align*}
&\text{fp\_check\_target\_register}(f_1); \\
&\text{if } (\text{tmp\_isrcode }= \text{fp\_reg\_disabled}(f_1, f_2, f_3, 0)) \\
&\quad \text{disabled\_fp\_register\_fault}(\text{tmp\_isrcode}, 0); \\
&\text{if } (\text{fp\_is\_natval}(\text{FR}[f_2]) \mid \text{fp\_is\_natval}(\text{FR}[f_3])) \{
\begin{align*}
&\quad \text{FR}[f_1] = \text{NATVAL}; \\
&\text{else } \{
\begin{align*}
&\quad \text{FR}[f_1].\text{significand} = \text{FR}[f_2].\text{significand} \& \sim\text{FR}[f_3].\text{significand}; \\
&\quad \text{FR}[f_1].\text{exponent} = \text{FP\_INTEGER\_EXP}; \\
&\quad \text{FR}[f_1].\text{sign} = \text{FP\_SIGN\_POSITIVE}; \\
&\quad \text{fp\_update\_psr}(f_1);
&\}\end{align*}
&\}\end{align*}
\}
\}
\]

**FP Exceptions:** None

**Interruptions:** Illegal Operation fault Disabled Floating-point Register fault
fc — Flush Cache

Format: 

- \((qp)\) fc r\_3\) invalid
- \((qp)\) fc.\_i r\_3\) coherent

Description: In the invalidate_line form, the cache line associated with the address specified by the value of GR r\_3\ is invalidated from all levels of the processor cache hierarchy. The invalidation is broadcast throughout the coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory it is written to memory before invalidation. The line size affected is at least 32-bytes (aligned on a 32-byte boundary). An implementation may flush a larger region.

In the instruction_cache_coherent form, the cache line specified by GR r\_3\ 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: 

```c
if (PR[qp]) {
    itype = NON_ACCESS|FC|READ;
    if (GR[r\_3].nat)
        register_nat_consumption_fault(itype);
    tmp_paddr = tlb_translate_nonaccess(GR[r\_3], itype);

    if (invalidate_line_form)
        mem_flush(tmp_paddr);
    else // instruction_cache_coherent_form
        make_icache_coherent(tmp_paddr);
}
```
<table>
<thead>
<tr>
<th>Interruptions</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register NaT Consumption fault</td>
<td>Data TLB fault</td>
</tr>
<tr>
<td>Unimplemented Data Address fault</td>
<td>Data Page Not Present fault</td>
</tr>
<tr>
<td>Data Nested TLB fault</td>
<td>Data NaT Page Consumption fault</td>
</tr>
<tr>
<td>Alternate Data TLB fault</td>
<td>Data Access Rights fault</td>
</tr>
<tr>
<td>VHPT Data fault</td>
<td></td>
</tr>
</tbody>
</table>
fchkf — Floating-point Check Flags

Format: \((qp)\) fchkf.sf target\(_{25}\)

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

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

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

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

Operation:

```c
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((imm\_{21} << 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(imm\_{21}, 21));
    } else
        speculation_fault(FCHKF, zero_ext(imm\_{21}, 21));
}
```

FP Exceptions: None

Interruptions: Speculative Operation fault taken Branch trap
              Unimplemented Instruction Address trap
fclass — Floating-point Class

Format: \((qp)\) fclass.fcrel.fctype \(p_1, p_2 = f_2, fclass_9\)

Description: The contents of FR \(f_2\) are classified according to the \(fclass_9\) completer as shown in Table 2-25. This produces a boolean result based on whether the contents of FR \(f_2\) agrees with the floating-point number format specified by \(fclass_9\), as specified by the \(fcrel\) completer. This result is written to the two predicate register destinations, \(p_1\) and \(p_2\). The result written to the destinations is determined by the compare type specified by \(fctype\).

The allowed types are Normal (or none) and unc. See Table 2-26 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

<table>
<thead>
<tr>
<th>fcrel</th>
<th>Test Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>FR (f_2) agrees with the pattern specified by (fclass_9) (is a member)</td>
</tr>
<tr>
<td>nm</td>
<td>FR (f_2) does not agree with the pattern specified by (fclass_9) (is not a member)</td>
</tr>
</tbody>
</table>

A number agrees with the pattern specified by \(fclass_9\) if:

- the number is NaTVal and \(fclass_9\) \(\{8\}\) is 1, or
- the number is a quiet NaN and \(fclass_9\) \(\{7\}\) is 1, or
- the number is a signaling NaN and \(fclass_9\) \(\{6\}\) is 1, or
- the sign of the number agrees with the sign specified by one of the two low-order bits of \(fclass_9\), and the type of the number (disregarding the sign) agrees with the number-type specified by the next four bits of \(fclass_9\), as shown in Table 2-25.

Note: An \(fclass_9\) 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

<table>
<thead>
<tr>
<th>(fclass_9)</th>
<th>Class</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Either these cases can be tested for</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0100</td>
<td>NaTVal</td>
<td>@nat</td>
</tr>
<tr>
<td>0x080</td>
<td>Quiet NaN</td>
<td>@qnan</td>
</tr>
<tr>
<td>0x040</td>
<td>Signaling NaN</td>
<td>@snan</td>
</tr>
<tr>
<td>or the OR of the following two cases</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x001</td>
<td>Positive</td>
<td>@pos</td>
</tr>
<tr>
<td>0x002</td>
<td>Negative</td>
<td>@neg</td>
</tr>
<tr>
<td>AND'd with OR of the following four cases</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x004</td>
<td>Zero</td>
<td>@zero</td>
</tr>
<tr>
<td>0x008</td>
<td>Unnormalized</td>
<td>@unorm</td>
</tr>
<tr>
<td>0x010</td>
<td>Normalized</td>
<td>@norm</td>
</tr>
<tr>
<td>0x020</td>
<td>Infinity</td>
<td>@inf</td>
</tr>
</tbody>
</table>
Operation:

```c
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);

               && ((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 \quad 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: \((qp)\ fcmp.\ frel.\ fctype.\ sf\ p_1,\ p_2 = f_2,\ f_3\)

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

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
\(fctype\) & PR\([qp] == 0\) & PR\([qp] == 1\) & Result == 0, No Source NaTVals & Result == 1, No Source NaTVals & One or More Source NaTVals \\
\hline
none & PR\([p_1]\) & PR\([p_2]\) & PR\([p_1]\) & PR\([p_2]\) & PR\([p_1]\) & PR\([p_2]\) \\
unc & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
\hline
\end{tabular}
\caption{Floating-point Comparison Types}
\end{table}

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.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
frel & frel Complete Unabbreviated & Relation & Pseudo-op of & Quiet NaN as Operand Signals Invalid \\
\hline
eq & equal & \(f_2 == f_3\) & & No \\
lt & less than & \(f_2 < f_3\) & & Yes \\
le & less than or equal & \(f_2 <= f_3\) & & Yes \\
gt & greater than & \(f_2 > f_3\) & \(lt \leftrightarrow f_3\) & Yes \\
ge & greater than or equal & \(f_2 >= f_3\) & \(le \leftrightarrow f_3\) & Yes \\
ord & unordered & \(f_2 ? f_3\) & & No \\
neq & not equal & !\(f_2 == f_3\) & eq & \(p_1 \leftrightarrow p_2\) & No \\
nlt & not less than & !\(f_2 < f_3\) & lt & \(p_1 \leftrightarrow p_2\) & Yes \\
nle & not less than or equal & !\(f_2 <= f_3\) & le & \(p_1 \leftrightarrow p_2\) & Yes \\
ngt & not greater than & !\(f_2 > f_3\) & lt & \(f_2 \leftrightarrow f_3\) & \(p_1 \leftrightarrow p_2\) & Yes \\
nge & not greater than or equal & !\(f_2 >= f_3\) & le & \(f_2 \leftrightarrow f_3\) & \(p_1 \leftrightarrow p_2\) & Yes \\
\hline
\end{tabular}
\caption{Floating-point Comparison Relations}
\end{table}
**fcmp**

**Operation:**

```c
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, frl, 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 (frl == 'eq')
            tmp_rel = fp_equal(tmp_fr2, tmp_fr3);
        else if (frl == 'lt')
            tmp_rel = fp_less_than(tmp_fr2, tmp_fr3);
        else if (frl == 'le')
            tmp_rel = fp_lesser_or_equal(tmp_fr2, tmp_fr3);
        else if (frl == 'gt')
            tmp_rel = fp_less_than(tmp_fr3, tmp_fr2);
        else if (frl == 'ge')
            tmp_rel = fp_lesser_or_equal(tmp_fr3, tmp_fr2);
        else if (frl == 'unord')
            tmp_rel = fp_unordered(tmp_fr2, tmp_fr3);
        else if (frl == 'neq')
            tmp_rel = !fp_equal(tmp_fr2, tmp_fr3);
        else if (frl == 'nlt')
            tmp_rel = !fp_less_than(tmp_fr2, tmp_fr3);
        else if (frl == 'nle')
            tmp_rel = !fp_lesser_or_equal(tmp_fr2, tmp_fr3);
        else if (frl == 'ngt')
            tmp_rel = !fp_less_than(tmp_fr3, tmp_fr2);
        else if (frl == '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
fcvt.fx — Convert Floating-point to Integer

Format:

\[(qp) \text{ fcvt.fx, sf } f_1 = f_2\]

\[(qp) \text{ fcvt.fx.trunc, sf } f_1 = f_2\]

\[(qp) \text{ fcvt.fxu, sf } f_1 = f_2\]

\[(qp) \text{ fcvt.fxu.trunc, sf } f_1 = f_2\]

signed_form \hspace{1cm} F10

trunc_form \hspace{1cm} 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:

\[
\text{if (PR[qp])} \{
\text{fp_check_target_register}(f_1);}
\text{if (tmp_isrcode = fp_reg_disabled(f_1, f_2, 0, 0))}
\text{disabled_fp_register_fault(tmp_isrcode, 0);} \]

\[
\text{if (fp_is_natval(FR[f_2]) \{ }
\text{FR[f_1] = NATVAL;}
\text{fp_update_psr(f_1);} \}
\]\[
\text{else} \{
\text{tmp_default_result = fcvt_exception_fault_check(f_2, signed_form, trunc_form, sf, \&tmp_fp_env);} \]
\[
\text{if (fp_raise_fault(tmp_fp_env))}
\text{fp_exception_fault(fp_decode_fault(tmp_fp_env));} \]
\[
\text{if (fp_is_nan(tmp_default_result)) \{ }
\text{FR[f_1].significand = INTEGER_INDEFINITE;}
\text{FR[f_1].exponent = FP_INTEGER_EXP;}
\text{FR[f_1].sign = FP_SIGN_POSITIVE;}
\} \}
\]
\[
\text{else} \{
\text{tmp_res = fp_ieee_rnd_to_int(fp_reg_read(FR[f_2]), \&tmp_fp_env);} \]
\[
\text{if (tmp_res.exponent)}
\text{tmp_res.significand = fp_U64_rsh(}
\text{tmp_res.significand, (FP_INTEGER_EXP - tmp_res.exponent)}); \]
\[
\text{if (signed_form \&\& tmp_res.sign)}
\text{tmp_res.significand = (~tmp_res.significand) + 1;} \]
\]
\[
\text{FR[f_1].significand = tmp_res.significand;}
\text{FR[f_1].exponent = FP_INTEGER_EXP;}
\text{FR[f_1].sign = FP_SIGN_POSITIVE;}
\} \]
\[
\text{fp_update_fpsr(sf, tmp_fp_env);} \]
\[
\text{fp_update_psr(f_1);} \]
\[
\text{if (fp_raise_traps(tmp_fp_env))}
\text{fp_exception_trap(fp_decode_trap(tmp_fp_env));} \}
\} \}
<table>
<thead>
<tr>
<th>FP Exceptions:</th>
<th>FP Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Operation (V)</td>
<td>Inexact (I)</td>
</tr>
<tr>
<td>Denormal/Unnormal Operand (D)</td>
<td></td>
</tr>
<tr>
<td>Software Assist (SWA) fault</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interruptions:</th>
<th>Interruptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal Operation fault</td>
<td>Floating-point Exception fault</td>
</tr>
<tr>
<td>Disabled Floating-point Register fault</td>
<td>Floating-point Exception trap</td>
</tr>
</tbody>
</table>
fcvt.xf — Convert Signed Integer to Floating-point

Format: \((qp) \text{ fcvt.xf } f_t = f_2\)

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 \(f_t\).

If FR \(f_2\) is a NaTVal, FR \(f_t\) is set to NaTVal instead of the computed result.

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

Operation:

```c
if (PR[qp]) {
    fp_check_target_register(f_t);
    if (tmp_isrcode = fp_reg_disabled(f_t, f_2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f_2])) {
        FR[f_t] = NATVAL;
    } else {
        tmp_res = FR[f_2];
        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[f_t].significand = tmp_res.significand;
        FR[f_t].exponent = tmp_res.exponent;
        FR[f_t].sign = tmp_res.sign;
    }
    fp_update_psr(f_t);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault
**fcvt.xuf — Convert Unsigned Integer to Floating-point**

**Format:**  
\[(qp)\text{fcvt.xuf}.pc.sf f_1 = f_3\]  
pseudo-op of: \[(qp)\text{fma}.pc.sf f_1 = f_3, f_1, f_0\]

**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.\ldhint r1 = [r3], inc3 \quad \text{four\_byte\_form} \quad \text{M17}

\[(qp)\) fetchadd8.\sem.\ldhint r1 = [r3], inc3 \quad \text{eight\_byte\_form} \quad \text{M17}

Description:

A value consisting of four or eight bytes is read from memory starting at the address specified by the value in GR \[r3\]. The value is zero extended and added to the sign-extended immediate value specified by \(inc3\). The values that may be specified by \(inc3\) 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 \[r3\]. The zero-extended value read from memory is placed in GR \[r1\] and the NaT bit corresponding to GR \[r1\] 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

<table>
<thead>
<tr>
<th>(sem) Completer</th>
<th>Ordering Semantics</th>
<th>Semaphore Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>acq</td>
<td>Acquire</td>
<td>The memory read/write is made visible prior to all subsequent data memory accesses.</td>
</tr>
<tr>
<td>rel</td>
<td>Release</td>
<td>The memory read/write is made visible after all previous data memory accesses.</td>
</tr>
</tbody>
</table>

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 \[r3\] 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 \(ldhint\) 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.
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_used);
    if (!ma_supports_fetchadd(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);
    if (sem == 'acq')
        val = mem_xchg_add(inc3, paddr, size, UM.be, mattr, ACQUIRE, ldhint);
    else // 'rel'
        val = mem_xchg_add(inc3, paddr, size, UM.be, mattr, RELEASE, ldhint);
    alat_inval_multiple_entries(paddr, size);
    GR[r1] = zero_ext(val, size * 8);
    GR[r1].nat = 0;
}

Interruptions:
Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
flushrs — Flush Register Stack

Format: flushrs

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();
}
```

Interruptions:

| 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: \((qp)\ {fma}.pc.sf\ f_1 = f_3, f_4, f_2\)

Description: The product of \(f_3\) and \(f_4\) is computed to infinite precision and then \(f_2\) 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 \(f_1\).

If any of \(f_3\), \(f_4\), or \(f_2\) is a NaNVal, \(f_1\) is set to NaNVal 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:

\[
\begin{aligned}
&\text{if} \ (PR[qp]) \ {\text{\{}} \\
&\quad \text{fp_check_target_register}(f_1); \\
&\quad \text{if} \ (\text{tmp_isrcode} = \text{fp_reg_disabled}(f_1, f_2, f_3, f_4)) \\
&\quad \quad \text{disabled_fp_register_fault}(\text{tmp_isrcode}, 0); \\
&\quad \text{if} \ (\text{fp_is_natval}(f_2) || \text{fp_is_natval}(f_3)) \ || \text{fp_is_natval}(f_4)) \ {\text{\{} } \\
&\quad \quad \text{FR}[f_1] = \text{NATVAL}; \\
&\quad \quad \text{fp_update_psr}(f_1); \\
&\quad \text{else} \ {\text{\{} } \\
&\quad \quad \text{tmp_default_result} = \text{fma_exception_fault_check}(f_2, f_3, f_4, \\ pc, \ sf, &\text{tmp_fp_env}); \\
&\quad \quad \text{if} \ (\text{fp_raise_fault}(\text{tmp_fp_env}) \\
&\quad \quad \quad \text{fp_exception_fault}(\text{fp_decode_fault}(\text{tmp_fp_env})); \\
&\quad \quad \text{if} \ (\text{fp_is_nan_or_inf}(\text{tmp_default_result})) \ {\text{\{} } \\
&\quad \quad \quad \text{FR}[f_1] = \text{tmp_default_result}; \\
&\quad \quad \text{else} \ {\text{\{} } \\
&\quad \quad \quad \text{tmp_res} = \text{fp_mul}(\text{fp_reg_read}(f_3), \text{fp_reg_read}(f_4)); \\
&\quad \quad \quad \text{if} \ (f_2 != 0) \\
&\quad \quad \quad \quad \text{tmp_res} = \text{fp_add}(\text{tmp_res}, \text{fp_reg_read}(f_2), \text{tmp_fp_env}); \\
&\quad \quad \quad \quad \text{FR}[f_1] = \text{fp_ieee_round}(\text{tmp_res}, &\text{tmp_fp_env}); \\
&\quad \quad \text{\}} \\
&\quad \text{fp_update_fpsr}(sf, \text{tmp_fp_env);} \\
&\quad \text{fp_update_psr}(f_1); \\
&\quad \text{if} \ (\text{fp_raise_traps}(\text{tmp_fp_env}) \\
&\quad \quad \text{fp_exception_trap}(\text{fp_decode_trap}(\text{tmp_fp_env})); \\
&\quad \text{\} } \\
&\text{\}} \\
\end{aligned}
\]

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
- Disabled Floating-point Register fault
- Floating-point Exception fault
- Floating-point Exception trap
**fmax — Floating-point Maximum**

**Format:**
(qp) fmax.

**Description:**
The operand with the larger value is placed in FR.

- If FR equals FR,

  - FR gets FR.

- If either FR or FR is a NaN, FR gets FR.

- If either FR or FR is a NaTVal, FR 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:**

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

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

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

        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
**fmerge — Floating-point Merge**

**Format:**

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

**Description:**

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

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

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

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

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

---

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

![Figure 2-8](image1.png)

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

![Figure 2-9](image2.png)

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

![Figure 2-10](image3.png)
Operation:

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

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

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault
**fmin — Floating-point Minimum**

**Format:** \((qp) \text{fmin} \ f_1 = f_2, f_3\)

**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:**

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

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

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

        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
**fmix — Floating-point Mix**

**Format:**

\[(qp) \text{ fmix.l } f_1 = f_2, f_3\]
\[(qp) \text{ fmix.r } f_1 = f_2, f_3\]
\[(qp) \text{ fmix.lr } f_1 = f_2, f_3\]

**mix_l_form** F9
**mix_r_form** F9
**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**

```
FR f2 81 80 64 63 32 31 0
FR f3                 32 31 0
FR f1 1003E
```

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

```
FR f2 81 80 64 63 32 31 0
FR f3                 32 31 0
FR f1 1003E
```

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

```
FR f2 81 80 64 63 32 31 0
FR f3                 32 31 0
FR f1 1003E
```
Operation:

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

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

FP Exceptions: None

Interruptions: Illegal Operation fault

Disabled Floating-point Register fault
**fmpy — Floating-point Multiply**

**Format:**

\[(qp \text{ fmpy}.pc.sf) f1 = f3, f4\]

pseudo-op of: \((qp \text{ fma}.pc.sf) f1 = f3, f4, f0\)

**Description:**
The product \(FR_f3\) and \(FR_f4\) 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 \(FR_f1\).

If either \(FR_f3\) or \(FR_f4\) is a NaTVal, \(FR_f1\) 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.
\textbf{fms — Floating-point Multiply Subtract}

**Format:** \((\text{qp}) \ fms.pc sf \ f_1 = f_3, f_4, f_2\)

**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 \(f_0\), an IEEE multiply operation is performed instead of a multiply and subtract. See “\textit{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:**

```plaintext
if (PR[\text{qp}]) {
    \text{fp_check_target_register}(f_1);
    if (tmp_isrcode = \text{fp_reg_disabled}(f_1, f_2, f_3, f_4))
        \text{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])) {
        \text{FR}[f_1] = \text{NATVAL};
        \text{fp_update_psr}(f_1);
    } else {
        \text{tmp_default_result} = \text{fms_fnma_exception_fault_check}(f_2, f_3, f_4, pc, sf, \&\text{tmp_fp_env});
        if (fp_raise_fault(tmp_fp_env))
            \text{fp_exception_fault}\left(\text{fp_decode_fault}(tmp_fp_env)\right);
        if (fp_is_nan_or_inf(tmp_default_result)) {
            \text{FR}[f_1] = \text{tmp_default_result};
        } else {
            \text{tmp_res} = \text{fp_mul}(%fr_reg_read(FR[f_3]), %fr_reg_read(FR[f_4]));
            tmp_fr2 = %fr_reg_read(FR[f_2]);
            tmp_fr2.sign = !tmp_fr2.sign;
            if (f_2 != 0)
                \text{tmp_res} = \text{fp_add}(tmp_res, tmp_fr2, \&\text{tmp_fp_env});
            \text{FR}[f_1] = \text{fp_ieee_round}(tmp_res, \&\text{tmp_fp_env});
        }
    }
    \text{fp_update_fpsr}(sf, \&\text{tmp_fp_env});
    \text{fp_update_psr}(f_1);
    if (fp_raise_traps(tmp_fp_env))
        \text{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
Disabled Floating-point Register fault

Floating-point Exception fault
Floating-point Exception trap
fneg — Floating-point Negate

Format: \((qp)\) fneg \(f_1 = f_3\)  
pseudo-op of: \((qp)\) fmerge.ns \(f_1 = f_2, f_3\)

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: \((qp)\) fnegabs \(f_1 = f_3\)

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.

fnma — Floating-point Negative Multiply Add

Format: \((qp)\) fnma.pc.sf \(f_1 = f_3, f_4, f_2\)

Description: The product of \(f_3\) and \(f_4\) is computed to infinite precision, negated, and then \(f_2\) 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 \(f_1\).

If any of \(f_3\), \(f_4\), or \(f_2\) is a NaTVal, \(f_1\) 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:

```c
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);
    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fms_fnma_exception_fault_check(f2, f3, f4,
            pc, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));
        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[f3]), fp_reg_read(FR[f4]));
            tmp_res.sign = !tmp_res.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read(FR[f2]), tmp_fp_env);
            FR[f1] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }
        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions: Invalid Operation (V) 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
fnmpy

fnmpy — Floating-point Negative Multiply

Format: (qp) fnmpy.pc.sf f1 = f3, f4  
        pseudo-op of: (qp) fnma.pc.sf f1 = f3, f4,f0

Description:  The product FR f3 and FR f4 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 f1.

   If either FR f3 or FR f4 is a NaTVal, FR f1 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.

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: \( (\text{qp}) \text{ for } f_1 = f_2, f_3 \)

Description: The bit-wise logical OR of the significand fields of \( f_2 \) and \( f_3 \) is computed. The resulting value is stored in the significand field of \( f_1 \). The exponent field of \( f_1 \) is set to the biased exponent for \( 2^{0.63} (0x1003E) \) and the sign field of \( f_1 \) is set to positive (0).

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

Operation:

\[
\text{if (FR[qp])} \{
\text{fp_check_target_register}(f_1);
\text{if (tmp_isrcode = fp_reg_disabled(f_1, f_2, f_3, 0))}
\text{disabled_fp_register_fault(tmp_isrcode, 0);}
\text{if (fp_is_natval(FR[f_2]) || fp_is_natval(FR[f_3])))} \{
\text{FR[f_1] = NATVAL;}
\text{else} \{
\text{FR[f_1].significand = FR[f_2].significand | FR[f_3].significand;}
\text{FR[f_1].exponent = FP_INTEGER_EXP;}
\text{FR[f_1].sign = FP_SIGN_POSITIVE;}
\}
\text{fp_update_psr(f_1);}
\}
\]

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault
fpabs — Floating-point Parallel Absolute Value

**Format:**

\[(qp)\text{ fpabs } t_1 = t_3\]  \hspace{1cm} \text{pseudo-op of: } (qp)\text{ fpmerge.s } t_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:**

\[(qp) \text{fp} \text{pack } f_1 = f_2, f_3\]

**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**

![Diagram of floating-point pack]

**Operation:**

```c
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 \(t_1 = t_2, t_3\)

Description: The paired single precision values in the significands of \(F_1\) and \(F_1\) are returned. The operands with the larger absolute value are returned in the significand field of \(F_1\).

If the magnitude of high (low) \(F_3\) is less than the magnitude of high (low) \(F_2\), high (low) \(F_1\) gets high (low) \(F_2\). Otherwise high (low) \(F_1\) gets high (low) \(F_3\).

If high (low) \(F_2\) or high (low) \(F_3\) is a NaN, and neither \(F_2\) or \(F_3\) is a NaTVal, high (low) \(F_1\) gets high (low) \(F_3\).

The exponent field of \(F_1\) is set to the biased exponent for \(2.063\) (0x1003E) and the sign field of \(F_1\) is set to positive (0).

If either \(F_2\) or \(F_3\) is a NaTVal, \(F_1\) 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:
```c
if (PR[gp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);
    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raiseFault(tmp_fp_env))
            fp_exception_fault(fp_decodeFault(tmp_fp_env));
    }
    tmp_fr2 = tmp_right = fp_reg_read_hi(f2);
    tmp_fr3 = tmp_left = fp_reg_read_hi(f3);
    tmp_right.sign = FP_SIGN_POSITIVE;
    tmp_left.sign = FP_SIGN_POSITIVE;
    tmp_bool_res = fp_less_than(tmp_left, tmp_right);
    tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2 : tmp_fr3);
    tmp_fr2 = tmp_right = fp_reg_read_lo(f2);
    tmp_fr3 = tmp_left = fp_reg_read_lo(f3);
    tmp_right.sign = FP_SIGN_POSITIVE;
    tmp_left.sign = FP_SIGN_POSITIVE;
    tmp_bool_res = fp_less_than(tmp_left, tmp_right);
    tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2 : tmp_fr3);
    FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
    FR[f1].exponent = FP_INTEGER_EXP;
    FR[f1].sign = FP_SIGN_POSITIVE;
    fp_update_fpsr(sf, tmp_fp_env);
    } else {
        fp_update_psr(f1);
    }
```
FP Exceptions: Invalid Operation (V)  
Denormal/Unnormal Operand (D)  
Software Assist (SWA) fault

Interruptions:  Illegal Operation fault  
Disabled Floating-point Register fault  
Floating-point Exception fault
## fpamin — Floating-point Parallel Absolute Minimum

**Format:**
\[(qp) \text{fpamin.sf} f_1 = f_2, f_3\]

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

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

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

The exponent field of \(f_1\) is set to the biased exponent for 2.063 (0x1003E) and the sign field of \(f_1\) is set to positive (0).

If either \(f_2\) or \(f_3\) is NaNVal, \(f_1\) is set to NaNVal 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:**
```c
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        fp_exception_fault(fp_decode_fault(tmp_fp_env));
    }
}
```
```c
tmp_fr2 = tmp_left = fp_reg_read_hi(f2);
tmp_fr3 = tmp_right = fp_reg_read_hi(f3);
tmp_left.sign = FP_SIGN_POSITIVE;
tmp_right.sign = FP_SIGN_POSITIVE;
tmp_bool_res = fp_less_than(tmp_left, tmp_right);
tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);
```
```c
tmp_fr2 = tmp_left = fp_reg_read_lo(f2);
tmp_fr3 = tmp_right = fp_reg_read_lo(f3);
tmp_left.sign = FP_SIGN_POSITIVE;
tmp_right.sign = FP_SIGN_POSITIVE;
tmp_bool_res = fp_less_than(tmp_left, tmp_right);
tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);
```
```
FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
FR[f1].exponent = FP_INTEGER_EXP;
FR[f1].sign = FP_SIGN_POSITIVE;
```
```
fp_update_fpsr(sf, tmp_fp_env);
```
FP 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.\(frf, \sf1= \sf2, \sf3\)

Description: The two pairs of single precision source operands in the significand fields of FR \(\sf2\) and FR \(\sf3\) are compared for one of twelve relations specified by \(frf\). 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 \(\sf1\). The exponent field of FR \(\sf1\) is set to the biased exponent for 2.0\(^{63}\) (0x1003E) and the sign field of FR \(\sf1\) is set to positive (0).

Table 2-29. Floating-point Parallel Comparison Results

<table>
<thead>
<tr>
<th>PR([qp])==0</th>
<th>PR([qp])==1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result==false, No Source NaTVals</td>
<td>Result==true, No Source NaTVals</td>
</tr>
<tr>
<td>unchanged 0...0</td>
<td>1...1</td>
</tr>
</tbody>
</table>

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 \(\sf2\) or FR \(\sf3\) is a NaTVal, FR \(\sf1\) is set to NaTVal instead of the computed result.

Table 2-30. Floating-point Parallel Comparison Relations

<table>
<thead>
<tr>
<th>(frf)</th>
<th>(frf) Completer Unabbreviated</th>
<th>Relation</th>
<th>Pseudo-op of</th>
<th>Quiet NaN as Operand Signals Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>equal</td>
<td>(f2 == f3)</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>lt</td>
<td>less than</td>
<td>(f2 &lt; f3)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>le</td>
<td>less than or equal</td>
<td>(f2 \leq f3)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>gt</td>
<td>greater than</td>
<td>(f2 &gt; f3)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ge</td>
<td>greater than or equal</td>
<td>(f2 \geq f3)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>unord</td>
<td>unordered</td>
<td>(f2 ? f3)</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>neq</td>
<td>not equal</td>
<td>((f2 == f3))</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>nlt</td>
<td>not less than</td>
<td>((f2 &lt; f3))</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>nie</td>
<td>not less than or equal</td>
<td>((f2 \leq f3))</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ngl</td>
<td>not greater than</td>
<td>((f2 &gt; f3))</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>nge</td>
<td>not greater than or equal</td>
<td>((f2 \geq f3))</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ord</td>
<td>ordered</td>
<td>((f2 ? f3))</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>
fpcmp

Operation:

```c
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_less_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_less_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_less_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_less_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_less_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_less_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_less_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_less_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

Format:

\[
\begin{align*}
(qp) \ fpcvt.fx.sf & \ f_1 = f_2 & \text{signed form} & \ F10 \\
(qp) \ fpcvt.fx.trunc.sf & \ f_1 = f_2 & \text{signed form, trunc form} & \ F10 \\
(qp) \ fpcvt.fxu.sf & \ f_1 = f_2 & \text{unsigned form} & \ F10 \\
(qp) \ fpcvt.fxu.trunc.sf & \ f_1 = f_2 & \text{unsigned form, trunc form} & \ F10 \\
\end{align*}
\]

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:

```c
if (PR[gp]) {
    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)
- Inexact (I)
- Denormal/Unormal Operand (D)
- Software Assist (SWA) Fault
**fpcvt.fx**

<table>
<thead>
<tr>
<th>Interruptions:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal Operation fault</td>
<td>Floating-point Exception fault</td>
</tr>
<tr>
<td>Disabled Floating-point Register fault</td>
<td>Floating-point Exception trap</td>
</tr>
</tbody>
</table>

*Volume 3: Instruction Reference*
fpma — Floating-point Parallel Multiply Add

Format: \((qp) \text{ fpma} sf\ f_1 = f_3, f_4, f_2\)

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 \(f_1\), 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:**

\[(qp) \text{fpmax.sf } f_1 = f_2, f_3\]  

**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 \(f_2\) 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 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:**

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

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

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

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

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

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

**FP Exceptions:**
- Invalid Operation (V)
- Denormal/Unnormal Operand (D)
- Software Assist (SWA) fault
**Interruptions:**
- Illegal Operation fault
- Disabled Floating-point Register fault
- Floating-point Exception fault
fpmerge — Floating-point Parallel Merge

Format:

- \((aq)\) fpmerge.ns \(f_1 = f_2, f_3\)  
  neg_sign_form \(F9\)
- \((aq)\) fpmerge.s \(f_1 = f_2, f_3\)  
  sign_form \(F9\)
- \((aq)\) 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**
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: \( (qp) \) \( fpmin.sf \ f_1 = f_2, f_3 \)

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

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

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

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

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

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

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

Operation:

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

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

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

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

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

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

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault
**Interceptions:**
- Illegal Operation fault
- Disabled Floating-point Register fault
- Floating-point Exception fault
fpmpy — Floating-point Parallel Multiply

Format:  \((qp) \text{ fpmpy} sf \ t_1 = t_3, t_4\)  

pseudo-op of:  \((qp) \text{ fpma} sf \ t_1 = t_3, t_4, f0\)

Description:  The pair of products of the pairs of single precision values in the significand fields of FR \(t_3\) and FR \(t_4\) 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 \(t_1\). The exponent field of FR \(t_1\) is set to the biased exponent for \(2.0^{63}\) (0x1003E) and the sign field of FR \(t_1\) is set to positive (0).

If either FR \(t_3\), or FR \(t_4\) is a NaTVal, FR \(t_1\) 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.

fpms — Floating-point Parallel Multiply Subtract

Format: \((qp)\) fpms.sf \(f_1 = f_3, f_4, f_2\)

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:

```c
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_fpnm_a_exception_fault_check(f2, f3,
                                                                 f4, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

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

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

```c
    tmp_res_lo = fp_ieee_round_sp(tmp_res, LOW, &tmp_fp_env);

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

    fp_update_fpsr(sf, tmp_fp_env);
    fp_update_psr(f);
    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
```
fpneg — Floating-point Parallel Negate

Format: \((qp) \text{ fpneg } f_1 = f_3\)  
pseudo-op of: \((qp) \text{ fpmerge.ns } f_1 = f_2, f_3\)

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

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

Operation: See “fpmerge — Floating-point Parallel Merge” 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: \((qp) \, \text{fpnma} \, sf \, t_1 = t_3, t_4, t_2\)

Description: The pair of products of the pairs of single precision values in the significand fields of FR \(t_3\) and FR \(t_4\) are computed to infinite precision, negated, and then the pair of single precision values in the significand field of FR \(t_2\) 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.\(sf.\text{rc}\). The pair of rounded results are stored in the significand field of FR \(t_1\). The exponent field of FR \(t_1\) is set to the biased exponent for \(2.0^{63}\) (0x1003E) and the sign field of FR \(t_1\) is set to positive (0).

If any of FR \(t_3\), FR \(t_4\), or FR \(t_2\) is a NaNVal, FR \(t_1\) is set to NaNVal instead of the computed result.

Note: If \(t_2\) 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(f3, f2, f4, sf, &tmp_fp_env);
        if (fp_raise Fault(tmp_fp_env))
            fp_exception_fault(f3, Fault); // this is the fault type
        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_div(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_div(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);
    }
}

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
fpnmpy — Floating-point Parallel Negative Multiply

Format: \((qp)\ fpnmpy.sf\ f_1 = f_3, f_4\)

pseudo-op of: \((qp)\ fpnma.sf\ f_1 = f_3, f_4,f0\)

Description: The pair of products of the pairs of single precision values in the significand fields of FR
\(f_3\) and FR \(f_4\) are computed to infinite precision and then negated. The resulting values
are then rounded to single precision using the rounding mode specified by FPSR.\(sf,rc\).
The pair of rounded results are stored in the significand field of FR \(f_1\). The exponent field
of FR \(f_1\) is set to the biased exponent for 2.063 (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 “fpnma — Floating-point Parallel Negative Multiply Add” on page 3:120.
fprcpa — Floating-point Parallel Reciprocal Approximation

Format: \((qp)\) fprcpa.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:

- 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 2.063 (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 NaN 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:

```c
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_INFINITY;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_hi = 0;
            } else {
                // More cases...
            }
        }
    }
}
```
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
**fprsqrta — Floating-point Parallel Reciprocal Square Root Approximation**

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

**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:

- 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 fprsqrta 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 2.63 (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:**

```c
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;
        FR[p2] = 0;
    } else {
        tmp_default_result_pair = fprsqrta_exception_fault_check(f3, sf,
                                                                   &tmp_fp_env, &limits_check);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));
        if (fp_is_nan(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
            tmp_pred_hi = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read_hi(f3));
            if (fp_is_zero(tmp_fr3)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = tmp_fr3.sign;
                tmp_pred_hi = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                tmp_res = FP_ZERO;
                tmp_pred_hi = 0;
            } else {
                tmp_res = fp_ieee_recip_sqrt(tmp_fr3);
            }
        }
    }
}
```
if (limits_check.hi)
    tmp_pred_hi = 0;
else
    tmp_pred_hi = 1;
}

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

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

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

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

**Interruptions:**
- Illegal Operation fault
- Floating-point Exception fault
- Disabled Floating-point Register fault
**frcpa — Floating-point Reciprocal Approximation**

**Format:**
\[(qp) \text{frcpa.sf } t_1, p_2 = t_2, t_3\]

**Description:**
If PR \(qp\) is 0, PR \(p_2\) is cleared and FR \(t_1\) remains unchanged.

If PR \(qp\) is 1, the following will occur:

- FR \(t_1\) is either set to an approximation (with a relative error < \(2^{-8.886}\)) of the reciprocal of FR \(t_3\), 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 \(t_1\) is set to the approximation of the reciprocal of FR \(t_3\), 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 \(t_3\)'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 \(t_1\) 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:**
```c
if (FR[qp]) {
    fp_check_target_register(t_1);
    if (tmp_isrcode = fp_reg_disabled(t_1, t_2, t_3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f_2]) || fp_is_natval(FR[f_3])) {
        FR[t_1] = NATVAL;
        PR[p_2] = 0;
    } else {
        tmp_default_result = frcpa_exception_fault_check(t_2, t_3, 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;
            PR[p_2] = 0;
        } else {
            num = fp_normalize(fp_reg_read(FR[f_2]));
            den = fp_normalize(fp_reg_read(FR[f_3]));
            if (fp_is_inf(num) && fp_is_finite(den)) {
                FR[f_1] = FP_INFINITY;
                FR[f_1].sign = num.sign ^ den.sign;
                PR[p_2] = 0;
            } else if (fp_is_finite(num) && fp_is_inf(den)) {
                FR[f_1] = FP_ZERO;
                FR[f_1].sign = num.sign ^ den.sign;
                PR[p_2] = 0;
            } else if (fp_is_zero(num) && fp_is_finite(den)) {
                FR[f_1] = FP_ZERO;
                FR[f_1].sign = num.sign ^ den.sign;
                PR[p_2] = 0;
            }
        }
    }
}
```
} else {
    FR[fj] = fp_ieee_recip(den);
    PR[p2] = 1;
}

} else {
    PR[p2] = 0;
}

// fp_ieee_recip()

fp_ieee_recip(den)
{
    RECIP_TABLE[256] = {
        0x3fc, 0x3f4, 0x3ec, 0x3e4, 0x3dd, 0x3d5, 0x3cd, 0x3c6,
        0x3be, 0x3a8, 0x3a1, 0x399, 0x392, 0x38b,
        0x384, 0x37d, 0x376, 0x36f, 0x361, 0x35b, 0x354,
        0x34d, 0x346, 0x340, 0x339, 0x333, 0x32c, 0x326, 0x320,
        0x319, 0x333, 0x30d, 0x307, 0x300, 0x2fa, 0x2f4, 0x2ee,
        0x2e8, 0x2e2, 0x2dc, 0x2d7, 0x2d1, 0x2cb, 0x2c5, 0x2bf,
        0x2ba, 0x2b4, 0x2af, 0x2a9, 0x2a3, 0x29e, 0x299, 0x293,
        0x28e, 0x283, 0x27e, 0x279, 0x273, 0x26e, 0x269,
        0x264, 0x25f, 0x25a, 0x255, 0x250, 0x24f, 0x246, 0x241,
        0x23c, 0x237, 0x232, 0x22e, 0x229, 0x224, 0x21f, 0x21b,
        0x216, 0x211, 0x20d, 0x208, 0x204, 0x1ff, 0x1fb, 0x1f6,
        0x1f2, 0x1ed, 0x1e9, 0xe5, 0xe0, 0xd8, 0xd4,
        0x1cf, 0x1cb, 0x1c7, 0x1c3, 0x1bf, 0xb6, 0xb2,
        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, 0x0e9, 0x0e5, 0xe2, 0xdf, 0xdc, 0xd9, 0xd6,
        0x0d3, 0x0d0, 0x0dc, 0x0ca, 0xc8, 0xc5, 0xc2, 0xbf,
        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
- Floating-point Exception fault
- Disabled Floating-point Register fault
frsqrta — Floating-point Reciprocal Square Root Approximation

Format: \((qp)\) frsqrta \(f1, p2 = f3\)

Description: If PR \(qp\) is 0, PR \(p2\) is cleared and FR \(f1\) remains unchanged.

If PR \(qp\) is 1, the following will occur:

- FR \(f1\) is either set to an approximation (with a relative error \(< 2^{-8.831}\)) of the reciprocal square root of FR \(f3\), or set to the IEEE-754 mandated square root of FR \(f3\) — if FR \(f3\) is in the set \{-Infinity, -Finite, -0, Pseudo-zero, +0, +Infinity, NaN, Unsupported\}.
- If FR \(f3\) is set to an approximation of the reciprocal square root of FR \(f3\), then PR \(p2\) is set to 1; otherwise, it is set to 0.
- If FR \(f3\) 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 \(f1\), and set PR \(p2\) to 0.
- If FR \(f3\) is a NaTVal, FR \(f1\) is set to NaTVal instead of the computed result, and PR \(p2\) is cleared.

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

Operation:

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

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

        if (fp_is_nan(tmp_default_result)) {
            FR[f1] = tmp_default_result;
            PR[p2] = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read(FR[f3]));
            if (fp_is_zero(tmp_fr3)) {
                FR[f1] = tmp_fr3;
                PR[p2] = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                FR[f1] = tmp_fr3;
                PR[p2] = 0;
            } else {
                FR[f1] = fp_ieee_recip_sqrt(tmp_fr3);
                PR[p2] = 1;
            }
        }
    }
    fp_update_fpsr(sf, tmp_fp_env);
}
```
fp_update_psr(fj);
} else {
    PR[pj] = 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, 0xca, 0xc7, 0xc3, 0xc0, 0xbd, 0xb9,
0x0b6, 0x0b3, 0xb0, 0xad, 0xa9, 0xa6, 0xa3, 0xa0,
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, 0x33c, 0x336, 0x330, 0x32b, 0x325,
0x31f, 0x31a, 0x314, 0x30f, 0x309, 0x304, 0x2fe,
0x2e9, 0x2f4, 0x2ee, 0xe9, 0xe4, 0xe2d, 0xe2a, 0xe2d,
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, 0x1e8, 0x1e6, 0x1e3, 0x1df, 0x1dc, 0x1d9,
0x1d6, 0x1d3, 0x1d0, 0x1c8, 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

fselect — Floating-point Select

Format: \((qp)\) fselect \(f_3 = f_2, f_4\)

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 2.0^3 (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:

```c
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:**  
\[(qp) \text{ fsetc} sf \text{ amask}_7, \text{ omask}_7\]

**Description:** The status field's control bits are initialized to the value obtained by logically AND-ing the \(sf0\text{.controls}\) and \(\text{amask}_7\) immediate field and logically OR-ing the \(\text{omask}_7\) immediate field.  

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

**Operation:**  
\[
\text{if (PR[qp])} \{ \\
\quad \text{tmp\_controls = (AR[FPSR].sf0\text{.controls} & amask}) | \text{omask};} \\
\quad \text{if (is\_reserved\_field(FSETC, sf, tmp\_controls))} \\
\quad \quad \text{reserved\_register\_field\_fault();} \\
\quad \text{fp\_set\_sf\_controls(sf, tmp\_controls);} \\
\}
\]

**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**

**Format:**

\[(qp) \text{ fswap } f_1 = f_2, f_3\]  
\[(qp) \text{ fswap.nl } f_1 = f_2, f_3\]  
\[(qp) \text{ fswap.nr } f_1 = f_2, f_3\]

**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**
**Operation:**

```c
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:**

\[(qp) \text{ fsxt.l } f_1 = f_2, f_3\]  
\[(qp) \text{ fsxt.r } f_1 = f_2, f_3\]

\[\text{sxt}_\text{I}_\text{form} \quad \text{F9}\]
\[\text{sxt}_\text{R}_\text{form} \quad \text{F9}\]

**Description:**

For the \(\text{sxt}_\text{I}\) form (\(\text{sxt}_\text{R}\) form), the sign of the left (right) single precision value in \(f_2\) is extended to 32-bits and is concatenated with the left (right) single precision value in \(f_3\).

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

For all forms, if either \(f_2\) or \(f_3\) is a NaN, \(f_1\) is set to NaN instead of the computed result.

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

**Figure 2-22. Floating-point Sign Extend Right**
Operation: if (FR[gp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (sxt_l_form) {
            tmp_res_hi = (FR[f2].significand{63} ? 0xFFFFFFFF : 0x00000000);
            tmp_res_lo = FR[f3].significand{63:32};
        } else { // sxt_r_form
            tmp_res_hi = (FR[f2].significand{31} ? 0xFFFFFFFF : 0x00000000);
            tmp_res_lo = FR[f3].significand{31:0};
        }

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

fp_update_psr(f1);

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault
**fwb — Flush Write Buffers**

**Format:**  
(qp) fwb

**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: \((qp) \text{ fxor } f_1 = f_2, f_3\)

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:

\[
\text{if (PR[qp]) } \{
\text{fp_check_target_register}(f_1); \\
\text{if (tmp_isrcode = fp_reg_disabled}(f_1, f_2, f_3, 0)) \\
\text{disabled_fp_register_fault(tmp_isrcode, 0);} \\
\text{if (fp_is_natval(FR[f_2]) || fp_is_natval(FR[f_3]))) } \{
\text{FR[f_1] = NATVAL;} \\
\text{else } \{
\text{FR[f_1].significand = FR[f_2].significand ^ FR[f_3].significand;} \\
\text{FR[f_1].exponent = FP_INTEGER_EXP;} \\
\text{FR[f_1].sign = FP_SIGN_POSITIVE;} \\
\text{fp_update_psr(f_1);} \\
\text{\}}
\]

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault
getf — Get Floating-point Value or Exponent or Significand

Format:

- \((qp)\) getf.s \(r_1 = f_2\)  \hspace{1cm} \text{single\_form} \hspace{0.5cm} M19
- \((qp)\) getf.d \(r_1 = f_2\)  \hspace{1cm} \text{double\_form} \hspace{0.5cm} M19
- \((qp)\) getf.exp \(r_1 = f_2\)  \hspace{1cm} \text{exponent\_form} \hspace{0.5cm} M19
- \((qp)\) getf.sig \(r_1 = f_2\)  \hspace{1cm} \text{significand\_form} \hspace{0.5cm} 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 195, 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**

[Diagram showing the 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**

[Diagram showing the function of getf.sig]

For all forms, if FR \(f_2\) contains a NaTVal, then the NaT bit corresponding to GR \(r_1\) is set to 1.
Operation: if (PR[qp]) {
    check_target_register(r1);
    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);
    if (single_form) {
        GR[r1]{31:0} = fp_fr_to_mem_format(FR[f2], 4, 0);
        GR[r1]{63:32} = 0;
    } else if (double_form) {
        GR[r1] = fp_fr_to_mem_format(FR[f2], 8, 0);
    } else if (exponent_form) {
        GR[r1]{63:18} = 0;
        GR[r1]{16:0} = FR[f2].exponent;
        GR[r1]{17} = FR[f2].sign;
    } else // significand_form
        GR[r1] = FR[f2].significand;
    if (fp_is_natval(FR[f2]))
        GR[r1].nat = 1;
    else
        GR[r1].nat = 0;
}

Interruptions: Illegal Operation fault          Disabled Floating-point Register fault
**hint — Performance Hint**

**Format:**

- \((qp)\) hint imm_{21}
- \((qp)\) hint.i imm_{21}
- \((qp)\) hint.b imm_{21}
- \((qp)\) hint.m imm_{21}
- \((qp)\) hint.f imm_{21}
- \((qp)\) hint.x imm_{62}

**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 \textit{nop} instruction except for its performance effects.

The immediate, \(imm_{21}\) or \(imm_{62}\), specifies the hint. For the \textit{x_unit_form}, the L slot of the bundle contains the upper 41 bits of \(imm_{62}\).

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

**Table 2-31. Hint Immediates**

<table>
<thead>
<tr>
<th>(imm_{21}) or (imm_{62})</th>
<th>Mnemonic</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>@pause</td>
<td>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 hint @pause; 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.</td>
</tr>
<tr>
<td>0x1</td>
<td>@priority</td>
<td>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.</td>
</tr>
<tr>
<td>0x02-0x3f</td>
<td></td>
<td>These values are available for future architected extensions and will execute as a \textit{nop} on all current processors. Use of these values may cause unexpected performance issues on future processors and should not be used.</td>
</tr>
<tr>
<td>other</td>
<td></td>
<td>Implementation specific. Performs an implementation-specific hint action. Consult processor model-specific documentation for details.</td>
</tr>
</tbody>
</table>

**Operation:**

```c
if (PR[gp]) {
    if (x_unit_form)
        hint = imm_{62};
    else // i_unit_form || b_unit_form || b_unit_form || f_unit_form
        hint = imm_{21};

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

**Interruptions:** None
### invala — Invalidate ALAT

**Format:**
- `(qp) invala complete_form M24`
- `(qp) invala.e r1 gr_form, entry_form M26`
- `(qp) invala.e f1 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 `r1` (gr_form), or the floating-point register specifier `f1` (fr_form), and if any ALAT entry matches, it is invalidated.

**Operation:**
```c
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 r2`  
  - instruction_form  
  - M41
- `(qp) itc.d r2`  
  - 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:

```c
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);
    }
}
```

Interruptions:

- Machine Check abort
- Reserved Register/Field fault
- Illegal Operation fault
- Unimplemented Data Address fault
- Privileged Operation fault
- Virtualized fault
- Register NaT Consumption fault

Serialization:

For the instruction_form, software must issue an instruction serialization operation before a dependent instruction fetch access. For the data_form, software must issue a data serialization operation before issuing a data access or non-access reference dependent on the new translation.
itr — Insert Translation Register

**Format:**

- `(qp) itr [r3] = r2`  
  Instruction form  
  M42
- `(qp) itr [r3] = r2`  
  Data form  
  M42

**Description:**

A translation is inserted into the instruction or data translation register specified by the contents of GR\(_r3\). GR\(_r2\) 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\(_r3\) 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:**

```c
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);
    }
}
```
### Interruptions:
- Machine Check abort
- Illegal Operation fault
- Privileged Operation fault
- Register NaT Consumption fault
- Reserved Register/Field fault
- Unimplemented Data Address fault
- Virtualization fault

### Serialization:
For the instruction_form, software must issue an instruction serialization operation before a dependent instruction fetch access. For the data_form, software must issue a data serialization operation before issuing a data access or non-access reference dependent on the new translation.

### Notes:
The processor may use invalid translation registers for translation cache entries. Performance can be improved on some processor models by ensuring translation registers are allocated beginning at translation register zero and continuing contiguously upwards.
ld — Load

Format:

\[(q)\]  
\[
\text{ldsz}., \text{ldtype}, \text{ldhint} \quad r_1 = [r_3] \\
\text{ldsz}., \text{ldtype}, \text{ldhint} \quad r_1 = [r_3], r_2 \\
\text{ldsz}., \text{ldtype}, \text{ldhint} \quad r_1 = [r_3], \text{imm}_9 \\
\text{ld16}, \text{ldhint} \quad r_1, \text{ar.csd} = [r_3] \\
\text{ld16}, \text{acq}, \text{ldhint} \quad r_1, \text{ar.csd} = [r_3] \\
\text{ld8}, \text{fill}, \text{ldhint} \quad r_1 = [r_3] \\
\text{ld8}, \text{fill}, \text{ldhint} \quad r_1 = [r_3], r_2 \\
\text{ld8}, \text{fill}, \text{ldhint} \quad r_1 = [r_3], \text{imm}_9
\]

Description: A value consisting of \(\text{sz}\) bytes is read from memory starting at the address specified by the value in GR \(r_3\). The value is then zero extended and placed in GR \(r_1\). The values of the \(\text{sz}\) completer are given in Table 2-32. The NaT bit corresponding to GR \(r_1\) is cleared, except as described below for speculative loads. The \(\text{ldtype}\) 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 \(r_1\), 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 \(r_3\) is added to either a signed immediate value (\(\text{imm}_9\)) 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. Base register update is not supported for the \text{ld16} instruction.

Table 2-32. sz Completers

<table>
<thead>
<tr>
<th>sz Completer</th>
<th>Bytes Accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 byte</td>
</tr>
<tr>
<td>2</td>
<td>2 bytes</td>
</tr>
<tr>
<td>4</td>
<td>4 bytes</td>
</tr>
<tr>
<td>8</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

Table 2-33. Load Types

<table>
<thead>
<tr>
<th>ldtype Completer</th>
<th>Interpretation</th>
<th>Special Load Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Normal load</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>Speculative load</td>
<td>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.</td>
</tr>
<tr>
<td>a</td>
<td>Advanced load</td>
<td>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.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>( ldhint ) Completer</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Temporal locality, level 1</td>
</tr>
</tbody>
</table>

Table 2-33. Load Types (Continued)

<table>
<thead>
<tr>
<th>( ldtype ) Completer</th>
<th>Interpretation</th>
<th>Special Load Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sa</td>
<td>Speculative Advanced load</td>
<td>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.</td>
</tr>
<tr>
<td>c.nc</td>
<td>Check load – no clear</td>
<td>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).</td>
</tr>
<tr>
<td>c.clr</td>
<td>Check load – clear</td>
<td>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.</td>
</tr>
<tr>
<td>c.clr.acq</td>
<td>Ordered check load – clear</td>
<td>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.</td>
</tr>
<tr>
<td>acq</td>
<td>Ordered load</td>
<td>An ordered load is performed with acquire semantics.</td>
</tr>
<tr>
<td>bias</td>
<td>Biased load</td>
<td>A hint is provided to the implementation to acquire exclusive ownership of the accessed cache line.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>$ldhint$ Completer</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>nt1</td>
<td>No temporal locality, level 1</td>
</tr>
<tr>
<td>nta</td>
<td>No temporal locality, all levels</td>
</tr>
</tbody>
</table>
\textbf{Operation:} \hspace{1em} \text{if} (\text{PR[gp]}) \{ \\
  \text{size} = \text{fill\_form} \ ? \ 8 : (\text{sixteen\_byte\_form} \ ? \ 16 : \text{sz}); \\
  \text{speculative} = (\text{ldtype} == \text{'s'} || \text{ldtype} == \text{'sa'}); \\
  \text{advanced} = (\text{ldtype} == \text{'a'} || \text{ldtype} == \text{'sa'}); \\
  \text{check\_clear} = (\text{ldtype} == \text{'c.clr'} || \text{ldtype} == \text{'c.clr.acq'}); \\
  \text{check\_no\_clear} = (\text{ldtype} == \text{'c.nc'}); \\
  \text{check} = \text{check\_clear} || \text{check\_no\_clear}; \\
  \text{acquire} = (\text{acquire\_form} || \text{ldtype} == \text{'acq'} || \text{ldtype} == \text{'c.clr.acq'}); \\
  \text{otype} = \text{acquire} \ ? \ \text{ACQUIRE} : \text{UNORDERED}; \\
  \text{bias} = (\text{ldtype} == \text{'bias'}) \ ? \ \text{BIAS} : 0; \\
  \text{translate\_address} = 1; \\
  \text{read\_memory} = 1; \\
  \text{itype} = \text{READ}; \\
  \text{if} (\text{speculative}) \ \text{itype} |= \text{SPEC}; \\
  \text{if} (\text{advanced}) \ \text{itype} |= \text{ADVANCE}; \\
  \text{if} (\text{size} == 16) \ \text{itype} |= \text{UNCACHE\_OPT}; \\
  \text{if} (\text{sixteen\_byte\_form} && !\text{instruction\_implemented(}LD16)) \\
    \text{illegal\_operation\_fault();} \\
  \text{if} ((\text{reg\_base\_update\_form} || \text{imm\_base\_update\_form}) && (r_1 == r_3)) \\
    \text{illegal\_operation\_fault();} \\
  \text{check\_target\_register(r_1);} \\
  \text{if} ((\text{reg\_base\_update\_form} || \text{imm\_base\_update\_form}) \\
    \text{check\_target\_register(r_3);} \\
  \text{if} (\text{reg\_base\_update\_form}) \{ \\
    \text{tmp\_r2} = \text{GR} \[r_2\]; \\
    \text{tmp\_r2nat} = \text{GR} \[r_2\].nat; \\
  \} \\
  \text{if} (!\text{speculative} && \text{GR} \[r_3\].nat) \hspace{1em} // \text{fault on NaT address} \\
    \text{register\_nat\_consumption\_fault(itype);} \\
  \text{defer} = \text{speculative} && (\text{GR} \[r_3\].nat || \text{PSR.ed}); // \text{defer exception if spec} \\
  \text{if} (\text{check} && \text{alat\_cmp(GENERAL, r_1)}) \{ \\
    \text{translate\_address} = \text{alat\_translate\_address\_on\_hit(ldtype, GENERAL,} \\
      \text{r_1);} \\
    \text{read\_memory} = \text{alat\_read\_memory\_on\_hit(ldtype, GENERAL,} \\
      \text{r_1);} \\
  \} \hspace{1em} // \text{remove any old alat entry} \\
  \text{if} (!\text{translate\_address}) \{ \\
    \text{alat\_inval\_single\_entry(GENERAL, r_1);} \\
  \} \hspace{1em} \text{else} \{ \\
    \text{if} (!\text{defer}) \{ \\
      \text{paddr} = \text{tlb\_translate} \[\text{GR} \[r_3\], \text{size}, \text{itype}, \text{PSR.cpl}, \text{&mattr,} \\
        \&\text{defer}] ; \\
      \text{spontaneous\_deferral(paddr, \text{size, UM.be, mattr, otype,} \\
        \text{bias} \ | \ \text{ldhint}, \&\text{defer]);} \\
      \text{if} (!\text{defer} \ && \ \text{read\_memory}) \{ \\
        \text{if} (\text{size} == 16) \{ \\
          \text{mem\_read\_pair($val, \&val\_ar, paddr, \text{size, UM.be, mattr,} \\
            \text{otype, \text{ldhint});} \\
        \} \hspace{1em} \text{else} \{ \\
          \} \hspace{1em} \text{else} \{ \\
          \}}} \} \hspace{1em} \text{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 (req_base_update_form) {
    GR[r3] = GR[r3] + tmp_r2;
    GR[r3].nat = GR[r3].nat || tmp_r2nat;
}
if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
    mem_implicit_prefetch(GR[r3], ldhint | bias, itype);
### Interruptions:

<table>
<thead>
<tr>
<th>Data</th>
<th>Register NaT Consumption fault</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data Key Miss fault</td>
</tr>
<tr>
<td></td>
<td>Data Key Permission fault</td>
</tr>
<tr>
<td></td>
<td>Data Access Rights fault</td>
</tr>
<tr>
<td></td>
<td>Data Access Bit fault</td>
</tr>
<tr>
<td></td>
<td>Data Debug fault</td>
</tr>
<tr>
<td></td>
<td>Unaligned Data Reference fault</td>
</tr>
<tr>
<td></td>
<td>Unsupported Data Reference fault</td>
</tr>
</tbody>
</table>
Idf — Floating-point Load

Format:

- \( \text{ldf} \text{sz} \text{ldhint} \ f_1 = [r_3] \) no_base_update_form M9
- \( \text{ldf} \text{sz} \text{ldhint} \ f_1 = [r_3], r_2 \) reg_base_update_form M7
- \( \text{ldf} \text{sz} \text{ldhint} \ f_1 = [r_3], \text{imm}_9 \) imm_base_update_form M8
- \( \text{ldf} \text{sz} \text{ldhint} \ f_1 = [r_3] \) integer_form, no_base_update_form M9
- \( \text{ldf} \text{sz} \text{ldhint} \ f_1 = [r_3], r_2 \) integer_form, reg_base_update_form M7
- \( \text{ldf} \text{sz} \text{ldhint} \ f_1 = [r_3], \text{imm}_9 \) integer_form, imm_base_update_form M8
- \( \text{ldf} \text{fill} \text{ldhint} \ f_1 = [r_3] \) fill_form, no_base_update_form M9
- \( \text{ldf} \text{fill} \text{ldhint} \ f_1 = [r_3], r_2 \) fill_form, reg_base_update_form M7
- \( \text{ldf} \text{fill} \text{ldhint} \ f_1 = [r_3], \text{imm}_9 \) 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 (\( \text{imm}_9 \)) 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

<table>
<thead>
<tr>
<th>( fsz ) Completer</th>
<th>Bytes Accessed</th>
<th>Memory Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>4 bytes</td>
<td>Single precision</td>
</tr>
<tr>
<td>d</td>
<td>8 bytes</td>
<td>Double precision</td>
</tr>
<tr>
<td>e</td>
<td>10 bytes</td>
<td>Extended precision</td>
</tr>
</tbody>
</table>

Table 2-36. FP Load Types

<table>
<thead>
<tr>
<th>( fldtype ) Completer</th>
<th>Interpretation</th>
<th>Special Load Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Normal load</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>Speculative load</td>
<td>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.</td>
</tr>
<tr>
<td>a</td>
<td>Advanced load</td>
<td>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.</td>
</tr>
</tbody>
</table>
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 \( ldh \) modifier specifies the locality of the memory access. The mnemonic values of \( ldh \) 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 \( ldh \). 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.

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

<table>
<thead>
<tr>
<th>( fltype \ Completer )</th>
<th>Interpretation</th>
<th>Special Load Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sa</td>
<td>Speculative Advanced load</td>
<td>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.</td>
</tr>
<tr>
<td>c.nc</td>
<td>Check load – no clear</td>
<td>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).</td>
</tr>
<tr>
<td>c.clr</td>
<td>Check load – clear</td>
<td>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.</td>
</tr>
</tbody>
</table>
Operation:

```c
if (PR[gp]) {
    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);
    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);
    } else {
        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) { // update base register
    GR[r3] = GR[r3] + sign_ext(imm9, 9);
    GR[r3].nat = GR[r3].nat;
} else if (req_base_update_form) {
    GR[r3] = GR[r3] + GR[r2];
    GR[r3].nat = GR[r3].nat || GR[r2].nat;
}

if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
    mem_implicit_prefetch(GR[r3], ldhint, itype);

fp_update_psr(f1);

**Interruptions:**
- Illegal Operation fault
- Disabled Floating-point Register fault
- Register NaT Consumption fault
- Unimplemented Data Address fault
- Data Nested TLB fault
- Alternate Data TLB fault
- VHPT Data fault
- Data TLB fault
- Data Page Not Present fault

- Data NaT Page Consumption fault
- Data Key Miss fault
- Data Key Permission fault
- Data Access Rights fault
- Data Access Bit fault
- Data Debug fault
- Unaligned Data Reference fault
- Unsupported Data Reference fault
ldfp — Floating-point Load Pair

Format:

\[
\begin{align*}
\text{(q) ldfs}.\text{ldtype}.\text{ldhint} & \; f_1, f_2 = [r_3] & \text{single_form, no_base_update_form} & \text{M11} \\
\text{(q) ldfs}.\text{ldtype}.\text{ldhint} & \; f_1, f_2 = [r_3], 8 & \text{single_form, base_update_form} & \text{M12} \\
\text{(q) ldpd}.\text{ldtype}.\text{ldhint} & \; f_1, f_2 = [r_3] & \text{double_form, no_base_update_form} & \text{M11} \\
\text{(q) ldpd}.\text{ldtype}.\text{ldhint} & \; f_1, f_2 = [r_3], 16 & \text{double_form, base_update_form} & \text{M12} \\
\text{(q) ldfp8}.\text{ldtype}.\text{ldhint} & \; f_1, f_2 = [r_3] & \text{integer_form, no_base_update_form} & \text{M11} \\
\text{(q) ldfp8}.\text{ldtype}.\text{ldhint} & \; f_1, f_2 = [r_3], 16 & \text{integer_form, base_update_form} & \text{M12}
\end{align*}
\]

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 \( \text{ldtype} \) 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 \( \text{ldhint} \) modifier specifies the locality of the memory access. The mnemonic values of \( \text{ldhint} \) 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 \( \text{ldhint} \). 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:

```c
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 (!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 && 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[f_2] = (integer_form ? FP_INT_ZERO : FP_ZERO);
} else {  // execute load normally
    FR[f_1] = fp_mem_to_fr_format(f1_val, size/2, integer_form);
    FR[f_2] = 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[r_3] = GR[r_3] + size;
    GR[r_3].nat = GR[r_3].nat;
    if (!GR[r_3].nat)
        mem_implicit_prefetch(GR[r_3], ldhint, itype);
}

fp_update_psr(f1);
fp_update_psr(f2);

**Interruptions:**
- Illegal Operation fault
- Disabled Floating-point Register fault
- Register NaT Consumption fault
- Unimplemented Data Address fault
- Data Nested TLB fault
- Alternate Data TLB fault
- VHPT Data fault
- Data TLB fault
- Data Page Not Present fault
- Data NaT Page Consumption fault
- Data Key Miss fault
- Data Key Permission fault
- Data Access Rights fault
- Data Access Bit fault
- Data Debug fault
- Unaligned Data Reference fault
Ifetch — Line Prefetch

Format:

\[(qp)\text{ lfetch.lftype.lfhint } [r_3]\]

\[(qp)\text{ lfetch.lftype.lfhint } [r_3], r_2\]

\[(qp)\text{ lfetch.lftype.lfhint } [r_3], \text{imm}_9\]

\[(qp)\text{ lfetch.lftype.excl.lfhint } [r_3]\]

\[(qp)\text{ lfetch.lftype.excl.lfhint } [r_3], r_2\]

\[(qp)\text{ lfetch.lftype.excl.lfhint } [r_3], \text{imm}_9\]

Description: The line containing the address specified by the value in GR \(r_3\) is moved to the highest level of the data memory hierarchy. The value of the \(lh\) 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 \(lh\) 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

<table>
<thead>
<tr>
<th>(lftype) Mnemonic</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>No faults are raised</td>
</tr>
<tr>
<td>fault</td>
<td>Raise faults</td>
</tr>
</tbody>
</table>

In the base update forms, after being used to address memory, the value in GR \(r_3\) is incremented by either the sign-extended value in \(\text{imm}_9\) (in the \text{imm_base_update_form}) or the value in GR \(r_2\) (in the \text{reg_base_update_form}). In the \text{reg_base_update_form}, if the NaT bit corresponding to GR \(r_2\) is set, then the NaT bit corresponding to GR \(r_3\) is set – no fault is raised.

In the \text{reg_base_update_form} and the \text{imm_base_update_form}, if the NaT bit corresponding to GR \(r_3\) is clear, then the address specified by the value in GR \(r_3\) after the post-increment acts as a hint to implicitly prefetch the indicated cache line. This implicit prefetch uses the locality hints specified by \(lh\). The implicit prefetch does not affect program functionality, does not raise any faults, and may be ignored by the implementation.

In the no_base_update_form, the value in GR \(r_3\) is not modified and no implicit prefetch hint is implied.

If the NaT bit corresponding to GR \(r_3\) is set then the state of memory is not affected. In the \text{reg_base_update_form} and \text{imm_base_update_form}, the post increment of GR \(r_3\) is performed and prefetch is hinted as described above.

Ifetch instructions, like hardware prefetches, are not orderable operations, i.e., they have no order with respect to prior or subsequent memory operations.
A faulting \texttt{lfetch} to an unimplemented address results in an Unimplemented Data Address fault. A non-faulting \texttt{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 \texttt{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 \texttt{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 \texttt{rfi} to go back to the process, which will allow the \texttt{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 \texttt{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 \texttt{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 \texttt{lfetch.fault}, but rather to the base update portion, Illegal Operation faults on \texttt{lfetch.fault} should be handled the same as for any other instruction.

**Table 2-38. \texttt{lfhint} Mnemonic Values**

<table>
<thead>
<tr>
<th>\texttt{lfhint} Mnemonic</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Temporal locality, level 1</td>
</tr>
<tr>
<td>nt1</td>
<td>No temporal locality, level 1</td>
</tr>
<tr>
<td>nt2</td>
<td>No temporal locality, level 2</td>
</tr>
<tr>
<td>nta</td>
<td>No temporal locality, all levels</td>
</tr>
</tbody>
</table>

Volume 3: Instruction Reference 3:165
Ifetch

Operation:

```c
if (PR[gp]) {
    itype = READ|NON_ACCESS;
    itype |= (lftype == 'fault') ? LFETCH_FAULT : LFETCH;

    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);

    if (lftype == 'fault') { // faulting form
        if (GR[r3].nat && !PSR.ed) // fault on NaT address
            register_nat_consumption_fault(itype);
    }

    excl_hint = (exclusive_form) ? EXCLUSIVE : 0;

    if (!GR[r3].nat && !PSR.ed) {// faulting form already faulted if r3 is nat
        paddr = tlb_translate(GR[r3], 1, itype, PSR.cpl, &mattr, &defer);
        if (!defer)
            mem_promote(paddr, mattr, lfhint | excl_hint);
    }

    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = GR[r3].nat;
    } else if (reg_base_update_form) {
        GR[r3] = GR[r3] + GR[r2];
        GR[r3].nat = GR[r2].nat || GR[r3].nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
        mem_implicit_prefetch(GR[r3], lfhint | excl_hint, itype);
}
```

Interruptions:

<table>
<thead>
<tr>
<th>Illegal Operation fault</th>
<th>Data Page Not Present fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register NaT Consumption fault</td>
<td>Data NaT Page Consumption fault</td>
</tr>
<tr>
<td>Unimplemented Data Address fault</td>
<td>Data Key Miss fault</td>
</tr>
<tr>
<td>Data Nested TLB fault</td>
<td>Data Key Permission fault</td>
</tr>
<tr>
<td>Alternate Data TLB fault</td>
<td>Data Access Rights fault</td>
</tr>
<tr>
<td>VHPT Data fault</td>
<td>Data Access Bit fault</td>
</tr>
<tr>
<td>Data TLB fault</td>
<td>Data Debug fault</td>
</tr>
</tbody>
</table>
loadrs — Load Register Stack

Format:  loadrs

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:

```c
if (AR[RSC].mode != 0)
    illegal_operation_fault();

if ((CFM.sof != 0) && (AR[RSC].loadrs != 0))
    illegal_operation_fault();

rse_ensure_regs_loaded(AR[RSC].loadrs); // can raise faults listed below
AR[RNAT] = undefined();
```

Interruptions:

- Illegal Operation fault
- Unimplemented Data Address fault
- Data Nested TLB fault
- Alternate Data TLB fault
- VHPT Data fault
- Data TLB fault
- Data Page Not Present fault
- Data NaT Page Consumption fault
- Data Key Miss fault
- Data Key Permission fault
- Data Access Rights fault
- Data Access Bit fault
- Data Debug fault
mf — Memory Fence

Format:  
(qp) mf  
(qp) mf.a  
(ordering_form M24  
(acceptance_form M24

Description:  
This instruction forces ordering between prior and subsequent memory accesses. The ordering_form ensures all prior data memory accesses are made visible prior to any subsequent data memory accesses being made visible. It does not ensure prior data memory references have been accepted by the external platform, nor that prior data memory references are visible.

The acceptance_form prevents any subsequent data memory accesses by the processor from initiating transactions to the external platform until:

• all prior loads to sequential pages have returned data, and
• all prior stores to sequential pages have been accepted by the external platform.

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

Operation:  
if (PR[qp]){
  if (acceptance_form)
    acceptance_fence();
  else // ordering_form
    ordering_fence();
}

Interruptions:  None
mix — Mix

Format:

- \((qp)\) mix1.l \(r_1 = r_2, r_3\)
- \((qp)\) mix2.l \(r_1 = r_2, r_3\)
- \((qp)\) mix4.l \(r_1 = r_2, r_3\)
- \((qp)\) mix1.r \(r_2 = r_1, r_3\)
- \((qp)\) mix2.r \(r_2 = r_1, r_3\)
- \((qp)\) mix4.r \(r_2 = r_1, r_3\)

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:

```c
if (PR[gp]) {
    check_target_register(rj);
    if (one_byte_form) { // one-byte elements
        x[0] = GR[r2]{7:0}; y[0] = GR[r3]{7:0};
        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};
        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};
        if (left_form)
            GR[r1] = concatenate2(x[1], y[1]);
        else // right_form
            GR[r1] = concatenate2(x[0], y[0]);
    }
    GR[rj].nat = GR[rj].nat || GR[rj].nat;
}
```

Interruptions: Illegal Operation fault
mov ar

mov — Move Application Register

Format:

\[(qp)\text{ mov }r_1 = ar_3\] pseudo-op
\[(qp)\text{ mov }ar_3 = r_2\] pseudo-op
\[(qp)\text{ mov }ar_3 = \text{imm}_8\] pseudo-op
\[(qp)\text{ mov.i }r_1 = ar_3\] i_form, from_form I28
\[(qp)\text{ mov.i }ar_3 = r_2\] i_form, register_form, to_form I26
\[(qp)\text{ mov.i }ar_3 = \text{imm}_8\] i_form, immediate_form, to_form I27
\[(qp)\text{ mov.m }r_1 = ar_3\] m_form, from_form M31
\[(qp)\text{ mov.m }ar_3 = r_2\] m_form, register_form, to_form M29
\[(qp)\text{ mov.m }ar_3 = \text{imm}_8\] 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 \(\text{imm}_8\) (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 \text{ld.fill} and mov to UNAT).
Operation:
if (PR[gp]) {
    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
                Privileged Register fault
                Register NaT Consumption fault
                Virtualization fault
                Reserved Register/Field fault
**mov br**

### mov — Move Branch Register

**Format:**

(qp) `mov r₁ = b₂`  
(qp) `mov b₁ = r₂`  
(qp) `mov.mwh.ih b₁ = r₂, tag₁₃`  
(qp) `mov.ret.mwh.ih b₁ = r₂, tag₁₃`

**Description:**

The source operand is copied to the destination register.

In the from form, the branch register specified by `b₂` is copied into GR `r₁`. The NaT bit corresponding to GR `r₁` is cleared.

In the to form, the value in GR `r₂` is copied into BR `b₁`. If the NaT bit corresponding to GR `r₂` 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 `b₁`. 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**

<table>
<thead>
<tr>
<th><code>mwh</code> Completer</th>
<th>Move to BR Whether Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Ignore all hints</td>
</tr>
<tr>
<td>splk</td>
<td>Static Taken</td>
</tr>
<tr>
<td>dptk</td>
<td>Dynamic</td>
</tr>
</tbody>
</table>

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 `tag₁₃` and values corresponding to `none` for the hint completers.

**Operation:**

```c
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:

\[(qp)\text{mov} \quad r_1 = cr_3\]  
\[(qp)\text{mov} \quad cr_3 = r_2\]

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

For the from_form, the control register specified by \textit{cr}_3 is read and the value copied into \textit{GR}_r_1.

For the to_form, \textit{GR}_r_2 is read and the value copied into \textit{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:

\[
\text{if } (PR[qp]) \{
\text{if } (\text{is\_reserved\_reg}(CR\_TYPE, \text{cr}_3) \\text{|| to\_form} \&\& \text{is\_read\_only\_reg}(CR\_TYPE, \text{cr}_3) \\text{|| PSR.ic} \&\& \text{is\_interruption\_cr}(\text{cr}_3))
\{ \text{illegal\_operation\_fault();} \}
\}
\]

\[
\text{if } (\text{from\_form}) \quad \text{check\_target\_register}(r_1);
\text{if } (\text{PSR.cpl} \neq 0) \quad \text{privileged\_operation\_fault}(0);
\]

\[
\text{if } (\text{from\_form}) \quad \{ 
\text{if } (\text{PSR.vm} == 1) \quad \text{virtualization\_fault();}
\text{if } (\text{cr}_3 == \text{IVR}) \quad \text{check\_interrupt\_request();}
\text{if } (\text{cr}_3 == \text{ITIR}) \quad \text{GR}[r_1] = \text{impl\_itir\_cwi\_mask}(\text{CR}[\text{ITIR}]);
\text{else} \quad \text{GR}[r_1] = \text{CR}[\text{cr}_3];
\text{GR}[r_1].nat = 0;
\}
\text{else } \quad \{ // \text{to\_form}
\text{if } (\text{GR}[r_2].nat) \quad \text{register\_nat\_consumption\_fault}(0);
\text{if } (\text{is\_reserved\_field}(CR\_TYPE, \text{cr}_3, \text{GR}[r_2])) \quad \text{reserved\_register\_field\_fault();}
\text{if } ((\text{cr}_3 == \text{IFA}) \&\& \text{impl\_check\_mov\_ifa()} \&\& \text{unimplemented\_virtual\_address}(\text{GR}[r_2], \text{PSR.vm})) \quad \text{unimplemented\_data\_address\_fault}(0);
\text{if } (\text{PSR.vm} == 1) \quad \text{virtualization\_fault();}
\text{if } (\text{cr}_3 == \text{EOI}) \quad \text{end\_of\_interrupt();}
\text{tmp\_val = ignored\_field\_mask}(CR\_TYPE, \text{cr}_3, \text{GR}[r_2]);
\text{CR}[\text{cr}_3] = \text{tmp\_val};
\text{if } (\text{cr}_3 == \text{IIPA})
\}
Interruptions:

- Illegal Operation fault
- Reserved Register/Field fault
- Privileged Operation fault
- Unimplemented Data Address fault
- Register NaT Consumption fault
- Virtualization fault

Serialization:

Reads of control registers reflect the results of all prior instruction groups and interruptions.

In general, writes to control registers do not immediately affect subsequent instructions. Software must issue a serialize operation before a dependent instruction uses a modified resource.

Control register writes are not implicitly synchronized with a corresponding control register read and requires data serialization.
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 \(f_1\).

mov gr

mov — Move General Register

Format: \((qp)\) mov \(r_1 = r_3\)  

pseudo-op of: \((qp)\) adds \(r_1 = 0, r_3\)

Description: The value of GR \(r_3\) is copied to GR \(r_1\).

mov — Move Immediate

Format:  \((qp)\) mov \(r_1 = imm_{22}\)  
pseudo-op of:  \((qp)\) addl \(r_1 = imm_{22}, r_0\)

Description:  The immediate value, \(imm_{22}\), is sign extended to 64 bits and placed in GR \(r_1\).

**mov indirect**

### mov — Move Indirect Register

**Format:**

(qp) mov \( r_1 = i\text{reg}[r_3] \)  
(qp) mov \( i\text{reg}[r_3] = r_2 \)

described in Form M43 from_form  
M42 to_form

**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 \( i\text{reg} \) (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 \( i\text{reg} \). GR \( r_2 \) is read and its value copied into the indexed register.

**Table 2-40. Indirect Register File Mnemonics**

<table>
<thead>
<tr>
<th>( i\text{reg} )</th>
<th>Register File</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpuid</td>
<td>Processor Identification Register</td>
</tr>
<tr>
<td>dbr</td>
<td>Data Breakpoint Register</td>
</tr>
<tr>
<td>ibr</td>
<td>Instruction Breakpoint Register</td>
</tr>
<tr>
<td>prk</td>
<td>Protection Key Register</td>
</tr>
<tr>
<td>pmc</td>
<td>Performance Monitor Configuration Register</td>
</tr>
<tr>
<td>pmd</td>
<td>Performance Monitor Data Register</td>
</tr>
<tr>
<td>rr</td>
<td>Region Register</td>
</tr>
</tbody>
</table>

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 \( \text{PSR.it} == 1 \), or
- the data space for an eager RSE reference when \( \text{PSR.rt} == 1 \)

is an undefined operation.

**Operation:**

```c
if (PR[qp]) {
    if (i\text{reg} == \text{PR_TYPE})
        tmp_index = GR[r_3]{63:61};
    else // all other register types
        tmp_index = GR[r_3]{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))
    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_pkr(GR[r2][31:8])) != NOT_FOUND)
      PKR[tmp_slot].v = 0; // clear valid bit of matching key reg
  }
  tmp_val = ignored_field_mask(ireg, tmp_index, GR[r2]);

  switch (ireg) {
    case DBR_TYPE: DBR[tmp_index] = tmp_val; break;
    case IBR_TYPE: IBR[tmp_index] = tmp_val; break;
    case PKR_TYPE: PKR[tmp_index] = tmp_val; break;
    case PMC_TYPE: pmc_write(tmp_index, tmp_val); break;
  }
mov indirect

    case PMD_TYPE:  pmd_write(tmp_index, tmp_val); break;
    case RR_TYPE:   RR[tmp_index]= tmp_val; break;
    }
    }

Interruptions:  Illegal Operation fault
                Reserved Register/Field fault
                Privileged Operation fault
                Virtualization fault
                Register NaT Consumption 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:**

\[(qp) \text{ mov } r_1 = \text{ip}\]

**Description:** The Instruction Pointer (IP) for the bundle containing this instruction is copied into GR \[r_1\].

**Operation:**

\[
\text{if } \{\text{PR}[qp]\} \{
\text{check\_target\_register}(r_1);
\text{GR}[r_1] = \text{IP};
\text{GR}[r_1].\text{nat} = 0;
\}
\]

**Interruptions:** Illegal Operation fault
**mov pr**

---

**mov — Move Predicates**

**Format:**

- \((qp)\) mov \(r_1 = pr\)  
- \((qp)\) mov \(pr = r_2, mask_{17}\)  
- \((qp)\) mov \(pr.rot = imm_{44}\)

**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 the predicates, the source can either be a general register, or an immediate value. In the to_form, the source operand is GR \(r_2\) and only those predicates specified by the immediate value \(mask_{17}\) are written. The value \(mask_{17}\) is encoded in the instruction in an \(imm_{16}\) field such that: \(imm_{16} = mask_{17} >> 1\). Predicate register 0 is always one. The \(mask_{17}\) value is sign extended. The most significant bit of \(mask_{17}\), therefore, is the mask bit for all of the rotating predicates. If there is a deferred exception for GR \(r_2\) (the NaT bit is 1), a Register NaT Consumption fault is taken.

In the to_rotate_form, only the 48 rotating predicates can be written. The source operand is taken from the \(imm_{44}\) operand (which is encoded in the instruction in an \(imm_{28}\) field, such that: \(imm_{28} = imm_{44} >> 16\)). The low 16-bits correspond to the static predicates. The immediate is sign extended to set the top 21 predicates. Bit position i in the source operand is copied to PR i.

This instruction operates as if the predicate rotation base in the Current Frame Marker (CFM.rrb.pr) were zero.

**Operation:**

```c
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(mask_{17}, 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(imm_{44}, 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

**Format:**

- \( (qp) \text{mov } r_1 = \text{psr} \) from_form \( \text{M36} \)
- \( (qp) \text{mov } \text{psr}.l = r_2 \) to_form \( \text{M35} \)

**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:**

```c
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};
    }
}
```

**Interruptions:**
- Illegal Operation fault
- Reserved Register/Field fault
- Privileged Operation fault
- Virtualization fault
- Register NaT Consumption fault

**Serialization:**
Software must issue an instruction or data serialize operation before issuing instructions dependent upon the altered PSR bits. Unlike with the \( \text{rsm} \) instruction, the PSR.i bit is not treated specially when cleared.
mov - Move User Mask

Format:  
\[(qp)\] mov \(r_1 = \text{psr.um}\)  
\[(qp)\] mov \(\text{psr.um} = r_2\)

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

For move from user mask, PSR\{5:0\} is read, zero-extend, and copied into GR \(r_1\).

For move to user mask, PSR\{5:0\} is written by bits \{5:0\} of GR \(r_2\). PSR.up can only be modified if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Writing a non-zero value into any other parts of the PSR results in a Reserved Register/Field fault.

Operation:
\[
\text{if } (\text{PR[qp]}) \{
\text{if } (\text{from_form}) \{
    \text{check_target_register}(r_1);
    \text{GR}[r_1] = \text{zero_ext(PSR[5:0], 6)};
    \text{GR}[r_1].\text{nat} = 0;
\} \text{ else } {\text{// to_form}}
    \text{if } (\text{GR}[r_2].\text{nat})
        \text{register_nat_consumption_fault}(0);
    \text{if } (\text{is_reserved_field(PSR_TYPE, PSR_UM, GR}[r_2]))
        \text{reserved_register_field_fault}();
    \text{PSR}[1:0] = \text{GR}[r_2][1:0];
    \text{if } (\text{PSR.sp == 0}) \text{// unsecured perf monitor}
        \text{PSR}[2] = \text{GR}[r_2][2];
        \text{PSR}[5:3] = \text{GR}[r_2][5:3];
\}
\}
\]

Interruptions: Illegal Operation fault  
Reserved Register/Field fault  
Register NaT Consumption fault

Serialization: All user mask modifications are observed by the next instruction group.
movl — Move Long Immediate

Format: \((qp) \text{ movl } r_1 = \text{imm}_{64}\)

Description: The immediate value \(\text{imm}_{64}\) is copied to GR \(r_1\). The L slot of the bundle contains 41 bits of \(\text{imm}_{64}\).

Operation: \[
\text{if (PR[qp]) } \{
    \text{check_target_register}(r_1);
    \text{GR}[r_1] = \text{imm}_{64};
    \text{GR}[r_1].nat = 0;
\}
\]

Interruptions: Illegal Operation fault
mpy4 — Unsigned Integer Multiply

Format: \((qp)\) mpy4  \(r_1 = r_2, r_3\)

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:

\[
\text{if (PR[qp])} \{
\text{if (!instruction_implemented(mpy4))}
\text{illegal_operation_fault();}
\text{check_target_register(r_1);}
\text{GR[r_1] = zero_ext(GR[r_2], 32) * zero_ext(GR[r_3], 32);}
\text{GR[r_2].nat = GR[r_2].nat || GR[r_3].nat;}
\}
\]

Interruptions: Illegal Operation fault
mpyshl4 — Unsigned Integer Shift Left and Multiply

Format: \((qp)\) mpyshl4 \(r_1 = r_2, r_3\)

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 \((r_c = r_a \times r_b)\):

\[
\begin{align*}
\text{mpy4} & \quad r_1 = r_a, r_b; \quad //\text{partial product low 32 bits} \times \text{low 32 bits} \\
\text{mpyshl4} & \quad r_2 = r_a, r_b; \quad //\text{partial product high 32 bits} \times \text{low 32 bits} \\
\text{mpyshl4} & \quad r_3 = r_b, r_a; \quad //\text{partial product low 32 bits} \times \text{high 32 bits} \\
\text{add} & \quad r_1 = r_1, r_2; \quad //\text{partial sum} \\
\text{add} & \quad r_c = r_1, r_3 \quad //\text{final sum}
\end{align*}
\]

Operation: if \((\text{PR[qp]})\) {
    if (!\text{instruction_implemented(MPYSHL4)})
        \text{illegal_operation_fault();}
    \text{check_target_register}(r_1); \\
    \text{GR}[r_1] = (\text{zero_ext(}(\text{GR}[r_2] >> 32), 32) \times \text{zero_ext(}\text{GR}[r_3], 32) \ll 32; \\
    \text{GR}[r_1].\text{nat} = \text{GR}[r_2].\text{nat} | \text{GR}[r_3].\text{nat};
}

Interruptions: Illegal Operation fault
**mux — Mux**

**Format:**

\[
(qp) \text{mux1} \ r_1 = r_2, \text{mbtype}_4
\]

\[
(qp) \text{mux2} \ r_1 = r_2, \text{mbtype}_8
\]

**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**

<table>
<thead>
<tr>
<th>mbtype(_4)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>@rev</td>
<td>Reverse the order of the bytes</td>
</tr>
<tr>
<td>@mix</td>
<td>Perform a Mix operation on the two halves of GR ( r_2 )</td>
</tr>
<tr>
<td>@shuf</td>
<td>Perform a Shuffle operation on the two halves of GR ( r_2 )</td>
</tr>
<tr>
<td>@alt</td>
<td>Perform an Alternate operation on the two halves of GR ( r_2 )</td>
</tr>
<tr>
<td>@brcst</td>
<td>Perform a Broadcast operation on the least significand byte of GR ( r_2 )</td>
</tr>
</tbody>
</table>

**Figure 2-26. Mux1 Operation (8-bit elements)**

- mux1 \( r_1 = r_2, \text{rev} \)
- mux1 \( r_1 = r_2, \text{mix} \)
- mux1 \( r_1 = r_2, \text{shuf} \)
- mux1 \( r_1 = r_2, \text{alt} \)
- mux1 \( r_1 = r_2, \text{brcst} \)
For 16-bit elements, all possible permutations, with and without repetitions can be specified. They are expressed with an 8-bit \textit{mhtype}_8 field, which encodes the indices of the four 16-bit data elements. The indexed 16-bit elements of GR \textit{r}_2 are copied to corresponding 16-bit positions in the target register GR \textit{r}_1. The indices are encoded in little-endian order. (The 8 bits of \textit{mhtype}_8[7:0] are grouped in pairs of bits and named \textit{mhtype}_8[3], \textit{mhtype}_8[2], \textit{mhtype}_8[1], \textit{mhtype}_8[0] in the Operation section).

\textbf{Figure 2-27. Mux2 Examples (16-bit elements)}

\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
GR \textit{r}_1: & GR \textit{r}_2: \\
\hline
\multicolumn{2}{|c|}{mux2 \textit{r}_1 = \textit{r}_2, 0x8d (shuffle 10 00 11 01)} \\
\hline
GR \textit{r}_1: & GR \textit{r}_2: \\
\hline
\multicolumn{2}{|c|}{mux2 \textit{r}_1 = \textit{r}_2, 0x1b (reverse 00 01 10 11)} \\
\hline
GR \textit{r}_1: & GR \textit{r}_2: \\
\hline
\multicolumn{2}{|c|}{mux2 \textit{r}_1 = \textit{r}_2, 0xaa (broadcast 10 10 10 10)} \\
\hline
GR \textit{r}_1: & GR \textit{r}_2: \\
\hline
\multicolumn{2}{|c|}{mux2 \textit{r}_1 = \textit{r}_2, 0xd8 (alternate 11 01 10 00)} \\
\hline
\end{tabular}
\end{figure}
mux

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[3] = GR[r2]{31:24};
    x[5] = GR[r2]{47:40};
    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**

**Format:**

- (qp) nop $imm_{21}$
- (qp) nop.i $imm_{21}$
- (qp) nop.b $imm_{21}$
- (qp) nop.m $imm_{21}$
- (qp) nop.f $imm_{21}$
- (qp) nop.x $imm_{62}$

- pseudo-op
- i_unit_form I18
- b_unit_form B9
- m_unit_form M48
- f_unit_form F16
- x_unit_form X5

**Description:**

No operation is done. The immediate, $imm_{21}$ or $imm_{62}$, can be used by software as a marker in program code. It is ignored by hardware.

For the x_unit_form, the L slot of the bundle contains the upper 41 bits of $imm_{62}$.

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

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

**Operation:**

```c
if (PR[qp]) {
    ; // no operation
}
```

**Interruptions:** None
or — Logical Or

Format: \( (qp) \text{ or } r_1 = r_2, r_3 \)  
\( (qp) \text{ or } r_1 = \text{imm}_8, r_3 \)

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 \( \text{imm}_8 \) encoding field.

Operation:

\[
\text{if} \ (\text{PR}[qp]) \ {\}
\quad \text{check_target_register}(r_1);
\quad \text{tmp}_\text{src} = \{\text{register_form} \ ? \ \text{GR}[r_2] : \text{sign_ext}(\text{imm}_8, 8)\};
\quad \text{tmp}_\text{nat} = \{\text{register_form} \ ? \ \text{GR}[r_2].\text{nat} : 0\};
\quad \text{GR}[r_1] = \text{tmp}_\text{src} | \text{GR}[r_3];
\quad \text{GR}[r_1].\text{nat} = \text{tmp}_\text{nat} || \text{GR}[r_3].\text{nat};
\]

Interruptions: Illegal Operation fault
**pack — Pack**

**Format:**
- $(qp)$ pack2.sss $r_1 = r_2, r_3$
- $(qp)$ pack2.us $r_1 = r_2, r_3$
- $(qp)$ pack4.sss $r_1 = r_2, r_3$

**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**

<table>
<thead>
<tr>
<th>Size</th>
<th>Source Element Width</th>
<th>Result Element Width</th>
<th>Saturation</th>
<th>Upper Limit</th>
<th>Lower Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>16 bit</td>
<td>8 bit</td>
<td>signed</td>
<td>0x7f</td>
<td>0x80</td>
</tr>
<tr>
<td>2</td>
<td>16 bit</td>
<td>8 bit</td>
<td>unsigned</td>
<td>0xff</td>
<td>0x00</td>
</tr>
<tr>
<td>4</td>
<td>32 bit</td>
<td>16 bit</td>
<td>signed</td>
<td>0x7fff</td>
<td>0x8000</td>
</tr>
</tbody>
</table>

**Figure 2-28. Pack Operation**
pack

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

    if (two_byte_form) {
        if (signed_saturation_form) {
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);
        } else { // unsigned_saturation_form
            max = 0xff;
            min = 0x00;
        }
        temp[0] = sign_ext(GR[r2]{15:0}, 16);
        temp[1] = sign_ext(GR[r2]{31:16}, 16);
        temp[2] = sign_ext(GR[r2]{47:32}, 16);
        temp[3] = sign_ext(GR[r2]{63:48}, 16);
        temp[4] = sign_ext(GR[r3]{15:0}, 16);
        temp[5] = sign_ext(GR[r3]{31:16}, 16);
        temp[6] = sign_ext(GR[r3]{47:32}, 16);
        temp[7] = sign_ext(GR[r3]{63:48}, 16);

        for (i = 0; i < 8; i++) {
            if (temp[i] > max)
                temp[i] = max;
            if (temp[i] < min)
                temp[i] = min;
        }

        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                               temp[3], temp[2], temp[1], temp[0]);
    } else { // four_byte_form
        max = sign_ext(0x7fff, 16); // signed_saturation_form
        min = sign_ext(0x8000, 16);
        temp[0] = sign_ext(GR[r2]{31:0}, 32);
        temp[1] = sign_ext(GR[r2]{63:32}, 32);
        temp[2] = sign_ext(GR[r3]{31:0}, 32);
        temp[3] = sign_ext(GR[r3]{63:32}, 32);

        for (i = 0; i < 4; i++) {
            if (temp[i] > max)
                temp[i] = max;
            if (temp[i] < min)
                temp[i] = min;
        }

        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}

Interruptions: Illegal Operation fault
padd — Parallel Add

Format:

\[(qp)\] padd1 \( r_1 = r_2, r_3 \) one_byte_form, modulo_form A9
\[(qp)\] padd1.sss \( r_1 = r_2, r_3 \) one_byte_form, sss_saturation_form A9
\[(qp)\] padd1.uus \( r_1 = r_2, r_3 \) one_byte_form, uus_saturation_form A9
\[(qp)\] padd1.uuu \( r_1 = r_2, r_3 \) one_byte_form, uuu_saturation_form A9
\[(qp)\] padd2 \( r_1 = r_2, r_3 \) two_byte_form, modulo_form A9
\[(qp)\] padd2.sss \( r_1 = r_2, r_3 \) two_byte_form, sss_saturation_form A9
\[(qp)\] padd2.uus \( r_1 = r_2, r_3 \) two_byte_form, uus_saturation_form A9
\[(qp)\] padd2.uuu \( r_1 = r_2, r_3 \) two_byte_form, uuu_saturation_form A9
\[(qp)\] padd4 \( r_1 = r_2, r_3 \) four_byte_form, modulo_form A9

Description: The sets of elements from the two source operands are added, and the results placed in GR \( r_1 \).

If a sum of two elements cannot be represented in the result element and a saturation completer is specified, then saturation clipping is performed. The saturation can either be signed or unsigned, as given in Table 2-43. 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

<table>
<thead>
<tr>
<th>Completer</th>
<th>Result ( r_1 ) treated as</th>
<th>Source ( r_2 ) treated as</th>
<th>Source ( r_3 ) treated as</th>
</tr>
</thead>
<tbody>
<tr>
<td>sss</td>
<td>signed</td>
<td>signed</td>
<td>signed</td>
</tr>
<tr>
<td>uus</td>
<td>unsigned</td>
<td>unsigned</td>
<td>signed</td>
</tr>
<tr>
<td>uuu</td>
<td>unsigned</td>
<td>unsigned</td>
<td>unsigned</td>
</tr>
</tbody>
</table>

Table 2-44. Parallel Add Saturation Limits

<table>
<thead>
<tr>
<th>Size</th>
<th>Element Width</th>
<th>Result ( r_1 ) Signed</th>
<th>Result ( r_1 ) Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Upper Limit</td>
<td>Lower Limit</td>
</tr>
<tr>
<td>1</td>
<td>8 bit</td>
<td>0x7f</td>
<td>0x80</td>
</tr>
<tr>
<td>2</td>
<td>16 bit</td>
<td>0x7fff</td>
<td>0x8000</td>
</tr>
</tbody>
</table>

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};
        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};
    }
}
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++) {
        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**

**Format:**

- (qp) pavg1 \( r_1 = r_2, r_3 \)
- (qp) pavg1.raz \( r_1 = r_2, r_3 \)
- (qp) pavg2 \( r_1 = r_2, r_3 \)
- (qp) pavg2.raz \( r_1 = r_2, r_3 \)

**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:

```c
if (PR[gp]) {
    check_target_register(rj);
    if (one_byte_form) {
        x[0] = GR[r2]{7:0};  y[0] = GR[r3]{7:0};
        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};
        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
**pavgsub — Parallel Average Subtract**

**Format:**

- `(qp) pavgsub1 r1 = r2, r3`  
  - one_byte_form  
  - A9
- `(qp) pavgsub2 r1 = r2, r3`  
  - 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[gp]) {
    check_target_register(rj);
}

if (one_byte_form) {
    x[0] = GR[r2]{7:0};  y[0] = GR[r3]{7:0};
}

    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[rj] = 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};

    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[rj] = concatenate4(res[3], res[2], res[1], res[0]);
}

    GR[rj].nat = GR[r2].nat || GR[r3].nat;
}

Interruptions:  Illegal Operation fault
**pcmp — Parallel Compare**

**Format:**

(\(qp\)) \(\text{pcmp1.prel } r_1 = r_2, r_3\)  
(\(qp\)) \(\text{pcmp2.prel } r_1 = r_2, r_3\)  
(\(qp\)) \(\text{pcmp4.prel } r_1 = r_2, r_3\)

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**

<table>
<thead>
<tr>
<th>prel</th>
<th>Compare Relation ((r_2 \text{ prel } r_3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>(r_2 == r_3)</td>
</tr>
<tr>
<td>gt</td>
<td>(r_2 &gt; r_3) (signed)</td>
</tr>
</tbody>
</table>

**Figure 2-33. Parallel Compare Examples**

```plaintext
GR r3:
GR r2:
GR r1:

pcmp1.gt

GR r3:
GR r2:
GR r1:

pcmp2.eq

GR r3:
GR r2:
GR r1:

pcmp4.eq
```
Operation:
if (PR[gp]) {
    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[5] = GR[r2][47:40]; y[5] = GR[r3][47:40];
        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];
        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];
        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]);
    }
}
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:**

\[
(qp) \text{pmax1.u } r_1 = r_2, r_3 \\
(qp) \text{pmax2 } r_1 = r_2, r_3
\]

**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**

![Parallel Maximum Examples Diagram](image-url)
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};
        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};
        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) \( pmin1.u \) \( r_1 = r_2, r_3 \)
- (qp) \( pmin2 \) \( r_1 = r_2, r_3 \)

**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**

![Parallel Minimum Examples Diagram](image-url)
Operation:  
  if (PR[gp]) {
    check_target_register(r1);

    if (one_byte_form) {  // one-byte elements
      x[0] = GR[r2]{7:0};  y[0] = GR[r3]{7:0};
      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};
      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:

(\text{qp}) \text{pmpy2.r} \ r_1 = r_2, r_3 \quad \text{right_form} \ l2
(\text{qp}) \text{pmpy2.l} \ r_1 = r_2, r_3 \quad \text{left_form} \ l2

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

\text{GR}[r_1].\text{nat} = \text{GR}[r_2].\text{nat} \ | | \ \text{GR}[r_3].\text{nat};

Interruptions: Illegal Operation fault
pmpyshr — Parallel Multiply and Shift Right

Format: $(qp)$ pmpyshr2 $r_1 = r_2, r_3, count_2$
$(qp)$ pmpyshr2.u $r_1 = r_2, r_3, count_2$

signed_form $|1$
unsigned_form $|1$

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

<table>
<thead>
<tr>
<th>$count_2$</th>
<th>Selected Bit Field from Each 32-bit Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15:0</td>
</tr>
<tr>
<td>7</td>
<td>22:7</td>
</tr>
<tr>
<td>15</td>
<td>30:15</td>
</tr>
<tr>
<td>16</td>
<td>31:16</td>
</tr>
</tbody>
</table>

Figure 2-37. Parallel Multiply and Shift Right Operation
Operation:

```c
if (PR[gp]) {
    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];
    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:**

```c
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

Format:

\[
\begin{align*}
(qp) \text{ probe.r } r_1 &= r_3, r_2 \\
(qp) \text{ probe.w } r_1 &= r_3, r_2 \\
(qp) \text{ probe.r } r_1 &= r_3, \text{imm}_2 \\
(qp) \text{ probe.w } r_1 &= r_3, \text{imm}_2 \\
(qp) \text{ probe.r.fault } r_3, \text{imm}_2 \\
(qp) \text{ probe.w.fault } r_3, \text{imm}_2 \\
(qp) \text{ probe.rw.fault } r_3, \text{imm}_2
\end{align*}
\]

regular_form, read_form, register_form M38
regular_form, write_form, register_form M38
regular_form, read_form, immediate_form M39
regular_form, write_form, immediate_form M39
fault_form, read_form, immediate_form M40
fault_form, write_form, immediate_form M40
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 \( \text{imm}_2 \). 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:
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.

### Table 2-47. Faults for regular_form and fault_form Probe Instructions

<table>
<thead>
<tr>
<th>Probe Form Type</th>
<th>Faults</th>
</tr>
</thead>
</table>
| regular_form    | Register NaT Consumption fault  
Virtualization 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 |
| fault_form      | Register NaT Consumption fault  
Unimplemented Data Address fault  
Virtualization fault filtrates 
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.
Operation: if (PR[gp]) {
nitype = NON_ACCESS;
nitype |= (read_write_form) ? READ|WRITE : ((write_form) ? WRITE : READ);
nitype |= (fault_form) ? PROBE_FAULT : PROBE;
nitype |= (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(nitype);

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

Interruptions: Illegal Operation fault
            Register NaT Consumption fault
            Unimplemented Data Address fault
            Virtualization fault
            Data Nested TLB fault
            Alternate Data TLB fault
            VHPT Data fault
            Data TLB fault
            Data Page Not Present fault
            Data NaT Page Consumption fault
            Data Key Miss fault
            Data Key Permission fault
            Data Access Rights fault
            Data Dirty Bit fault
            Data Access Bit fault
            Data Debug fault
**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[gp]) {
    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[5] = GR[r2][47:40]; y[5] = GR[r3][47:40];
    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

Format:

\[
\begin{align*}
(qp) \text{ pshl2 } r_1 &= r_2, r_3, \\
(qp) \text{ pshl2 } r_1 &= r_2, \text{ count}_5, \\
(qp) \text{ pshl4 } r_1 &= r_2, r_3, \\
(qp) \text{ pshl4 } r_1 &= r_2, \text{ count}_5,
\end{align*}
\]

two_byte_form, variable_form \quad I7

two_byte_form, fixed_form \quad I8

four_byte_form, variable_form \quad I7

four_byte_form, fixed_form \quad I8

Description: The data elements of GR \( r_2 \) are each independently shifted to the left by the scalar shift count in \( r_3 \), or in the immediate field \( \text{count}_5 \). 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:

\[
\begin{align*}
\text{if } (\text{PR}[qp]) \{ \\
\text{check_target_register}(r_1); \\
\text{shift_count} = (\text{variable_form} ? \text{GR}[r_3] : \text{count}_5); \\
\text{tmp_nat} = (\text{variable_form} ? \text{GR}[r_3].\text{nat} : 0); \\
\text{if } (\text{two_byte_form}) \{ \\
\quad \text{if } (\text{shift_count} > 16) \\
\quad \quad \text{shift_count} = 16; \\
\quad \text{GR}[r_1][15:0] = \text{GR}[r_2][15:0] \ll \text{shift_count}; \\
\quad \text{GR}[r_1][31:16] = \text{GR}[r_2][31:16] \ll \text{shift_count}; \\
\quad \text{GR}[r_1][47:32] = \text{GR}[r_2][47:32] \ll \text{shift_count}; \\
\quad \text{GR}[r_1][63:48] = \text{GR}[r_2][63:48] \ll \text{shift_count}; \\
\} \quad \text{else } \{ \\
\quad \text{if } (\text{shift_count} > 32) \\
\quad \quad \text{shift_count} = 32; \\
\quad \text{GR}[r_1][31:0] = \text{GR}[r_2][31:0] \ll \text{shift_count}; \\
\quad \text{GR}[r_1][63:32] = \text{GR}[r_2][63:32] \ll \text{shift_count}; \\
\} \\
\quad \text{GR}[r_1].\text{nat} = \text{GR}[r_2].\text{nat} \| \text{tmp_nat}; \\
\}
\end{align*}
\]

Interruptions: Illegal Operation fault
pshladd — Parallel Shift Left and Add

Format:  \((qp) \text{ pshladd} r_1 = r_2, \text{ count}_2, r_3\)

Description: The four signed 16-bit data elements of GR \(r_2\) are each independently shifted to the left by \(\text{count}_2\) 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:  
\[
\begin{align*}
\text{if } & (\text{PR}[qp]) \{ \\
& \quad \text{check\_target\_register}(r_1); \\
& \quad x[0] = \text{GR}[r_2][15:0]; \quad y[0] = \text{GR}[r_3][15:0]; \\
& \quad x[1] = \text{GR}[r_2][31:16]; \quad y[1] = \text{GR}[r_3][31:16]; \\
& \quad x[2] = \text{GR}[r_2][47:32]; \quad y[2] = \text{GR}[r_3][47:32]; \\
& \quad x[3] = \text{GR}[r_2][63:48]; \quad y[3] = \text{GR}[r_3][63:48]; \\
& \quad \text{max} = \text{sign\_ext}(0x7fff, 16); \\
& \quad \text{min} = \text{sign\_ext}(0x8000, 16); \\
& \quad \text{for } (i = 0; i < 4; i++) \{ \\
& \quad \quad \text{temp}[i] = \text{sign\_ext}(x[i], 16) \ll \text{count}_2; \\
& \quad \quad \text{if } (\text{temp}[i] > \text{max}) \\
& \quad \quad \quad \text{res}[i] = \text{max}; \\
& \quad \quad \text{else if } (\text{temp}[i] < \text{min}) \\
& \quad \quad \quad \text{res}[i] = \text{min}; \\
& \quad \quad \text{else } \{ \\
& \quad \quad \quad \text{res}[i] = \text{temp}[i] + \text{sign\_ext}(y[i], 16); \\
& \quad \quad \quad \text{if } (\text{res}[i] > \text{max}) \\
& \quad \quad \quad \quad \text{res}[i] = \text{max}; \\
& \quad \quad \quad \text{if } (\text{res}[i] < \text{min}) \\
& \quad \quad \quad \quad \text{res}[i] = \text{min}; \\
& \quad \quad \}\}
\end{align*}
\]

\[
\text{GR}[r_1] = \text{concatenate4}(\text{res}[3], \text{res}[2], \text{res}[1], \text{res}[0]); \\
\text{GR}[r_1].\text{nat} = \text{GR}[r_2].\text{nat} \lor \text{GR}[r_3].\text{nat};
\]

Interruptions: Illegal Operation fault
pshr — Parallel Shift Right

Format:

- (qp) pshr2  \( r_1 = r_3, r_2 \)
- (qp) pshr2  \( r_1 = r_3, \text{count}_5 \)
- (qp) pshr2.u  \( r_1 = r_3, r_2 \)
- (qp) pshr2.u  \( r_1 = r_3, \text{count}_5 \)
- (qp) pshr4  \( r_1 = r_3, r_2 \)
- (qp) pshr4  \( r_1 = r_3, \text{count}_5 \)
- (qp) pshr4.u  \( r_1 = r_3, r_2 \)
- (qp) pshr4.u  \( r_1 = r_3, \text{count}_5 \)

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 \( \text{count}_5 \). 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[gp]) {
    check_target_register(rj);

    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[rj]{15:0} = shift_right_unsigned(zero_ext(GR[r3]{15:0}, 16),
                        shift_count);
            GR[rj]{31:16} = shift_right_unsigned(zero_ext(GR[r3]{31:16}, 16),
                        shift_count);
            GR[rj]{47:32} = shift_right_unsigned(zero_ext(GR[r3]{47:32}, 16),
                        shift_count);
            GR[rj]{63:48} = shift_right_unsigned(zero_ext(GR[r3]{63:48}, 16),
                        shift_count);
        } else {  // signed shift
            GR[rj]{15:0} = shift_right_signed(sign_ext(GR[r3]{15:0}, 16),
                        shift_count);
            GR[rj]{31:16} = shift_right_signed(sign_ext(GR[r3]{31:16}, 16),
                        shift_count);
            GR[rj]{47:32} = shift_right_signed(sign_ext(GR[r3]{47:32}, 16),
                        shift_count);
            GR[rj]{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[rj]{31:0} = shift_right_unsigned(zero_ext(GR[r3]{31:0}, 32),
                        shift_count);
            GR[rj]{63:32} = shift_right_unsigned(zero_ext(GR[r3]{63:32}, 32),
                        shift_count);
        } else {  // signed shift
            GR[rj]{31:0} = shift_right_signed(sign_ext(GR[r3]{31:0}, 32),
                        shift_count);
            GR[rj]{63:32} = shift_right_signed(sign_ext(GR[r3]{63:32}, 32),
                        shift_count);
        }
    }

    GR[rj].nat = GR[rj].nat || tmp_nat;
}

Interruptions:  Illegal Operation fault
**pshradd — Parallel Shift Right and Add**

**Format:** \((qp)\) pshradd2 \(r_1 = r_2, \text{count}_2, r_3\)

**Description:** The four signed 16-bit data elements of GR \(r_2\) are each independently shifted to the right by \(\text{count}_2\) 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:**

```c
if (PR[qp]) {
    check_target_register(r1);
    x[0] = GR[r2]{15:0}; y[0] = GR[r3]{15:0};
    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

Format:

- \((qp)\) psub1 \(r_1 = r_2, r_3\)
- \((qp)\) psub1.sss \(r_1 = r_2, r_3\)
- \((qp)\) psub1.uus \(r_1 = r_2, r_3\)
- \((qp)\) psub1.uuu \(r_1 = r_2, r_3\)
- \((qp)\) psub2 \(r_1 = r_2, r_3\)
- \((qp)\) psub2.sss \(r_1 = r_2, r_3\)
- \((qp)\) psub2.uus \(r_1 = r_2, r_3\)
- \((qp)\) psub2.uuu \(r_1 = r_2, r_3\)
- \((qp)\) psub4 \(r_1 = r_2, r_3\)

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

<table>
<thead>
<tr>
<th>Completer</th>
<th>Result (r_1) treated as</th>
<th>Source (r_2) treated as</th>
<th>Source (r_3) treated as</th>
</tr>
</thead>
<tbody>
<tr>
<td>sss</td>
<td>signed</td>
<td>signed</td>
<td>signed</td>
</tr>
<tr>
<td>uus</td>
<td>unsigned</td>
<td>unsigned</td>
<td>signed</td>
</tr>
<tr>
<td>uuu</td>
<td>unsigned</td>
<td>unsigned</td>
<td>unsigned</td>
</tr>
</tbody>
</table>

Table 2-49. Parallel Subtract Saturation Limits

<table>
<thead>
<tr>
<th>Size</th>
<th>Element Width</th>
<th>Result (r_1) Signed</th>
<th>Result (r_1) Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Upper Limit</td>
<td>Lower Limit</td>
</tr>
<tr>
<td>1</td>
<td>8 bit</td>
<td>0x7f</td>
<td>0x80</td>
</tr>
<tr>
<td>2</td>
<td>16 bit</td>
<td>0x7ff</td>
<td>0x8000</td>
</tr>
</tbody>
</table>

Figure 2-40. Parallel Subtract Examples
psub

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};
        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);
            }
        }
        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};
        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);
            }
        }
        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
            temp[3], temp[2], temp[1], temp[0]);
    }
}
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 r3

**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 r3 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.

```c
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:**

```c
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat)
        register_nat_consumption_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();
    tlb_purge_translation_cache(GR[r3]);
}
```

**Interruptions:**  
Privileged Operation fault  Virtualization fault  
Register NaT Consumption fault

**Serialization:**  
Software must issue a data serialization operation to ensure the purge is complete before issuing a data access or non-access reference dependent upon the purge. Software must issue instruction serialize operation before fetching an instruction dependent upon the purge.
ptc.g, ptc.ga — Purge Global Translation Cache

Format:  

\[(qp)\ ptc.g\ r_3,\ r_2\]  
\[(qp)\ ptc.ga\ r_3,\ r_2\]  

Description:  
The instruction and data translation cache for each processor in the local TLB coherence domain are searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. These entries are removed.

The purge virtual address is specified by GR \(r_3\) bits\{60:0\} and the purge region identifier is selected by GR \(r_3\) bits \{63:61\}. GR \(r_2\) specifies the address range of the purge as \(1<<GR[r_2]\{7:2\}\) bytes in size. 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:

```markdown
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);
}
```

Interruptions:

- Machine Check abort
- Unimplemented Data Address fault
- Privileged Operation fault
- Virtualization fault
- Register NaT Consumption 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.l — Purge Local Translation Cache**

**Format:**  
\[(qp) \text{ ptc.l } r_3, r_2\]  

**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:**  
\[
\text{if (PR[qp]) } \{
  \text{if (PSR.cpl != 0)}
  \quad \text{privileged_operation_fault(0);} 
  \text{if (GR[\text{r}_3].nat || GR[\text{r}_2].nat)}
  \quad \text{register_nat_consumption_fault(0);} 
  \text{if (unimplemented_virtual_address(GR[\text{r}_3], PSR.vm))}
  \quad \text{unimplemented_data_address_fault(0);} 
  \text{if (PSR.vm == 1)}
  \quad \text{virtualization_fault();}
  \text{tmp_rid = RR[GR[\text{r}_3]\{63:61\}].rid;} 
  \text{tmp_va = GR[\text{r}_3]\{60:0\};} 
  \text{tmp_size = GR[\text{r}_2]\{7:2\};} 
  \text{tmp_va = align_to_size_boundary(tmp_va, tmp_size);} 
  \text{tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);} 
  \text{tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);} 
\} 
\]

**Interruptions:**  
Machine Check abort  
Privileged Operation fault  
Virtualization fault  
Unimplemented Data Address fault  
Register NaT Consumption 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

Format:
- $(qp)$ ptr.d $r_3, r_2$
- $(qp)$ ptr.i $r_3, r_2$

data_form M45
instruction_form M45

Description:

In the data form of this instruction, the data translation registers and caches are searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. All these entries are removed. Entries in the instruction translation registers are unaffected by the data form of the purge.

In the instruction form, the instruction translation registers and caches are searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. All these entries are removed. Entries in the data translation registers are unaffected by the instruction form of the purge.

In addition, in both forms, the instruction and data translation cache may be purged of more translations than specified by the purge parameters up to and including removal of all entries within the translation cache.

The purge virtual address is specified by GR $r_3$ bits $\{60:0\}$ and the purge region identifier is selected by GR $r_3$ bits $\{63:61\}$. GR $r_2$ specifies the address range of the purge as $1<<GR[r_2]{7:2}$ bytes in size. 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:

```c
if (PR[gp]) {
    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

**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))
        unimplemented_address = 1;
    if (CR[IPSR].it)
        unimplemented_virtual_address(tmp_IP, IPSR.vm);
    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
- Unimplemented Instruction Address trap
- 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 imm24

**Description:**  
The complement of the imm24 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 (im24{21}) PSR{21} = 0;   // pp
if (im24{22}) PSR{22} = 0;   // di
if (im24{23}) PSR{23} = 0;   // si
}

**Interruptions:**
- Privileged Operation fault
- Virtualization fault
- Reserved Register/Field fault

**Serialization:**
Software must use a data serialize or instruction serialize operation before issuing instructions dependent upon the altered PSR bits – except the PSR.i bit. The PSR.i bit is implicitly serialized and the processor ensures that external interrupts are masked by the time the next instruction executes.
rum — Reset User Mask

Format:  \((ap)\) rum \(imm_{24}\)  

Description:  The complement of the \(imm_{24}\) operand is ANDed with the user mask (PSR\{5:0\}) and the result is placed in the user mask. 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:  
\[
\text{if (PR[qp])} \ {\}
\text{if (is_reserved_field(PSR\_TYPE, PSR\_UM, imm_{24}))} \\
\text{reserved_register_field_fault();}
\]
\[
\text{if (imm_{24}(1)) PSR(1) = 0;} // \text{be}
\]
\[
\text{if (imm_{24}(2) \& PSR.sp == 0)} //\text{non-secure perf monitor}
\text{PSR(2) = 0;} // \text{up}
\]
\[
\text{if (imm_{24}(3)) PSR(3) = 0;} // \text{ac}
\]
\[
\text{if (imm_{24}(4)) PSR(4) = 0;} // \text{mfl}
\]
\[
\text{if (imm_{24}(5)) PSR(5) = 0;} // \text{mfh}
\]

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**

**Format:**

\[(qp)\text{ setf.s } f_1 = r_2\]

\[(qp)\text{ setf.d } f_1 = r_2\]

\[(qp)\text{ setf.exp } f_1 = r_2\]

\[(qp)\text{ setf.sig } f_1 = r_2\]

**single_form** M18

**double_form** M18

**exponent_form** M18

**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**

![Figure 2-41](image)

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**

![Figure 2-42](image)

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:  

```c
if (PR[gp]) {
    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) shl \( r_1 = r_2, r_3 \)  

(qp) shl \( r_1 = r_2, count_6 \)  
pseudo-op of: (qp) dep.z \( r_1 = r_2, count_6, 64-count_6 \)

**Description:** The value in GR \( r_2 \) is shifted to the left, with the vacated bit positions filled with zeroes, and placed in GR \( r_1 \). The number of bit positions to shift is specified by the value in GR \( r_3 \) or by an immediate value \( count_6 \). The shift count is interpreted as an unsigned number. If the value in GR \( r_3 \) is greater than 63, then the result is all zeroes.  

See “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 \(r_1 = r_2, count_2, r_3\)

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:

\[
\text{if } (PR[qp]) \{
\quad \text{check\_target\_register}(r_j);
\quad \text{GR}[r_j] = (\text{GR}[r_2] \ll count_2) + \text{GR}[r_3];
\quad \text{GR}[r_j].\text{nat} = \text{GR}[r_2].\text{nat} \mid\mid \text{GR}[r_3].\text{nat;}
\}
\]

Interruptions: Illegal Operation fault
shladdp4 — Shift Left and Add Pointer

**Format:**

\[(qp) \text{ shladdp}4 \ r1 = r2, \text{ count2}, r3\]

**Description:**
The first source operand is shifted to the left by \textit{count2} 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 \text{GR} r3 are copied to bits \{62:61\} of the result. This result is placed in \text{GR} r1. The first operand can be shifted by 1, 2, 3, or 4 bits.

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

**Operation:**

\[
\text{if (PR[qp])} \{
\text{check_target_register}(r1);}
\text{tmp_res} = (\text{GR}[r2] \ll \text{count2}) + \text{GR}[r3];
\text{tmp_res} = \text{zero_ext}((\text{tmp_res}(31:0), 32);\text{tmp_res}(62:61) = \text{GR}[r3](31:30);\text{GR}[r1] = \text{tmp_res};\text{GR}[r1].nat = \text{GR}[r2].nat || \text{GR}[r3].nat;\}
\]

**Interruptions:** Illegal Operation fault
**shr — Shift Right**

**Format:**

- (qp) shr \( r_1 = r_3, r_2 \)  
  signed_form \( I_5 \)
- (qp) shr.u \( r_1 = r_3, r_2 \)  
  unsigned_form \( I_5 \)
- (qp) shr \( r_1 = r_3, \text{count}_6 \)  
  pseudo-op of: (qp) extr \( r_1 = r_3, \text{count}_6, 64-\text{count}_6 \)
- (qp) shr.u \( r_1 = r_3, \text{count}_6 \)  
  pseudo-op of: (qp) extr.u \( r_1 = r_3, \text{count}_6, 64-\text{count}_6 \)

**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 \( \text{count}_6 \). 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:**

```c
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:**

\[
\text{if (PR[qp])} \{
\text{check_target_register} (r_1); \\
\text{temp1 = shift_right_unsigned} (\text{GR}[r_3], count_6); \\
\text{temp2 = GR}[r_2] \ll (64 - count_6); \\
\text{GR}[r_1] = \text{zero_ext} (\text{temp1}, 64 - count_6) \mid \text{temp2}; \\
\text{GR}[r_1].\text{nat} = \text{GR}[r_2].\text{nat} \mid | \text{GR}[r_3].\text{nat}; \\
\}
\]

**Interruptions:** Illegal Operation fault
srlz — Serialize

Format:  
(qp) srlz.i  
(qp) srlz.d

instruction_form  M24
data_form  M24

Description: Instruction serialization (srlz.i) ensures:
- prior modifications to processor register resources that affect fetching of 
  subsequent instruction groups are observed,
- prior modifications to processor register resources that affect subsequent execution 
  or data memory accesses are observed,
- prior memory synchronization (sync.i) operations have taken effect on the local 
  processor instruction cache,
- subsequent instruction group fetches are re-initiated after srlz.i completes.

The srlz.i instruction must be in an instruction group after the instruction group 
containing the operation that is to be serialized. Operations dependent on the 
serialization must be in an instruction group after the instruction group containing the 
srlz.i.

Data serialization (srlz.d) ensures:
- prior modifications to processor register resources that affect subsequent execution 
  or data memory accesses are observed.

The srlz.d instruction must be in an instruction group after the instruction group 
containing the operation that is to be serialized. Operations dependent on the 
serialization must follow the srlz.d, but they can be in the same instruction group as 
the srlz.d.

A srlz cannot be used to stall processor data memory references until prior data 
memory references, or memory fences are visible or “accepted” by the external 
platform.

The following processor resources require a serialize to ensure side-effects are 
observed; CRs, PSR, DBRs, IBRs, PMDs, PMCs, RRs, PKRs, TRs and TCs (refer to 
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 \(imm_{24}\)

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:

\[
\text{if (PR[qp])} \{
\text{if (PSR.cpl \neq 0) privileged_operation_fault(0);} \\
\text{if (is_reserved_field(PSR_TYPE, PSR_SM, imm_{24})) reserved_register_field_fault();} \\
\text{if (PSR.vm == 1) virtualization_fault();} \\
\text{if (imm_{24}[1]) PSR(1) = 1;} // be \\
\text{if (imm_{24}[2]) PSR(2) = 1;} // up \\
\text{if (imm_{24}[3]) PSR(3) = 1;} // ac \\
\text{if (imm_{24}[4]) PSR(4) = 1;} // mfl \\
\text{if (imm_{24}[5]) PSR(5) = 1;} // mfh \\
\text{if (imm_{24}[13]) PSR(13) = 1;} // ic \\
\text{if (imm_{24}[14]) PSR(14) = 1;} // i \\
\text{if (imm_{24}[15]) PSR(15) = 1;} // pk \\
\text{if (imm_{24}[17]) PSR(17) = 1;} // dt \\
\text{if (imm_{24}[18]) PSR(18) = 1;} // df1 \\
\text{if (imm_{24}[19]) PSR(19) = 1;} // dfh \\
\text{if (imm_{24}[20]) PSR(20) = 1;} // sp \\
\text{if (imm_{24}[21]) PSR(21) = 1;} // pp \\
\text{if (imm_{24}[22]) PSR(22) = 1;} // di \\
\text{if (imm_{24}[23]) PSR(23) = 1;} // si 
\}
\]

Interruptions: Privileged Operation fault
Virtualization fault
Reserved Register/Field fault

Serialization: Software must issue a data serialize or instruction serialize operation before issuing instructions dependent upon the altered PSR bits from the ssm instruction. Unlike with the rsm instruction, setting the PSR.i bit is not treated specially. Refer to Section 3.2, "Serialization" on page 2:17 for a description of serialization.
**st — Store**

**Format:**

- \((qp)\) stsz.sttype.sthint \[r3\] = \(r2\) normal_form, no_base_update_form M6
- \((qp)\) stsz.sttype.sthint \[r3\] = \(r2, imm_g\) normal_form, imm_base_update_form M5
- \((qp)\) st16.sttype.sthint \[r3\] = \(r2, ar.csd\) sixteen_byte_form, no_base_update_form M6
- \((qp)\) st8.spill.sthint \[r3\] = \(r2\) spill_form, no_base_update_form M6
- \((qp)\) st8.spill.sthint \[r3\] = \(r2, imm_g\) spill_form, imm_base_update_form M5

**Description:** A value consisting of the least significant \(sz\) bytes of the value in GR \(r2\) is written to memory starting at the address specified by the value in GR \(r3\). The values of the \(sz\) completer are given in Table 2-32 on page 3:151. The sttype completer specifies special store operations, which are described in Table 2-50. If the NaT bit corresponding to GR \(r3\) is 1, or in sixteen_byte_form or normal_form, if the NaT bit corresponding to GR \(r2\) 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 \(r2\) is written to memory starting at the address specified by the value in GR \(r3\). The value in the Compare and Store Data application register (AR[CS]) is written to memory starting at the address specified by the value in GR \(r3\) plus 8.

In the spill_form, an 8-byte value is stored, and the NaT bit corresponding to GR \(r2\) 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 \(r3\) is added to a signed immediate value \((imm_g)\) and the result is placed back in GR \(r3\). This base register update is done after the store, and does not affect the store address, nor the value stored (for the case where \(r2\) and \(r3\) specify the same register). Base register update is not supported for the st16 instruction.

**Table 2-50. Store Types**

<table>
<thead>
<tr>
<th>sttype Completer</th>
<th>Interpretation</th>
<th>Special Store Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Normal store</td>
<td></td>
</tr>
<tr>
<td>rel</td>
<td>Ordered store</td>
<td>An ordered store is performed with release semantics.</td>
</tr>
</tbody>
</table>

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 \(r3\) 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

<table>
<thead>
<tr>
<th>sthint</th>
<th>Completer</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td>Temporal locality, level 1</td>
</tr>
<tr>
<td>nta</td>
<td></td>
<td>Non-temporal locality, all levels</td>
</tr>
</tbody>
</table>

Operation:

```c
if (FR[qp]) {
    size = spill_form ? 8 : (sixteen_byte_form ? 16 : sz);
    itype = WRITE;
    if (size == 16) itype |= UNCACHE_OPT;
    otype = (sttype == '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);
    }
}
```

Interruptions:

- 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
<table>
<thead>
<tr>
<th>Data TLB fault</th>
<th>Unaligned Data Reference fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Page Not Present fault</td>
<td>Unsupported Data Reference fault</td>
</tr>
<tr>
<td>Data NaT Page Consumption fault</td>
<td></td>
</tr>
</tbody>
</table>
stf — Floating-point Store

Format: 

- (qp) stf$sz$.sthint $[r3] = f2$, normal_form, no_base_update_form M13
- (qp) stf$sz$.sthint $[r3] = f2, imm9$, normal_form, imm_base_update_form M10
- (qp) stf8.sthint $[r3] = f2$, integer_form, no_base_update_form M13
- (qp) stf8.sthint $[r3] = f2, imm9$, integer_form, imm_base_update_form M10
- (qp) stf.spill.sthint $[r3] = f2$, spill_form, no_base_update_form M13
- (qp) stf.spill.sthint $[r3] = f2, imm9$, spill_form, imm_base_update_form M10

Description: A value, consisting of $fsz$ bytes, is generated from the value in FR $f2$ and written to memory starting at the address specified by the value in GR $r3$. In the normal_form, the value in FR $f2$ is converted to the memory format and then stored. In the integer_form, the significand of FR $f2$ 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 $r3$ is 1 or if FR $f2$ 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 $f2$ 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 $r3$ is added to a signed immediate value $(imm9)$ and the result is placed back in GR $r3$. 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 $r3$ 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 stf (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 stf accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.
Operation:

```c
if (PR[qp]) {
    if (imm_base_update_form)
        check_target_register(r3);
    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, WRITE);
    if (GR[r3].nat || (!spill_form && (FR[f2] == NATVAL)))
        register_nat_consumption_fault(WRITE);
    size = spill_form ? 16 : (integer_form ? 8 : fsz);
    itype = WRITE;
    if (size == 10) itype |= UNCACHE_OPT;
    paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr, &tmp_unused);
    val = fp_fr_to_mem_format(FR[f2], size, integer_form);
    mem_write(val, paddr, size, UM.be, mattr, UNORDERED, sthint);
    alat_inval_multiple_entries(paddr, size);
    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = 0;
        mem_implicit_prefetch(GR[r3], sthint, WRITE);
    }
}
```

Interruptions:

- Illegal Operation fault
- Disabled Floating-point Register fault
- Register NaT Consumption fault
- Unimplemented Data Address fault
- Data Nested TLB fault
- Alternate Data TLB fault
- VHPT Data fault
- Data TLB fault
- Data Page Not Present fault
- Data NaT Page Consumption fault
- Data Key Miss fault
- Data Key Permission fault
- Data Access Rights fault
- Data Dirty Bit fault
- Data Access Bit fault
- Data Debug fault
- Unaligned Data Reference fault
- Unsupported Data Reference fault
sub — Subtract

Format:

\[
\begin{align*}
(qp) \text{ sub } r_1 &= r_2, r_3 & \text{register form} & \text{A1} \\
(qp) \text{ sub } r_1 &= r_2, r_3, 1 & \text{minus1 form, register form} & \text{A1} \\
(qp) \text{ sub } r_1 &= \text{imm}_8, r_3 & \text{imm8 form} & \text{A3}
\end{align*}
\]

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 \(\text{imm}_8\) encoding field.

The minus1 form is available only in the register form (although the equivalent effect can be achieved by adjusting the immediate).

Operation:

\[
\begin{align*}
\text{if } \{\text{PR}[qp]\} \{ \\
&\quad \text{check_target_register}(r_1); \\
&\quad \text{tmp\_src} = (\text{register form} ? \text{GR}[r_2] : \text{sign\_ext}(\text{imm}_8, 8)); \\
&\quad \text{tmp\_nat} = (\text{register form} ? \text{GR}[r_2].\text{nat} : 0); \\
&\quad \text{if } (\text{minus1 form}) \\
&\quad \quad \text{GR}[r_1] = \text{tmp\_src} - \text{GR}[r_3] - 1; \\
&\quad \text{else} \\
&\quad \quad \text{GR}[r_1] = \text{tmp\_src} - \text{GR}[r_3]; \\
&\quad \text{GR}[r_1].\text{nat} = \text{tmp\_nat} \| \text{GR}[r_3].\text{nat}; \\
&\}\}
\]

Interruptions: Illegal Operation fault
sum — Set User Mask

Format:  \((qp)\) sum \(imm_{24}\)  

Description: The \(imm_{24}\) 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:  

\[
\text{if } (PR[qp]) \{ \\
\quad \text{if } (\text{is_reserved_field}(PSR\_TYPE, PSR\_UM, imm_{24})) \\
\qquad \text{reserved_register_field_fault();} \\
\quad \text{if } (imm_{24}[1]) \quad \text{PSR}\{1\} = 1; \quad \text{// be} \\
\quad \text{if } (imm_{24}[2] \&\& \text{PSR.sp == 0}) \quad \text{//non-secure perf monitor} \\
\qquad \text{PSR}\{2\} = 1; \quad \text{// up} \\
\quad \text{if } (imm_{24}[3]) \quad \text{PSR}\{3\} = 1; \quad \text{// ac} \\
\quad \text{if } (imm_{24}[4]) \quad \text{PSR}\{4\} = 1; \quad \text{// mfl} \\
\quad \text{if } (imm_{24}[5]) \quad \text{PSR}\{5\} = 1; \quad \text{// mfh} \\
\}\]

Interruptions: Reserved Register/Field fault

Serialization: All user mask modifications are observed by the next instruction group.
**sxt — Sign Extend**

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

**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>
<thead>
<tr>
<th>xsz Mnemonic</th>
<th>Bit Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
</tr>
</tbody>
</table>

**Operation:**

```c
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

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):

```c
st [L1] = data              //store into local instruction stream
fc.i L1                      //flush stale datum from instruction/data cache
;;                          //require instruction boundary between fc.i and sync.i
sync.i                      //ensure local and remote data/inst caches
                           //are synchronized
;;
srlz.i                      //ensure sync has been observed by the local processor,
;;                          //ensure subsequent instructions observe
                           //modified memory
L1: target                  //instruction modified
```

Operation: if (PR[qp]) {
  instruction_synchronize();
}

Interruptions: None
tak — Translation Access Key

Format: \((qp)\) tak \(r_{1} = r_{3}\)

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:

\[
\text{if } (\text{PR}[qp]) \{ \\
\quad \text{itype} = \text{NON_ACCESS}\|\text{TAK}; \\
\quad \text{check_target_register(}r_{1}\text{);} \\
\quad \text{if } (\text{PSR.cpl} \neq 0) \\
\quad \quad \text{privileged_operation_fault(itype);} \\
\quad \text{if } (\text{GR}[r_{3}].nat) \\
\quad \quad \text{register_nat_consumption_fault(itype);} \\
\quad \text{if } (\text{PSR.vm} == 1) \\
\quad \quad \text{virtualization_fault();} \\
\quad \text{GR}[r_{1}] = \text{tlb_access_key(GR}[r_{3}], \text{itype);} \\
\quad \text{GR}[r_{1}].nat = 0; \\
\}
\]

Interruptions: Illegal Operation fault  Register NaT Consumption fault  Privileged Operation fault  Virtualization fault
**tbit — Test Bit**

**Format:** \((qp)\) tbit.trel.ctype \(p_1, p_2 = r_3, pos_6\)

**Description:** The bit specified by the \(pos_6\) immediate is selected from GR \(r_3\). The selected bit forms a single bit result either complemented or not depending on the \(trel\) completer. This result is written to the two predicate register destinations \(p_1\) and \(p_2\). The way the result is written to the destinations is determined by the compare type specified by \(ctype\). See the Compare instruction and Table 2-15 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 2-53. Test Bit Relations for Normal and unc tbits**

<table>
<thead>
<tr>
<th>(trel)</th>
<th>Test Relation</th>
<th>Pseudo-op of</th>
</tr>
</thead>
<tbody>
<tr>
<td>.(nz)</td>
<td>selected bit == 1</td>
<td>.(z) (p_1 \leftrightarrow p_2)</td>
</tr>
<tr>
<td>.(z)</td>
<td>selected bit == 0</td>
<td>(z) (p_1 \leftrightarrow p_2)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>(trel)</th>
<th>Test Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>.(nz)</td>
<td>selected bit == 1</td>
</tr>
<tr>
<td>.(z)</td>
<td>selected bit == 0</td>
</tr>
</tbody>
</table>

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:

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

    if (trel == 'nz') // 'nz' - test for 1
        tmp_rel = GR[r3][pos6];
    else // 'z' - test for 0
        tmp_rel = !GR[r3][pos6];

    switch (ctype) {
        case 'and': // and-type compare
            if ((GR[r3].nat || !tmp_rel) { // and-type compare
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;

        case 'or': // or-type compare
            if (!GR[r3].nat && tmp_rel) { // or-type compare
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;

        case 'or.andcm': // or.andcm-type compare
            if (!GR[r3].nat && tmp_rel) { // or.andcm-type compare
                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: \( \text{tf} \) \( \text{trel}.\text{ctype} \) \( p_1, p_2 = \text{imm} \)

Description: The \( \text{imm} \) 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 \( \text{trel} \) 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 \( \text{ctype} \). See the Compare instruction and Table 2-15 on page 3:39.

The \( \text{trel} \) 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

<table>
<thead>
<tr>
<th>( \text{trel} )</th>
<th>Test Relation</th>
<th>Pseudo-op of</th>
</tr>
</thead>
<tbody>
<tr>
<td>( .nz )</td>
<td>selected feature available</td>
<td>( z ) ( p_1 \leftrightarrow p_2 )</td>
</tr>
<tr>
<td>( .z )</td>
<td>selected feature unavailable</td>
<td></td>
</tr>
</tbody>
</table>

Table 2-56. Test Feature Relations for Parallel tf

<table>
<thead>
<tr>
<th>( \text{trel} )</th>
<th>Test Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( .nz )</td>
<td>selected feature available</td>
</tr>
<tr>
<td>( .z )</td>
<td>selected feature unavailable</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>( \text{imm} )</th>
<th>Feature Symbol</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>@clz</td>
<td>clz feature</td>
</tr>
<tr>
<td>33</td>
<td>@mpy</td>
<td>mpy4, mpyshl4 feature</td>
</tr>
<tr>
<td>34 - 63</td>
<td>none</td>
<td>Not currently defined</td>
</tr>
</tbody>
</table>
Operation:  

```c
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
thash — Translation Hashed Entry Address

Format: \((qp)\) thash \(r_1 = r_3\)  

Description: A Virtual Hashed Page Table (VHPT) entry address is generated based on the specified virtual address and the result is placed in GR \(r_1\). The virtual address is specified by GR \(r_3\) and the region register selected by GR \(r_3\) bits \{63:61\}.

If thash is given a NaT input argument or an unimplemented virtual address as an input, the resulting target register value is undefined, and its NaT bit is set to one.

When the processor is configured to use the region-based short format VHPT (PTA.vf=0), the value returned by thash is defined by the architected short format hash function. See 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), thash performs an implementation-specific long format hash function on the virtual address to generate a hash index into the long format VHPT.

In the long format, a translation in the VHPT must be uniquely identified by its hash index generated by this instruction and the hash tag produced from the ttag instruction.

The hash function must use all implemented region bits and only virtual address bits \{60:0\} to determine the offset into the VHPT. Virtual address bits \{63:61\} are used only by the short format hash to determine the region of the VHPT.

This instruction must be implemented on all processor models, even processor models that do not implement a VHPT walker.

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

Operation: if (PR[qp]) {
    check_target_register(r_1);
    if (PSR.vm == 1)
        virtualization_fault();
    if (GR[r_3].nat || unimplemented_virtual_address(GR[r_3], PSR.vm)) {
        GR[r_1] = undefined();
        GR[r_1].nat = 1;
    } else {
        tmp_vr = GR[r_3]{63:61};
        tmp_va = GR[r_3]{60:0};
        GR[r_1] = tlb_vhpt_hash(tmp_vr, tmp_va, RR[tmp_vr].rid,
                               RR[tmp_vr].ps);
        GR[r_1].nat = 0;
    }
}

Interruptions: Illegal Operation fault  Virtualization fault
tnat — Test NaT

Format: \((q)\text{ tnat.trel.typ } p_1, p_2 = r_3\)

Description: The NaT bit from GR \(r_3\) forms a single bit result, either complemented or not depending on the \(trel\) completer. This result is written to the two predicate register destinations, \(p_1\) and \(p_2\). The way the result is written to the destinations is determined by the compare type specified by \(ctype\). See the Compare instruction and Table 2-15 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

<table>
<thead>
<tr>
<th>(trel)</th>
<th>Test Relation</th>
<th>Pseudo-op of</th>
</tr>
</thead>
<tbody>
<tr>
<td>nz</td>
<td>selected bit == 1</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>selected bit == 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(p_1 \leftrightarrow p_2)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2-59. Test NaT Relations for Parallel tnats

<table>
<thead>
<tr>
<th>(trel)</th>
<th>Test Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>nz</td>
<td>selected bit == 1</td>
</tr>
<tr>
<td>z</td>
<td>selected bit == 0</td>
</tr>
</tbody>
</table>

If the two predicate register destinations are the same \((p_1\) and \(p_2\) specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is set, or if the compare type is unc.
Operation:
if (PR[qp]) {
   if (p1 == p2)
      illegal_operation_fault();

   if (trel == 'nz') // 'nz' - test for 1
      tmp_rel = GR[r3].nat;
   else // 'z' - test for 0
      tmp_rel = !GR[r3].nat;

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

Interruptions: Illegal Operation fault
tpa — Translate to Physical Address

**Format:** \((qp)\) tpa \(r_1 = r_3\)

**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:**

```c
if (PR[qp]) {
  itype = NON_ACCESS|TPA;
  check_target_register(r1);
  if (PSR.cpl != 0)
    privileged_operation_fault(itype);
  if (GR[r3].nat)
    register_nat_consumption_fault(itype);
  GR[r1] = tlb_translate_nonaccess(GR[r3], itype);
  GR[r1].nat = 0;
}
```

**Interruptions:**

<table>
<thead>
<tr>
<th>Illegal Operation fault</th>
<th>Alternate Data TLB fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>Privileged Operation fault</td>
<td>VHPT Data fault</td>
</tr>
<tr>
<td>Register NaT Consumption fault</td>
<td>Data TLB fault</td>
</tr>
<tr>
<td>Unimplemented Data Address fault</td>
<td>Data Page Not Present fault</td>
</tr>
<tr>
<td>Virtualization fault</td>
<td>Data NaT Page Consumption fault</td>
</tr>
<tr>
<td>Data Nested TLB fault</td>
<td></td>
</tr>
</tbody>
</table>
ttag — Translation Hashed Entry Tag

**Format:** \((qp)\) \(t\ t\ a\ g\ r_1 = r_3\)

**Description:** A tag used for matching during searches of the long format Virtual Hashed Page Table (VHPT) is generated and placed in \(r_1\). The virtual address is specified by \(r_3\) and the region register selected by \(r_3\) 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 \(\text{thash}\) 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:**

```c
if (PR[qp]) {
    check_target_register(r_1);
    if (PSR.vm == 1)
        virtualization_fault();
    if (GR[r_3].nat || unimplemented_virtual_address(GR[r_3], PSR.vm)) {
        GR[r_1] = undefined();
        GR[r_1].nat = 1;
    } else {
        tmp_vr = GR[r_3]{63:61};
        tmp_va = GR[r_3]{60:0};
        GR[r_1] = tlb_vhpt_tag(tmp_va, RR[tmp_vr].rid, RR[tmp_vr].ps);
        GR[r_1].nat = 0;
    }
}
```

**Interruptions:** Illegal Operation fault, Virtualization fault
unpack — Unpack

Format:

- `(qp) unpack1.h` \( r_1 = r_2, r_3 \)
- `(qp) unpack2.h` \( r_1 = r_2, r_3 \)
- `(qp) unpack4.h` \( r_1 = r_2, r_3 \)
- `(qp) unpack1.l` \( r_1 = r_2, r_3 \)
- `(qp) unpack2.l` \( r_1 = r_2, r_3 \)
- `(qp) unpack4.l` \( r_1 = r_2, r_3 \)

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
**unpack**

**Operation:**

```c
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  
vmsw.1  

zero_form  B8  
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;  
}

Interruptions:  
Illegal Operation fault  
Virtualization fault  
Privileged Operation fault
xchg — Exchange

Format: \((qp)\) xchg sz ldhint \(r_1 = [r_3], r_2\)

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

<table>
<thead>
<tr>
<th>(sz) Completer</th>
<th>Bytes Accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 byte</td>
</tr>
<tr>
<td>2</td>
<td>2 bytes</td>
</tr>
<tr>
<td>4</td>
<td>4 bytes</td>
</tr>
<tr>
<td>8</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

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 \(ldhint\) 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.
Operation:
if (PR[gp]) {
    check_target_register(r);
    if (GR[r].nat || GR[r].nat)
        register_nat_consumption_fault(SEMAPHORE);
    paddr = tlb_translate(GR[r], sz, SEMAPHORE, PSR.cpi, &mattr,
        &tmp_unused);
    if (!ma_supports_semaphores(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r]);
    val = mem_xchg(GR[r], paddr, sz, UM.be, mattr, ACQUIRE, Idhint);
    alat_inval_multiple_entries(paddr, sz);
    GR[r] = zero_ext(val, sz * 8);
    GR[r].nat = 0;
}

Interruptions:
Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault

Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault
xma

xma — Fixed-Point Multiply Add

Format:

(qp) xma.l \( f_1 = f_3, f_4, f_2 \)           low_form   F2
(qp) xma.lu \( f_1 = f_3, f_4, f_2 \)    pseudo-op of:  (qp) xma.l \( f_1 = f_3, f_4, f_2 \)
(qp) xma.h \( f_1 = f_3, f_4, f_2 \)   high_form   F2
(qp) xma.hu \( f_1 = f_3, f_4, f_2 \)   high_unsigned_form   F2

Description:

Two source operands (FR \( f_3 \) and FR \( f_4 \)) are treated as either signed or unsigned integers and multiplied. The third source operand (FR \( f_2 \)) is zero extended and added to the product. The upper or lower 64 bits of the resultant sum are selected and placed in FR \( f_1 \).

In the high_unsigned_form, the significand fields of FR \( f_3 \) and FR \( f_4 \) are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The significand field of FR \( f_2 \) is zero extended and added to the product. The most significant 64-bits of the resultant sum are placed in the significand field of FR \( f_1 \).

In the high_form, the significand fields of FR \( f_3 \) and FR \( f_4 \) are treated as signed integers and multiplied to produce a full 128-bit signed result. The significand field of FR \( f_2 \) is zero extended and added to the product. The most significant 64-bits of the resultant sum are placed in the significand field of FR \( f_1 \).

In the other forms, the significand fields of FR \( f_3 \) and FR \( f_4 \) are treated as signed integers and multiplied to produce a full 128-bit signed result. The significand field of FR \( f_2 \) is zero extended and added to the product. The least significant 64-bits of the resultant sum are placed in the significand field of FR \( f_1 \).

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

Note: \( f_1 \) 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:

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

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
    } else {
        if (low_form || high_form)
            tmp_res_128 =
                fp_I64_x_I64_to_I128(FR[f3].significand, FR[f4].significand);
        else // high_unsigned_form
            tmp_res_128 =
                fp_U64_x_U64_to_U128(FR[f3].significand, FR[f4].significand);

        tmp_res_128 =
            fp_U128_add(tmp_res_128, fp_U64_to_U128(FR[f2].significand));

        if (high_form || high_unsigned_form)
            FR[f1].significand = tmp_res_128.hi;
        else // low_form
            FR[f1].significand = tmp_res_128.lo;

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

Interruptions: Disabled Floating-point Register fault
xmpy — Fixed-Point Multiply

Format:

- (qp) xmpy.l  \( f_3 \times f_4 \)  
  pseudo-op of: (qp) xma.l  \( f_3 \times f_4 \), f0
- (qp) xmpy.lu  \( f_3 \times f_4 \)  
  pseudo-op of: (qp) xma.l  \( f_3 \times f_4 \), f0
- (qp) xmpy.h  \( f_3 \times f_4 \)  
  pseudo-op of: (qp) xma.h  \( f_3 \times f_4 \), f0
- (qp) xmpy.hu  \( f_3 \times f_4 \)  
  pseudo-op of: (qp) xma.hu  \( f_3 \times f_4 \), f0

Description: Two source operands (FR \( f_3 \) and FR \( f_4 \)) are treated as either signed or unsigned integers and multiplied. The upper or lower 64 bits of the resultant product are selected and placed in FR \( f_1 \).

In the high_unsigned_form, the significand fields of FR \( f_3 \) and FR \( f_4 \) are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The most significant 64-bits of the resultant product are placed in the significand field of FR \( f_1 \).

In the high_form, the significand fields of FR \( f_3 \) and FR \( f_4 \) are treated as signed integers and multiplied to produce a full 128-bit signed result. The most significant 64-bits of the resultant product are placed in the significand field of FR \( f_1 \).

In the other forms, the significand fields of FR \( f_3 \) and FR \( f_4 \) are treated as signed integers and multiplied to produce a full 128-bit signed result. The least significant 64-bits of the resultant product are placed in the significand field of FR \( f_1 \).

In all forms, the exponent field of FR \( f_1 \) is set to the biased exponent for 2.0^{128} (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.

 xor — Exclusive Or

**Format:**  
(\texttt{q}) \texttt{xor} \quad r_1 = r_2, r_3 \quad \text{register form} \quad \text{A1}  
(\texttt{q}) \texttt{xor} \quad r_1 = \text{imm}_8, r_3 \quad \text{imm8 form} \quad \text{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 \( \text{imm}_8 \) encoding field.

**Operation:**  
if \( \text{PR}[\texttt{q}] \) {  
  check_target_register(\( r_1 \));  
  \text{tmp} \_\text{src} = (\text{register_form} \ ? \ \text{GR}[r_2] \ : \ \text{sign\_ext}(\text{imm}_8, 8));  
  \text{tmp} \_\text{nat} = (\text{register_form} \ ? \ \text{GR}[r_2].\text{nat} : 0);  
  \text{GR}[r_1] = \text{tmp} \_\text{src} \ ^{\wedge} \ \text{GR}[r_3];  
  \text{GR}[r_1].\text{nat} = \text{tmp} \_\text{nat} \ | \ | \ \text{GR}[r_3].\text{nat};  
}

**Interruptions:**  
Illegal Operation fault
**zxt — Zero Extend**

**Format:**  
(qp) zxtxsz r1 = r3

**Description:** The value in GR r3 is zero extended above the bit position specified by xsz and the result is placed in GR r1. 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>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx_fault(parameters ...)</td>
<td>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.</td>
</tr>
<tr>
<td>xxx_trap(parameters ...)</td>
<td>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.</td>
</tr>
<tr>
<td>acceptance_fence()</td>
<td>Ensures prior data memory references to uncached ordered-sequential memory pages are “accepted” before subsequent data memory references are performed by the processor.</td>
</tr>
<tr>
<td>alat_cmp(rtype, raddr)</td>
<td>Returns a one if the implementation finds an ALAT entry which matches the register type specified by rtype and the register address specified by raddr, else returns zero. This function is implementation specific. Note that an implementation may optionally choose to return zero (indicating no match) even if a matching entry exists in the ALAT. This provides implementation flexibility in designing fast ALAT lookup circuits.</td>
</tr>
<tr>
<td>alat_frame_update(delta_bof, delta_sof)</td>
<td>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.</td>
</tr>
<tr>
<td>alat_inval()</td>
<td>Invalidate all entries in the ALAT.</td>
</tr>
<tr>
<td>alat_inval_multiple_entries(paddr, size)</td>
<td>The ALAT is queried using the physical memory address specified by paddr and the access size specified by size. All matching ALAT entries are invalidated. No value is returned.</td>
</tr>
<tr>
<td>alat_inval_single_entry(rtype, rega)</td>
<td>The ALAT is queried using the register type specified by rtype and the register address specified by rega. At most one matching ALAT entry is invalidated. No value is returned.</td>
</tr>
<tr>
<td>alat_read_memory_on_hit(ldtype, rtype, raddr)</td>
<td>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.</td>
</tr>
<tr>
<td>alat_translate_address_on_hit(ldtype, rtype, raddr)</td>
<td>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.</td>
</tr>
<tr>
<td>alat_write(ldtype, rtype, raddr, paddr, size)</td>
<td>Allocates a new ALAT entry or updates an existing entry using the load type specified by ldtype, the register type specified by rtype, the register address specified by raddr, the physical memory address specified by paddr, and the access size specified by size. No value is returned. This function guarantees that at most only one ALAT entry exists for a given paddr. Based on the load type ldtype, if a ld.c.nc, ldf.c.nc, or ldfp.c.nc instruction’s raddr matches an existing ALAT entry’s register tag, but the instruction’s size and/or paddr are different than that of the existing entry’s, then this function may either preserve the existing entry, or invalidate it and write a new entry with the instruction’s specified size and paddr.</td>
</tr>
<tr>
<td>align_to_size_boundary(vaddr, size)</td>
<td>Returns vaddr aligned to the boundary specified by size.</td>
</tr>
<tr>
<td>branch_predict(wh, ih, ret, target, tag)</td>
<td>Implementation-dependent routine which updates the processor’s branch prediction structures.</td>
</tr>
</tbody>
</table>
Table 3-1.  Pseudo-code Functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_branch_implemented(check_type)</td>
<td>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.</td>
</tr>
<tr>
<td>check_probe_virtualization_fault(type, cpl)</td>
<td>If implemented, this function may raise virtualization faults for specific probe instructions. Please refer to the instruction page for probe instruction for details.</td>
</tr>
<tr>
<td>check_target_register(r1)</td>
<td>If the r1 argument specifies an out-of-frame stacked register (as defined by CFM) or r1 specifies GR0, an Illegal Operation fault is delivered, and this function does not return.</td>
</tr>
<tr>
<td>check_target_register_sof(r1, newsof)</td>
<td>If the r1 argument specifies an out-of-frame stacked register (as defined by the newsof argument) or r1 specifies GR0, an Illegal Operation fault is delivered and this function does not return.</td>
</tr>
<tr>
<td>concatenate2(x1, x2)</td>
<td>Concatenates the lower 32 bits of the 2 arguments, and returns the 64-bit result.</td>
</tr>
<tr>
<td>concatenate4(x1, x2, x3, x4)</td>
<td>Concatenates the lower 16 bits of the 4 arguments, and returns the 64-bit result.</td>
</tr>
<tr>
<td>concatenate8(x1, x2, x3, x4, x5, x6, x7, x8)</td>
<td>Concatenates the lower 8 bits of the 8 arguments, and returns the 64-bit result.</td>
</tr>
<tr>
<td>data_serialize()</td>
<td>Ensures all prior register updates with side-effects are observed before subsequent execution and data memory references are performed.</td>
</tr>
<tr>
<td>deliver_unmasked_pending_interrupt()</td>
<td>This implementation-specific function checks whether any unmasked external interrupts are pending, and if so, transfers control to the external interrupt vector.</td>
</tr>
<tr>
<td>execute_hint(hint)</td>
<td>Executes the hint specified by hint.</td>
</tr>
<tr>
<td>fadd(fp_dp, fr2)</td>
<td>Adds a floating-point register value to the infinitely precise product and return the infinitely precise sum, ready for rounding.</td>
</tr>
<tr>
<td>fcmp_exception_fault_check(f2, f3, frel, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the fcmp instruction.</td>
</tr>
<tr>
<td>fctv_fx_exception_fault_check(fr2, signed_form, trunc_form, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the fcvt.fx, fcvt.fxu, fcvt.fx.trunc and fcvt.fxu.trunc instructions. It propagates NaNs.</td>
</tr>
<tr>
<td>fma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the fma instruction. It propagates NaNs and special IEEE results.</td>
</tr>
<tr>
<td>fnminmax_exception_fault_check(f2, f3, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the famin, famax, fmax, and fmin instructions.</td>
</tr>
<tr>
<td>fms_fnma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the fms and fnma instructions. It propagates NaNs and special IEEE results.</td>
</tr>
<tr>
<td>fmul(fr3, fr4)</td>
<td>Performs an infinitely precise multiply of two floating-point register values.</td>
</tr>
<tr>
<td>followed_by_stop()</td>
<td>Returns TRUE if the current instruction is followed by a stop; otherwise, returns FALSE.</td>
</tr>
<tr>
<td>fp_check_target_register(f1)</td>
<td>If the specified floating-point register identifier is 0 or 1, this function causes an illegal operation fault.</td>
</tr>
<tr>
<td>fp_decode_fault(tmp_fp_env)</td>
<td>Returns floating-point exception fault code values for ISR.code.</td>
</tr>
<tr>
<td>fp_decode_traps(tmp_fp_env)</td>
<td>Returns floating-point trap code values for ISR.code.</td>
</tr>
<tr>
<td>fp_equal(fr1, fr2)</td>
<td>IEEE standard equality relationship test.</td>
</tr>
<tr>
<td>fp_fr_to_mem_format(freg, size)</td>
<td>Converts a floating-point value in register format to floating-point memory format. It assumes that the floating-point value in the register has been previously rounded to the correct precision which corresponds with the size parameter.</td>
</tr>
<tr>
<td>fp_ieee_recip(num, den)</td>
<td>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.</td>
</tr>
<tr>
<td>fp_ieee_recip_sqrt(root)</td>
<td>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.</td>
</tr>
<tr>
<td>fp_is_nan(freg)</td>
<td>Returns true when floating register contains a NaN.</td>
</tr>
</tbody>
</table>
Table 3-1. Pseudo-code Functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp_is_nan_or_inf(freg)</td>
<td>Returns true if the floating-point exception fault_check functions returned a IEEE fault disabled default result or a propagated NaN.</td>
</tr>
<tr>
<td>fp_is_natval(freg)</td>
<td>Returns true when floating register contains a NaTVal</td>
</tr>
<tr>
<td>fp_is_normal(freg)</td>
<td>Returns true when floating register contains a normal number.</td>
</tr>
<tr>
<td>fp_is_pos_inf(freg)</td>
<td>Returns true when floating register contains a positive infinity.</td>
</tr>
<tr>
<td>fp_is_qnan(freg)</td>
<td>Returns true when floating register contains a quiet NaN.</td>
</tr>
<tr>
<td>fp_is_snan(freg)</td>
<td>Returns true when floating register contains a signalling NaN.</td>
</tr>
<tr>
<td>fp_is_unorm(freg)</td>
<td>Returns true when floating register contains an unnormalized number.</td>
</tr>
<tr>
<td>fp_is_unsupported(freg)</td>
<td>Returns true when floating register contains an unsupported format.</td>
</tr>
<tr>
<td>fp_less_than(fr1, fr2)</td>
<td>IEEE standard less-than relationship test.</td>
</tr>
<tr>
<td>fp_lesser_or_equal(fr1, fr2)</td>
<td>IEEE standard less-than or equal-to relationship test</td>
</tr>
<tr>
<td>fp_mem_to_fr_format(mem, size)</td>
<td>Converts a floating-point value in memory format to floating-point register format.</td>
</tr>
<tr>
<td>fp_normalize(fr1)</td>
<td>Normalizes an unnormalized fp value. This function flushes to zero any unnormal values which can not be represented in the register file</td>
</tr>
<tr>
<td>fp_raise_fault(tmp_fp_env)</td>
<td>Checks the local instruction state for any faulting conditions which require an interruption to be raised.</td>
</tr>
<tr>
<td>fp_raise_traps(tmp_fp_env)</td>
<td>Checks the local instruction state for any trapping conditions which require an interruption to be raised.</td>
</tr>
<tr>
<td>fp_reg_bank_conflict(f1, f2)</td>
<td>Returns true if the two specified FRs are in the same bank.</td>
</tr>
<tr>
<td>fp_reg_disabled(f1, f2, f3, f4)</td>
<td>Check for possible disabled floating-point register faults.</td>
</tr>
<tr>
<td>fp_reg_read(freg)</td>
<td>Reads the FR and gives canonical double-extended denormals (and pseudo-denormals) their true mathematical exponent. Other classes of operands are unaltered.</td>
</tr>
<tr>
<td>fp_unordered(fr1, fr2)</td>
<td>IEEE standard unordered relationship</td>
</tr>
<tr>
<td>fp_update_fpsr(sf, tmp_fp_env)</td>
<td>Copies a floating-point instruction's local state into the global FPSR.</td>
</tr>
<tr>
<td>fp_update_par(dest_freg)</td>
<td>Conditionally sets PSR.mfl or PSR.mfh based on dest_freg.</td>
</tr>
<tr>
<td>fpcmp_exception_fault_check(f2, f3, frel, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the fpcmp instruction.</td>
</tr>
<tr>
<td>fpcvt_exception_fault_check(f2, signed_form, trunc_form, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the fpcvt.fx, fpcvt.fxu, fpcvt.fx.trunc, and fpcvt.fxu.trunc instructions. It propagates NaNs.</td>
</tr>
<tr>
<td>fpma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the fpma instruction. It propagates NaNs and special IEEE results.</td>
</tr>
<tr>
<td>fpmminmax_exception_fault_check(f2, f3, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the fpmin, fpmax, fpamin and fpamax instructions.</td>
</tr>
<tr>
<td>fms.fpnuma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the fpma and fpnuma instructions. It propagates NaNs and special IEEE results.</td>
</tr>
<tr>
<td>fprcpa_exception_fault_check(f2, f3, sf, *tmp_fp_env, *limits_check)</td>
<td>Checks for all floating-point faulting conditions for the fprcpa instruction. It propagates NaNs and special IEEE results. It also indicates operand limit violations.</td>
</tr>
<tr>
<td>fprsqrta_exception_fault_check(f3, sf, *tmp_fp_env, *limits_check)</td>
<td>Checks for all floating-point faulting conditions for the fprsqrta instruction. It propagates NaNs and special IEEE results. It also indicates operand limit violations.</td>
</tr>
<tr>
<td>frcpa_exception_fault_check(f2, f3, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the frcpa instruction. It propagates NaNs and special IEEE results.</td>
</tr>
<tr>
<td>frsqrta_exception_fault_check(f3, sf, *tmp_fp_env)</td>
<td>Checks for all floating-point faulting conditions for the frsqrta instruction. It propagates NaNs and special IEEE results.</td>
</tr>
<tr>
<td>ignored_field_mask(regclass, reg, value)</td>
<td>Boolean function that returns value with bits cleared to 0 corresponding to ignored bits for the specified register and register type.</td>
</tr>
</tbody>
</table>
**Table 3-1. Pseudo-code Functions (Continued)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>impl_check_mov_itir()</td>
<td>Implementation-specific function that returns TRUE if ITIR is checked for reserved fields and encodings on a mov to ITIR instruction.</td>
</tr>
<tr>
<td>impl_check_mov_psr_l(gr)</td>
<td>Implementation-specific function to check bits (63:32) of gr corresponding to reserved fields of the PSR for Reserved Register/Field fault.</td>
</tr>
<tr>
<td>impl_check_tlb_itir()</td>
<td>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.</td>
</tr>
<tr>
<td>impl_gitc_enable()</td>
<td>Implementation-specific function that indicates whether guest MOV-from-AR.ITC optimization is enabled.</td>
</tr>
<tr>
<td>impl_ia32_ar_reserved_ignored(ar3)</td>
<td>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.</td>
</tr>
<tr>
<td>impl_iib()</td>
<td>Implementation-specific function which indicates whether Interruption Instruction Bundle registers (IIB0-1) are implemented.</td>
</tr>
<tr>
<td>impl_itir_owi_mask()</td>
<td>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}.</td>
</tr>
<tr>
<td>impl_iio()</td>
<td>Implementation-specific function which indicates whether Interval Timer Offset (ITO) register is implemented.</td>
</tr>
<tr>
<td>impl_probe_intercept()</td>
<td>Implementation-specific function indicates whether probe interceptions are supported.</td>
</tr>
<tr>
<td>impl_ruc()</td>
<td>Implementation-specific function which indicates whether Resource Utilization Counter (RUC) application register is implemented.</td>
</tr>
<tr>
<td>impl_uia_fault_supported()</td>
<td>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.</td>
</tr>
<tr>
<td>implemented_vm()</td>
<td>Returns TRUE if the processor implements the PSR.vm bit (regardless of whether virtual machine features are enabled or disabled).</td>
</tr>
<tr>
<td>instruction_implemented(inst)</td>
<td>Implementation-dependent routine which returns TRUE or FALSE, depending on whether inst is implemented.</td>
</tr>
<tr>
<td>instruction_serialize()</td>
<td>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.</td>
</tr>
<tr>
<td>instruction_synchronize()</td>
<td>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.</td>
</tr>
<tr>
<td>is_finite(freg)</td>
<td>Returns true when floating register contains a finite number.</td>
</tr>
<tr>
<td>is_ignored_reg(regnum)</td>
<td>Boolean function that returns true if regnum is an ignored application register, otherwise false.</td>
</tr>
<tr>
<td>is_inf(freg)</td>
<td>Returns true when floating register contains an infinite number.</td>
</tr>
<tr>
<td>is_interruption_cr(regnum)</td>
<td>Boolean function that returns true if regnum is one of the Interruption Control registers (see Section 3.3.5, “Interruption Control Registers” on page 2:36), otherwise false.</td>
</tr>
<tr>
<td>is_kernel_reg(ar_addr)</td>
<td>Returns a one if ar_addr is the address of a kernel register application register.</td>
</tr>
</tbody>
</table>
is_read_only_reg(rtype, raddr) Returns a one if the register addressed by raddr in the register bank of type rtype is a read only register.

is_reserved_field(regclass, arg2, arg3) Returns true if the specified data would write a one in a reserved field.

is_reserved_reg(regclass, regnum) Returns true if register regnum is reserved in the regclass register file.

is_supported_hint(hint) Returns true if the implementation supports the specified hint. This function may depend on factors other than the hint 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 INVALID_AR if PSR.it is 0.

make_icache_coherent(paddr) The cache line addressed by the physical address paddr 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 paddr is invalidated in all levels of the memory hierarchy above memory and written back to memory if it is inconsistent with memory.

mem_flush_pending_stores() The processor is instructed to start draining pending stores in write coalescing and write buffers. This operation is a hint. There is no indication when prior stores have actually been drained.

mem_implicit_prefetch(vaddr, hint, type) Moves the line addressed by vaddr to the location of the memory hierarchy specified by hint. This function is implementation dependent and can be ignored. The type allows the implementation to distinguish prefetches for different instruction types.

mem_promote(paddr, mtype, hint) Moves the line addressed by paddr to the highest level of the memory hierarchy conditioned by the access hints specified by hint. Implementation dependent and can be ignored.

mem_read(paddr, size, border, mattr, otype, hint) Returns the size bytes starting at the physical memory location specified by paddr with byte order specified by border, memory attributes specified by mattr, and access hint specified by hint. otype specifies the memory ordering attribute of this access, and must be UNORDERED or ACQUIRE.

mem_read_pair(*low_value, *high_value, paddr, size, border, mattr, otype, hint) Reads the size / 2 bytes of memory starting at the physical memory address specified by paddr into low_value, and the size / 2 bytes of memory starting at the physical memory address specified by (paddr + size / 2) into high_value, with byte order specified by border, memory attributes specified by mattr, and access hint specified by hint. otype specifies the memory ordering attribute of this access, and must be UNORDERED or ACQUIRE. No value is returned.

mem_write(value, paddr, size, border, mattr, otype, hint) Writes the least significant size bytes of value into memory starting at the physical memory address specified by paddr with byte order specified by border, memory attributes specified by mattr, and access hint specified by hint. otype specifies the memory ordering attribute of this access, and must be UNORDERED or RELEASE. No value is returned.

mem_write16(gr_value, ar_value, paddr, border, mattr, otype, hint) Writes the 8 bytes of gr_value into memory starting at the physical memory address specified by paddr, and the 8 bytes of ar_value into memory starting at the physical memory address specified by (paddr + 8), with byte order specified by border, memory attributes specified by mattr, and access hint specified by hint. otype specifies the memory ordering attribute of this access, and must be UNORDERED or RELEASE. No value is returned.

mem_xchg(data, paddr, size, byte_order, mattr, otype, hint) Returns size bytes from memory starting at the physical address specified by paddr. The read is conditioned by the locality hint specified by hint. After the read, the least significant size bytes of data are written to size bytes in memory starting at the physical address specified by paddr. The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by mattr and the byte ordering in memory is specified by byte_order. otype specifies the memory ordering attribute of this access, and must be ACQUIRE.

Table 3-1. Pseudo-code Functions (Continued)
<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem_xchg_add(add_val, paddr, size, byte_order, mattr, otype, hint)</td>
<td>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.</td>
</tr>
<tr>
<td>mem_xchg_cond(cmp_val, data, paddr, size, byte_order, mattr, otype, hint)</td>
<td>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 <code>data</code> 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.</td>
</tr>
<tr>
<td>mem_xchg16_cond(cmp_val, gr_data, ar_data, paddr, byte_order, mattr, otype, hint)</td>
<td>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.</td>
</tr>
<tr>
<td>ordering_fence()</td>
<td>Ensures prior data memory references are made visible before future data memory references are made visible by the processor.</td>
</tr>
<tr>
<td>partially_implemented_ip()</td>
<td>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.</td>
</tr>
<tr>
<td>pending_virtual_interrupt()</td>
<td>Check for unmasked pending virtual interrupt.</td>
</tr>
<tr>
<td>pr_phys_to_virt(phys_id)</td>
<td>Returns the virtual register id of the predicate from the physical register id, <code>phys_id</code> of the predicate.</td>
</tr>
<tr>
<td>rotate_regs()</td>
<td>Decrements the Register Rename Base registers, effectively rotating the register files. CFM.rrb.gr is decremented only if CFM.sor is non-zero.</td>
</tr>
<tr>
<td>rse_enable_current_frame_load()</td>
<td>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.</td>
</tr>
<tr>
<td>rse_ensure_regs_loaded(number_of_bytes)</td>
<td>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 Table 6-6, “RSE Interruption Summary” on page 6-145.</td>
</tr>
<tr>
<td>rseinvalidate_non_current_regs()</td>
<td>All registers outside the current frame are invalidated.</td>
</tr>
</tbody>
</table>
### Table 3-1. Pseudo-code Functions (Continued)

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>spontaneous_deferral(paddr, size, border, mattr, otype, hint, defer)</td>
<td>Implementation-dependent routine which optionally forces defer 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).</td>
</tr>
<tr>
<td>spontaneous_deferral_enabled()</td>
<td>Implementation-dependent routine which returns TRUE or FALSE, depending on whether spontaneous deferral of speculative loads is enabled or disabled in the processor.</td>
</tr>
<tr>
<td>tlb_access_key(vaddr, itype)</td>
<td>This function returns, in bits 31:8, the access key from the TLB for the entry corresponding to vaddr and itype; bits 63:32 and 7:0 return 0. If vaddr is an unimplemented virtual address, or a matching present translation is not found, the value 1 is returned.</td>
</tr>
<tr>
<td>tlb_broadcast_purge(rid, vaddr, size, type)</td>
<td>Sends a broadcast purge DTC and ITC transaction to other processors in the multiprocessor coherency domain, where the region identifier (rid), virtual address (vaddr) and page size (size) specify the translation entry to purge. The operation waits until all processors that receive the purge have completed the purge operation. The purge type (type) specifies whether the ALAT on other processors should also be purged in conjunction with the TC.</td>
</tr>
<tr>
<td>tlb_enter_privileged_code()</td>
<td>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.</td>
</tr>
</tbody>
</table>
| tlb_grant_permission(vaddr, type, pl)        | Returns a boolean indicating if read, write access is granted for the specified virtual memory address (vaddr) and privilege level (pl). The access type (type) specifies either read or write. The following faults are checked:  
- 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  
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. pte0, pte1 compose the translation. vaddr and rid specify the virtual address and region identifier for the translation. If tr is true the entry is placed in the TR section, otherwise the TC section. |
| tlb_insert_inst(slot, pte0, pte1, vaddr, rid, tr) | Inserts an entry into the ITLB, at the specified slot number. pte0, pte1 compose the translation. vaddr and rid specify the virtual address and region identifier for the translation. If tr is true, the entry is placed in the TR section, otherwise the TC section. |
| tlb_may_purge_dtc_entries(rid, vaddr, size)   | May locally purge DTC entries that match the specified virtual address (vaddr), region identifier (rid) and page size (size). May also invalidate entries that partially overlap the parameters. The extent of purging is implementation dependent. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache. |
### Table 3-1. Pseudo-code Functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tlb_may_purge_itc_entries(rid, vaddr, size)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>tlb_must_purge_dtc_entries(rid, vaddr, size)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>tlb_must_purge_dtr_entries(rid, vaddr, size)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>tlb_must_purge_itc_entries(rid, vaddr, size)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>tlb_must_purge_itr_entries(rid, vaddr, size)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>tlb_purge_translation_cache(loop)</code></td>
<td>Removes 1 to <code>N</code> 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.</td>
</tr>
<tr>
<td><code>tlb_replacement_algorithm(tlb)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>tlb_search_pkr(key)</code></td>
<td>Searches for a valid protection key register with a matching protection key. The search algorithm is implementation specific. Returns the PKR register slot number if found, otherwise returns Not Found.</td>
</tr>
</tbody>
</table>
Table 3-1.  Pseudo-code Functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>tlb_translate(vaddr, size, type, cpl, *attr, *defer)</td>
<td>Returns the translated data physical address for the specified virtual memory address (vaddr) when translation enabled; otherwise, returns vaddr. size specifies the size of the access, type specifies the type of access (e.g., read, write, advance, spec). cpl specifies the privilege level for access checking purposes. *attr returns the mapped physical memory attribute. If any fault conditions are detected and deferred, tlb_translate returns with *defer set. If a fault is generated but the fault is not deferred, tlb_translate does not return. tlb_translate checks the following faults:</td>
</tr>
<tr>
<td></td>
<td>• Unimplemented Data Address fault</td>
</tr>
<tr>
<td></td>
<td>• Data Nested TLB fault</td>
</tr>
<tr>
<td></td>
<td>• Alternate Data TLB fault</td>
</tr>
<tr>
<td></td>
<td>• VHPT Data fault</td>
</tr>
<tr>
<td></td>
<td>• Data TLB fault</td>
</tr>
<tr>
<td></td>
<td>• Data Page Not Present fault</td>
</tr>
<tr>
<td></td>
<td>• Data NaT Page Consumption fault</td>
</tr>
<tr>
<td></td>
<td>• Data Key Miss fault</td>
</tr>
<tr>
<td></td>
<td>• Data Key Permission fault</td>
</tr>
<tr>
<td></td>
<td>• Data Access Rights fault</td>
</tr>
<tr>
<td></td>
<td>• Data Dirty Bit fault</td>
</tr>
<tr>
<td></td>
<td>• Data Access Bit fault</td>
</tr>
<tr>
<td></td>
<td>• Data Debug fault</td>
</tr>
<tr>
<td></td>
<td>• Unaligned Data Reference fault</td>
</tr>
<tr>
<td></td>
<td>• Unsupported Data Reference fault</td>
</tr>
<tr>
<td>tlb_translate_nonaccess(vaddr, type)</td>
<td>Returns the translated data physical address for the specified virtual memory address (vaddr). type specifies the type of access (e.g., FC, TPA). If a fault is generated, tlb_translate_nonaccess does not return. The following faults are checked:</td>
</tr>
<tr>
<td></td>
<td>• Unimplemented Data Address fault</td>
</tr>
<tr>
<td></td>
<td>• Virtualization fault (tpa only)</td>
</tr>
<tr>
<td></td>
<td>• Data Nested TLB fault</td>
</tr>
<tr>
<td></td>
<td>• Alternate Data TLB fault</td>
</tr>
<tr>
<td></td>
<td>• VHPT Data fault</td>
</tr>
<tr>
<td></td>
<td>• Data TLB fault</td>
</tr>
<tr>
<td></td>
<td>• Data Page Not Present fault</td>
</tr>
<tr>
<td></td>
<td>• Data NaT Page Consumption fault</td>
</tr>
<tr>
<td></td>
<td>• Data Access Rights fault (fc only)</td>
</tr>
<tr>
<td>tlb_vhpt_hash(vrn, vaddr61, rid, size)</td>
<td>Generates a VHPT entry address for the specified virtual region number (vrn) and 61-bit virtual offset (vaddr61), region identifier (rid) and page size (size). Tlb_vhpt_hash hashes vaddr, rid and size parameters to produce a hash index. The hash index is then masked based on PTA.size and concatenated with PTA.base to generate the VHPT entry address. The long format hash is implementation specific.</td>
</tr>
<tr>
<td>tlb_vhpt_tag(vaddr, rid, size)</td>
<td>Generates a VHPT tag identifier for the specified virtual address (vaddr), region identifier (rid) and page size (size). Tlb_vhpt_tag hashes the vaddr, rid and size parameters to produce translation identifier. The tag in conjunction with the hash index is used to uniquely identify translations in the VHPT. Tag generation is implementation specific. All processor models tag function must guarantee that bit 63 of the generated tag is zero (ti bit).</td>
</tr>
<tr>
<td>undefined()</td>
<td>Returns an undefined 64-bit value.</td>
</tr>
<tr>
<td>undefined_behavior()</td>
<td>Causes undefined processor behavior. Extent of undefined behavior is described in Section 3.5, “Undefined Behavior on page 1:44.”</td>
</tr>
<tr>
<td>Function</td>
<td>Operation</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>unimplemented_physical_address(paddr)</code></td>
<td>Return TRUE if the presented physical address is unimplemented on this processor model; FALSE otherwise. This function is model specific.</td>
</tr>
<tr>
<td><code>unimplemented_virtual_address(vaddr, vm)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>vm_all_probes()</code></td>
<td>Returns TRUE if the processor is configured to virtualize all probe instructions when PSR.vm is 1. See Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344 for details.</td>
</tr>
<tr>
<td><code>vm_disabled()</code></td>
<td>Returns TRUE if the processor implements the PSR.vm bit and virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 in SDM and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:446 in SDM for details.</td>
</tr>
<tr>
<td><code>vm_select_probes()</code></td>
<td>Returns TRUE if the processor is configured to virtualize selected probe instructions when PSR.vm is 1. See Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344 for details.</td>
</tr>
<tr>
<td><code>vmsw_disabled()</code></td>
<td>Returns TRUE if the processor implements the PSR.vm bit and the vmsw instruction is disabled. See Section 3.4, “Processor Virtualization” on page 2:44 in SDM and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:446 in SDM for details.</td>
</tr>
<tr>
<td><code>zero_ext(value, pos)</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Description</th>
<th>Execution Unit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Integer ALU</td>
<td>I-unit or M-unit</td>
</tr>
<tr>
<td>I</td>
<td>Non-ALU integer</td>
<td>I-unit</td>
</tr>
<tr>
<td>M</td>
<td>Memory</td>
<td>M-unit</td>
</tr>
<tr>
<td>F</td>
<td>Floating-point</td>
<td>F-unit</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>B-unit</td>
</tr>
<tr>
<td>L+X</td>
<td>Extended</td>
<td>I-unit/B-unit*</td>
</tr>
</tbody>
</table>

* 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).
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

<table>
<thead>
<tr>
<th>Major Op (Bits 40:37)</th>
<th>I/A</th>
<th>M/A</th>
<th>F</th>
<th>B</th>
<th>L+X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Misc</td>
<td>Sys/Mem Mgmt</td>
<td>FP Misc</td>
<td>Misc/Indirect Branch</td>
<td>Misc</td>
</tr>
<tr>
<td>1</td>
<td>Sys/Mem Mgmt</td>
<td>FP Misc</td>
<td>Indirect Call</td>
<td>Indirect Call</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Indirect Predict/Nop</td>
<td>FP Class</td>
<td>IP-rel Call</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>FP Ld/St +Reg/setf</td>
<td>FP Ld/St +imm</td>
<td>IP-rel Predict</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Int Ld +Reg/setf</td>
<td>FP Compare</td>
<td>IP-relative Branch</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Int Ld/St +imm</td>
<td>FP Class</td>
<td>IP-rel Call</td>
<td>movl</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ALU/MM ALU</td>
<td>ALU/MM ALU</td>
<td>fma</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ALU/MM ALU</td>
<td>ALU/MM ALU</td>
<td>fma</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Add Imm22</td>
<td>Add Imm22</td>
<td>fma</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>A</td>
<td>fms</td>
<td>A</td>
<td>e</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>fms</td>
<td>B</td>
<td>e</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>fnma</td>
<td>C</td>
<td>e</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>Compare</td>
<td>Compare</td>
<td>fnma</td>
<td>e</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>Compare</td>
<td>Compare</td>
<td>fnma</td>
<td>e</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>fselect/xma</td>
<td>fselect/xma</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

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 Imm14 instruction (format A4) is formed from the imm7b, imm6d, 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

<table>
<thead>
<tr>
<th>Instruction Format</th>
<th>Format Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>Line Prefetch +Imm</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FP Load</td>
<td>M6</td>
<td></td>
</tr>
<tr>
<td>Int Load +Reg</td>
<td>M2</td>
<td></td>
</tr>
<tr>
<td>ALU Shift L and Add</td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>FP Store +Imm</td>
<td>M10</td>
<td></td>
</tr>
<tr>
<td>ALU Imm</td>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>FP Load Pair +Imm</td>
<td>M12</td>
<td></td>
</tr>
<tr>
<td>Add Imm</td>
<td>A4</td>
<td></td>
</tr>
<tr>
<td>FP Load +Reg</td>
<td>M7</td>
<td></td>
</tr>
<tr>
<td>Add Imm22</td>
<td>A5</td>
<td></td>
</tr>
<tr>
<td>FP Load +Imm</td>
<td>M8</td>
<td></td>
</tr>
<tr>
<td>Compare</td>
<td>A6</td>
<td></td>
</tr>
<tr>
<td>Move to Pred Immm4</td>
<td>I24</td>
<td></td>
</tr>
<tr>
<td>Compare to Zero</td>
<td>A7</td>
<td></td>
</tr>
<tr>
<td>Move to Pred Imm6</td>
<td>I25</td>
<td></td>
</tr>
<tr>
<td>Compare Imm6</td>
<td>A8</td>
<td></td>
</tr>
<tr>
<td>MM Shift and Add</td>
<td>A10</td>
<td></td>
</tr>
<tr>
<td>MM Multiply Shift</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>MM Mpy/Mix/ Pack</td>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>MM Mux1</td>
<td>I3</td>
<td></td>
</tr>
<tr>
<td>MM Mux2</td>
<td>I4</td>
<td></td>
</tr>
<tr>
<td>MM Shift R Variable</td>
<td>I5</td>
<td></td>
</tr>
<tr>
<td>MM Shift R Fixed</td>
<td>I6</td>
<td></td>
</tr>
<tr>
<td>MM Shift L Variable</td>
<td>I7</td>
<td></td>
</tr>
<tr>
<td>MM Shift L Fixed</td>
<td>I8</td>
<td></td>
</tr>
<tr>
<td>MM ALU</td>
<td>A9</td>
<td></td>
</tr>
<tr>
<td>Shift Right Pair</td>
<td>I10</td>
<td></td>
</tr>
<tr>
<td>Extract</td>
<td>I11</td>
<td></td>
</tr>
<tr>
<td>Dep.Z</td>
<td>I12</td>
<td></td>
</tr>
<tr>
<td>Dep.Z Imm6</td>
<td>I13</td>
<td></td>
</tr>
<tr>
<td>Deposit</td>
<td>I14</td>
<td></td>
</tr>
<tr>
<td>Deposit Imm</td>
<td>I15</td>
<td></td>
</tr>
<tr>
<td>Test Bit</td>
<td>I16</td>
<td></td>
</tr>
<tr>
<td>Test NaT</td>
<td>I17</td>
<td></td>
</tr>
<tr>
<td>Nop/Hint</td>
<td>I18</td>
<td></td>
</tr>
<tr>
<td>Bit Strings</td>
<td>I19</td>
<td></td>
</tr>
<tr>
<td>Int Spec Check</td>
<td>I20</td>
<td></td>
</tr>
<tr>
<td>Move to BR</td>
<td>I21</td>
<td></td>
</tr>
<tr>
<td>Move from BR</td>
<td>I22</td>
<td></td>
</tr>
<tr>
<td>Move to Pred</td>
<td>I23</td>
<td></td>
</tr>
<tr>
<td>Move to Pred Immm4</td>
<td>I24</td>
<td></td>
</tr>
<tr>
<td>Move from Pred/IP</td>
<td>I25</td>
<td></td>
</tr>
<tr>
<td>Move to AR</td>
<td>I26</td>
<td></td>
</tr>
<tr>
<td>Move to AR Imm</td>
<td>I27</td>
<td></td>
</tr>
<tr>
<td>Move from AR</td>
<td>I28</td>
<td></td>
</tr>
<tr>
<td>Sxt/Zxt/Czx</td>
<td>I29</td>
<td></td>
</tr>
<tr>
<td>Test Feature</td>
<td>I30</td>
<td></td>
</tr>
<tr>
<td>Int Load</td>
<td>M1</td>
<td></td>
</tr>
<tr>
<td>Int Load +Reg</td>
<td>M2</td>
<td></td>
</tr>
<tr>
<td>Int Load +Imm</td>
<td>M3</td>
<td></td>
</tr>
<tr>
<td>Int Store</td>
<td>M4</td>
<td></td>
</tr>
<tr>
<td>Int Store +Imm</td>
<td>M5</td>
<td></td>
</tr>
<tr>
<td>FP Load</td>
<td>M6</td>
<td></td>
</tr>
<tr>
<td>FP Load +Reg</td>
<td>M7</td>
<td></td>
</tr>
<tr>
<td>FP Load +Imm</td>
<td>M8</td>
<td></td>
</tr>
<tr>
<td>FP Store</td>
<td>M9</td>
<td></td>
</tr>
<tr>
<td>FP Store +Imm</td>
<td>M10</td>
<td></td>
</tr>
<tr>
<td>FP Load Pair +Imm</td>
<td>M12</td>
<td></td>
</tr>
<tr>
<td>Line Prefetch +Imm</td>
<td>M13</td>
<td></td>
</tr>
<tr>
<td>Line Prefetch +Reg</td>
<td>M14</td>
<td></td>
</tr>
<tr>
<td>(Cmp &amp; Exchg)</td>
<td>M16</td>
<td></td>
</tr>
<tr>
<td>Fetch &amp; Add</td>
<td>M17</td>
<td></td>
</tr>
<tr>
<td>Set FR</td>
<td>M18</td>
<td></td>
</tr>
<tr>
<td>Get FR</td>
<td>M19</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-4. Instruction Format Summary

40393837 363534 333231 302928 272625 2423 2221 2019 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)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int Spec Check</td>
<td>M20</td>
<td></td>
</tr>
<tr>
<td>FP Spec Check</td>
<td>M21</td>
<td></td>
</tr>
<tr>
<td>Int ALAT Check</td>
<td>M22</td>
<td></td>
</tr>
<tr>
<td>FP ALAT Check</td>
<td>M23</td>
<td></td>
</tr>
<tr>
<td>Sync/Srlz/ALAT</td>
<td>M24</td>
<td></td>
</tr>
<tr>
<td>RSE Control</td>
<td>M25</td>
<td></td>
</tr>
<tr>
<td>Int ALAT Inval</td>
<td>M26</td>
<td></td>
</tr>
<tr>
<td>FP ALAT Inval</td>
<td>M27</td>
<td></td>
</tr>
<tr>
<td>Flush Cache</td>
<td>M28</td>
<td></td>
</tr>
<tr>
<td>Move to AR</td>
<td>M29</td>
<td></td>
</tr>
<tr>
<td>Move to AR Imm8</td>
<td>M30</td>
<td></td>
</tr>
<tr>
<td>Move from AR</td>
<td>M31</td>
<td></td>
</tr>
<tr>
<td>Move to CR</td>
<td>M32</td>
<td></td>
</tr>
<tr>
<td>Move from CR</td>
<td>M33</td>
<td></td>
</tr>
<tr>
<td>Alloc</td>
<td>M34</td>
<td></td>
</tr>
<tr>
<td>Move to PSR</td>
<td>M35</td>
<td></td>
</tr>
<tr>
<td>Move from PSR</td>
<td>M36</td>
<td></td>
</tr>
<tr>
<td>Break</td>
<td>M37</td>
<td></td>
</tr>
<tr>
<td>Probe</td>
<td>M38</td>
<td></td>
</tr>
<tr>
<td>Probe Fault Imm2</td>
<td>M39</td>
<td></td>
</tr>
<tr>
<td>TC Insert</td>
<td>M41</td>
<td></td>
</tr>
<tr>
<td>Mv to Ind/TR Ins</td>
<td>M42</td>
<td></td>
</tr>
<tr>
<td>Mv from Ind</td>
<td>M43</td>
<td></td>
</tr>
<tr>
<td>Set/Reset Mask</td>
<td>M44</td>
<td></td>
</tr>
<tr>
<td>Translation Purge</td>
<td>M45</td>
<td></td>
</tr>
<tr>
<td>Translation Access</td>
<td>M46</td>
<td></td>
</tr>
<tr>
<td>TC Entry Purge</td>
<td>M47</td>
<td></td>
</tr>
<tr>
<td>Nop/Hint</td>
<td>M48</td>
<td></td>
</tr>
<tr>
<td>IP-Relative Branch</td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>Counted Branch</td>
<td>B2</td>
<td></td>
</tr>
<tr>
<td>IP-Relative Call</td>
<td>B3</td>
<td></td>
</tr>
<tr>
<td>Indirect Branch</td>
<td>B4</td>
<td></td>
</tr>
<tr>
<td>Indirect Call</td>
<td>B5</td>
<td></td>
</tr>
<tr>
<td>IP-Relative Predict</td>
<td>B6</td>
<td></td>
</tr>
<tr>
<td>Indirect Predict</td>
<td>B7</td>
<td></td>
</tr>
<tr>
<td>Misc</td>
<td>B8</td>
<td></td>
</tr>
<tr>
<td>Break/Nop/Hint</td>
<td>B9</td>
<td></td>
</tr>
<tr>
<td>FP Arithmetic</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td>Fixed Multiply Add</td>
<td>F2</td>
<td></td>
</tr>
<tr>
<td>FP Select</td>
<td>F3</td>
<td></td>
</tr>
<tr>
<td>FP Compare</td>
<td>F4</td>
<td></td>
</tr>
<tr>
<td>FP Recip Approx</td>
<td>F5</td>
<td></td>
</tr>
<tr>
<td>FP Recip Sqrt App</td>
<td>F6</td>
<td></td>
</tr>
<tr>
<td>FP Min/Max/Pcmp</td>
<td>F7</td>
<td></td>
</tr>
<tr>
<td>FP Merge/Logical</td>
<td>F8</td>
<td></td>
</tr>
<tr>
<td>Convert FP to Fixed</td>
<td>F9</td>
<td></td>
</tr>
<tr>
<td>Convert Fixed to FP</td>
<td>F10</td>
<td></td>
</tr>
<tr>
<td>FP Set Controls</td>
<td>F11</td>
<td></td>
</tr>
<tr>
<td>FP Clear Flags</td>
<td>F12</td>
<td></td>
</tr>
<tr>
<td>FP Check Flags</td>
<td>F13</td>
<td></td>
</tr>
<tr>
<td>Break</td>
<td>F14</td>
<td></td>
</tr>
<tr>
<td>Nop/Hint</td>
<td>F15</td>
<td></td>
</tr>
<tr>
<td>Move Imm864</td>
<td>X2</td>
<td></td>
</tr>
<tr>
<td>Long Branch</td>
<td>X3</td>
<td></td>
</tr>
<tr>
<td>Long Call</td>
<td>X4</td>
<td></td>
</tr>
<tr>
<td>Nop/Hint</td>
<td>X5</td>
<td></td>
</tr>
</tbody>
</table>

Volume 3: Instruction Formats
### Table 4-5. Instruction Field Color Key

<table>
<thead>
<tr>
<th>Field &amp; Color</th>
<th>ALU Instruction</th>
<th>Integer Instruction</th>
<th>Memory Instruction</th>
<th>Branch Instruction</th>
<th>Floating-point Instruction</th>
<th>Integer Source</th>
<th>Memory Source</th>
<th>Shift Source</th>
<th>Special Register Source</th>
<th>Memory Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode Extension</td>
<td>Opcode Extension</td>
<td>Opcode Hint Extension</td>
<td>Immediate</td>
<td>Indirect Source</td>
<td>Predicate Destination</td>
<td>Integer Destination</td>
<td>Memory Source &amp; Destination</td>
<td>Shift Immediate</td>
<td>Special Register Destination</td>
<td>Floating-point Destination</td>
</tr>
</tbody>
</table>

### Table 4-6. Instruction Field Names

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>af₃</td>
<td>application register source/target</td>
</tr>
<tr>
<td>b1, b2</td>
<td>branch register source/target</td>
</tr>
<tr>
<td>btype</td>
<td>branch type opcode extension</td>
</tr>
<tr>
<td>c</td>
<td>complement compare relation opcode extension</td>
</tr>
<tr>
<td>ccount₆c</td>
<td>multimedia shift left complemented shift count immediate</td>
</tr>
<tr>
<td>count₅b, count₆d</td>
<td>multimedia shift right/shift right pair shift count immediate</td>
</tr>
<tr>
<td>cposₓ</td>
<td>deposit complemented bit position immediate</td>
</tr>
<tr>
<td>cr₃</td>
<td>control register source/target</td>
</tr>
<tr>
<td>ct₂d</td>
<td>multimedia multiply shift/shift and add shift count immediate</td>
</tr>
<tr>
<td>d</td>
<td>branch cache deallocation hint opcode extension</td>
</tr>
<tr>
<td>fₙ</td>
<td>floating-point register source/target</td>
</tr>
<tr>
<td>fc₂, fclass₇c</td>
<td>floating-point class immediate</td>
</tr>
<tr>
<td>hint</td>
<td>memory reference hint opcode extension</td>
</tr>
<tr>
<td>ih</td>
<td>branch importance hint opcode extension</td>
</tr>
<tr>
<td>len₄d, len₆d</td>
<td>extract/deposit length immediate</td>
</tr>
<tr>
<td>m</td>
<td>memory reference post-modify opcode extension</td>
</tr>
<tr>
<td>maskₓ</td>
<td>predicate immediate mask</td>
</tr>
<tr>
<td>mbₓ₉c, mhf₈c</td>
<td>multimedia mux1/mux2 immediate</td>
</tr>
<tr>
<td>p</td>
<td>sequential prefetch hint opcode extension</td>
</tr>
<tr>
<td>p₁, p₂</td>
<td>predicate register target</td>
</tr>
<tr>
<td>pos₈b</td>
<td>test bit/extract bit position immediate</td>
</tr>
<tr>
<td>q</td>
<td>floating-point reciprocal/reciprocal square-root opcode extension</td>
</tr>
<tr>
<td>qp</td>
<td>qualifying predicate register source</td>
</tr>
<tr>
<td>rₙ</td>
<td>general register source/target</td>
</tr>
<tr>
<td>s</td>
<td>immediate sign bit</td>
</tr>
<tr>
<td>sf</td>
<td>floating-point status field opcode extension</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_{2a}$ Bits 35:34</th>
<th>$v_e$ Bit 33</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>adds – imm14 A4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addp4 – imm14 A4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4-8. Integer ALU 2-bit+1-bit Opcode Extensions

---

3:300 Volume 3: Instruction Formats
### 4.2.1.1 Integer ALU – Register-Register

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_{2a}$ Bits 35:34</th>
<th>$v_e$ Bit 33</th>
<th>$x_4$ Bits 32:29</th>
<th>$x_{2b}$ Bits 28:27</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

#### Instruction Operands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$r_1 = r_2, r_3$</td>
<td>8</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>sub</td>
<td>$r_1 = r_2, r_3$</td>
<td>8</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>addp4</td>
<td>$r_1 = r_2, r_3$</td>
<td>8</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>and</td>
<td>$r_1 = r_2, r_3$</td>
<td>8</td>
<td>0 0 1 3</td>
</tr>
<tr>
<td>or</td>
<td>$r_1 = r_2, r_3$</td>
<td>8</td>
<td>0 0 0 3</td>
</tr>
<tr>
<td>xor</td>
<td>$r_1 = r_2, r_3$</td>
<td>8</td>
<td>0 0 1 3</td>
</tr>
</tbody>
</table>

### 4.2.1.2 Shift Left and Add

#### Instruction Operands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>shladd</td>
<td>$r_1 = r_2, count_2, r_3$</td>
<td>8</td>
<td>0 0 4 6</td>
</tr>
<tr>
<td>shladdp4</td>
<td>$r_1 = r_2, count_2, r_3$</td>
<td>8</td>
<td>0 0 4 6</td>
</tr>
</tbody>
</table>
4.2.1.3 Integer ALU – Immediate8-Register

To accommodate immediate 8-bit values, the integer ALU instructions use an 8-bit immediate operand field in bits 31:24 of the instruction format, as shown in Table 4-11.

### Instruction Operands Opcode Extension

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td></td>
<td>8</td>
<td>0 9 1</td>
</tr>
<tr>
<td>and</td>
<td></td>
<td>8</td>
<td>0 0 B</td>
</tr>
<tr>
<td>andcm</td>
<td>( r_1 = \text{imm}_8, \ r_3 )</td>
<td>8</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td>8</td>
<td>0 0 B</td>
</tr>
<tr>
<td>xor</td>
<td></td>
<td>8</td>
<td>0 0 B</td>
</tr>
</tbody>
</table>

4.2.1.4 Add Immediate14

The add immediate14 instructions use an 14-bit immediate operand field in bits 35:22 of the instruction format, as shown in Table 4-11.

### Instruction Operands Opcode Extension

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>adds</td>
<td>( r_1 = \text{imm}_{14}, \ r_3 )</td>
<td>8</td>
<td>2 3 0</td>
</tr>
<tr>
<td>addp4</td>
<td></td>
<td>8</td>
<td>0 0 B</td>
</tr>
</tbody>
</table>

4.2.1.5 Add Immediate22

The add immediate22 instructions use an 22-bit immediate operand field in bits 35:13 of the instruction format, as shown in Table 4-11.

### Instruction Operands Opcode Extension

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>( r_1 = \text{imm}_{22}, \ r_3 )</td>
<td>9</td>
</tr>
</tbody>
</table>

4.2.2 Integer Compare

The integer compare instructions are encoded within major opcodes C - E using a 2-bit opcode extension field \( (x_2) \) in bits 35:34 and three 1-bit opcode extension fields in bits 33 (\( t_s \)), 36 (\( t_b \)), and 12 (\( c \)), as shown in Table 4-10. The integer compare immediate instructions are encoded within major opcodes C - E using a 2-bit opcode extension field \( (x_2) \) in bits 35:34 and two 1-bit opcode extension fields in bits 33 (\( t_a \)) and 12 (\( c \)), as shown in Table 4-11.
### Table 4-10. Integer Compare Opcode Extensions

<table>
<thead>
<tr>
<th>$x_2$ Bits 35:34</th>
<th>$t_3$ Bit 36</th>
<th>$t_2$ Bit 33</th>
<th>$c_3$ Bit 12</th>
<th>Opcode Bits 40:37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>cmp.lt A6</td>
<td>cmp.ltu A6</td>
<td>cmp.eq A6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp.lt.unc A6</td>
<td>cmp.ltu.unc A6</td>
<td>cmp.eq.unc A6</td>
</tr>
<tr>
<td>0 0 1</td>
<td>cmp.eq.and A6</td>
<td>cmp.eq.or A6</td>
<td>cmp.eq.or.andcm A6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp.ne.and A6</td>
<td>cmp.ne.or A6</td>
<td>cmp.ne.or.andcm A6</td>
</tr>
<tr>
<td>0 1 0</td>
<td>cmp.gt.and A7</td>
<td>cmp.gt.or A7</td>
<td>cmp.gt.or.andcm A7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp.le.and A7</td>
<td>cmp.le.or A7</td>
<td>cmp.le.or.andcm A7</td>
</tr>
<tr>
<td>0 1 1</td>
<td>cmp.ge.and A7</td>
<td>cmp.ge.or A7</td>
<td>cmp.ge.or.andcm A7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp.lt.and A7</td>
<td>cmp.lt.or A7</td>
<td>cmp.lt.or.andcm A7</td>
</tr>
<tr>
<td>1 0 0</td>
<td>cmp4.lt A6</td>
<td>cmp4.ltu A6</td>
<td>cmp4.eq A6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp4.lt.unc A6</td>
<td>cmp4.ltu.unc A6</td>
<td>cmp4.eq.unc A6</td>
</tr>
<tr>
<td>1 0 1</td>
<td>cmp4.eq.and A6</td>
<td>cmp4.eq.or A6</td>
<td>cmp4.eq.or.andcm A6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp4.ne.and A6</td>
<td>cmp4.ne.or A6</td>
<td>cmp4.ne.or.andcm A6</td>
</tr>
<tr>
<td>1 1 0</td>
<td>cmp4.gt.and A7</td>
<td>cmp4.gt.or A7</td>
<td>cmp4.gt.or.andcm A7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp4.le.and A7</td>
<td>cmp4.le.or A7</td>
<td>cmp4.le.or.andcm A7</td>
</tr>
<tr>
<td>1 1 1</td>
<td>cmp4.ge.and A7</td>
<td>cmp4.ge.or A7</td>
<td>cmp4.ge.or.andcm A7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp4.lt.and A7</td>
<td>cmp4.lt.or A7</td>
<td>cmp4.lt.or.andcm A7</td>
</tr>
</tbody>
</table>

### Table 4-11. Integer Compare Immediate Opcode Extensions

<table>
<thead>
<tr>
<th>$x_2$ Bits 35:34</th>
<th>$t_3$ Bit 33</th>
<th>$c_3$ Bit 12</th>
<th>Opcode Bits 40:37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>2 0 0</td>
<td>cmp.lt – immg A8</td>
<td>cmp.ltu – immg A8</td>
<td>cmp.eq – immg A8</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp.lt.unc – immg A8</td>
<td>cmp.ltu.unc – immg A8</td>
</tr>
<tr>
<td>2 0 1</td>
<td>cmp.eq.and – immg A8</td>
<td>cmp.eq.or – immg A8</td>
<td>cmp.eq.or.andcm – immg A8</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp.ne.and – immg A8</td>
<td>cmp.ne.or – immg A8</td>
</tr>
<tr>
<td>3 0 0</td>
<td>cmp4.lt – immg A8</td>
<td>cmp4.ltu – immg A8</td>
<td>cmp4.eq – immg A8</td>
</tr>
<tr>
<td>3 0 1</td>
<td>cmp4.eq.and – immg A8</td>
<td>cmp4.eq.or – immg A8</td>
<td>cmp4.eq.or.andcm – immg A8</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>cmp4.ne.and – immg A8</td>
<td>cmp4.ne.or – immg A8</td>
</tr>
</tbody>
</table>
### 4.2.2.1 Integer Compare – Register-Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp.lt</td>
<td></td>
<td>C</td>
<td>( x_2 )</td>
</tr>
<tr>
<td>cmp.ltu</td>
<td></td>
<td>D</td>
<td>( t_b )</td>
</tr>
<tr>
<td>cmp.eq</td>
<td></td>
<td>E</td>
<td>( t_a )</td>
</tr>
<tr>
<td>cmp.lt.unc</td>
<td></td>
<td>C</td>
<td>( c )</td>
</tr>
<tr>
<td>cmp.ltu.unc</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>cmp.eq.unc</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp.eq.and</td>
<td></td>
<td>C</td>
<td>( 0 )</td>
</tr>
<tr>
<td>cmp.eq.or</td>
<td></td>
<td>D</td>
<td>( 0 )</td>
</tr>
<tr>
<td>cmp.eq.or.andcm</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp.ne.and</td>
<td></td>
<td>C</td>
<td>( 1 )</td>
</tr>
<tr>
<td>cmp.ne.or</td>
<td></td>
<td>D</td>
<td>( 1 )</td>
</tr>
<tr>
<td>cmp.ne.or.andcm</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp4.lt</td>
<td></td>
<td>C</td>
<td>( 0 )</td>
</tr>
<tr>
<td>cmp4.ltu</td>
<td></td>
<td>D</td>
<td>( 0 )</td>
</tr>
<tr>
<td>cmp4.eq</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp4.lt.unc</td>
<td></td>
<td>C</td>
<td>( 0 )</td>
</tr>
<tr>
<td>cmp4.ltu.unc</td>
<td></td>
<td>D</td>
<td>( 1 )</td>
</tr>
<tr>
<td>cmp4.eq.unc</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp4.eq.and</td>
<td></td>
<td>C</td>
<td>( 0 )</td>
</tr>
<tr>
<td>cmp4.eq.or</td>
<td></td>
<td>D</td>
<td>( 0 )</td>
</tr>
<tr>
<td>cmp4.eq.or.andcm</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp4.ne.and</td>
<td></td>
<td>C</td>
<td>( 1 )</td>
</tr>
<tr>
<td>cmp4.ne.or</td>
<td></td>
<td>D</td>
<td>( 1 )</td>
</tr>
<tr>
<td>cmp4.ne.or.andcm</td>
<td></td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

Where \( p_1, p_2 = r_2, r_3 \)
### 4.2.2.2 Integer Compare to Zero – Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp.gt.and</td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>cmp.gt.or</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>cmp.gt.or.andcm</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp.le.and</td>
<td></td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>cmp.le.or</td>
<td></td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>cmp.le.or.andcm</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp.ge.and</td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>cmp.ge.or</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>cmp.ge.or.andcm</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp.lt.and</td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>cmp.lt.or</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>cmp.lt.or.andcm</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp4.gt.and</td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>cmp4.gt.or</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>cmp4.gt.or.andcm</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>cmp4.ge.and</td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>cmp4.ge.or</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>cmp4.ge.or.andcm</td>
<td></td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

\[ p_1, p_2 = r0, r3 \]
4.2.2.3 Integer Compare – Immediate-Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp.lt</td>
<td></td>
<td>C</td>
<td>x2</td>
</tr>
<tr>
<td>cmp.lt.u</td>
<td></td>
<td>D</td>
<td>t_a</td>
</tr>
<tr>
<td>cmp.eq</td>
<td></td>
<td>E</td>
<td>c</td>
</tr>
<tr>
<td>cmp.lt.unc</td>
<td></td>
<td>C</td>
<td>x2</td>
</tr>
<tr>
<td>cmp.lt.u.unc</td>
<td></td>
<td>D</td>
<td>t_a</td>
</tr>
<tr>
<td>cmp.eq.unc</td>
<td></td>
<td>E</td>
<td>c</td>
</tr>
<tr>
<td>cmp.eq.and</td>
<td></td>
<td>C</td>
<td>x2</td>
</tr>
<tr>
<td>cmp.eq.or</td>
<td></td>
<td>D</td>
<td>t_a</td>
</tr>
<tr>
<td>cmp.eq.or.andcm</td>
<td></td>
<td>E</td>
<td>c</td>
</tr>
<tr>
<td>cmp.ne.and</td>
<td></td>
<td>C</td>
<td>x2</td>
</tr>
<tr>
<td>cmp.ne.or</td>
<td></td>
<td>D</td>
<td>t_a</td>
</tr>
<tr>
<td>cmp.ne.or.andcm</td>
<td></td>
<td>E</td>
<td>c</td>
</tr>
<tr>
<td>cmp4.lt</td>
<td></td>
<td>C</td>
<td>x2</td>
</tr>
<tr>
<td>cmp4.lt.u</td>
<td></td>
<td>D</td>
<td>t_a</td>
</tr>
<tr>
<td>cmp4.eq</td>
<td></td>
<td>E</td>
<td>c</td>
</tr>
<tr>
<td>cmp4.lt.unc</td>
<td></td>
<td>C</td>
<td>x2</td>
</tr>
<tr>
<td>cmp4.lt.u.unc</td>
<td></td>
<td>D</td>
<td>t_a</td>
</tr>
<tr>
<td>cmp4.eq.unc</td>
<td></td>
<td>E</td>
<td>c</td>
</tr>
<tr>
<td>cmp4.eq.and</td>
<td></td>
<td>C</td>
<td>x2</td>
</tr>
<tr>
<td>cmp4.eq.or</td>
<td></td>
<td>D</td>
<td>t_a</td>
</tr>
<tr>
<td>cmp4.eq.or.andcm</td>
<td></td>
<td>E</td>
<td>c</td>
</tr>
<tr>
<td>cmp4.ne.and</td>
<td></td>
<td>C</td>
<td>x2</td>
</tr>
<tr>
<td>cmp4.ne.or</td>
<td></td>
<td>D</td>
<td>t_a</td>
</tr>
<tr>
<td>cmp4.ne.or.andcm</td>
<td></td>
<td>E</td>
<td>c</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x_2a Bits 35:34</th>
<th>z_a Bit 36</th>
<th>z_b Bit 33</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Volume 3: Instruction Formats
### Table 4-13. Multimedia ALU Size 1 4-bit+2-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>( x_{2a} ) Bits 35:34</th>
<th>( z_a ) Bit 36</th>
<th>( z_b ) Bit 33</th>
<th>( x_4 ) Bits 32:29</th>
<th>( x_{2b} ) Bits 28:27</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>padd1 ( A9 )</td>
<td>padd1.sss ( A9 )</td>
<td>padd1.uuu ( A9 )</td>
<td>padd1.uus ( A9 )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>psub1 ( A9 )</td>
<td>psub1.sss ( A9 )</td>
<td>psub1.uuu ( A9 )</td>
<td>psub1.uus ( A9 )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>pavg1 ( A9 )</td>
<td>pavg1.raz ( A9 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>pavgsub1 ( A9 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>pcmp1.eq ( A9 )</td>
<td>pcmp1.gt ( A9 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>( x_{2a} ) Bits 35:34</th>
<th>( z_a ) Bit 36</th>
<th>( z_b ) Bit 33</th>
<th>( x_4 ) Bits 32:29</th>
<th>( x_{2b} ) Bits 28:27</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>padd2 ( A9 )</td>
<td>padd2.sss ( A9 )</td>
<td>padd2.uuu ( A9 )</td>
<td>padd2.uus ( A9 )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>psub2 ( A9 )</td>
<td>psub2.sss ( A9 )</td>
<td>psub2.uuu ( A9 )</td>
<td>psub2.uus ( A9 )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>pavg2 ( A9 )</td>
<td>pavg2.raz ( A9 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>pavgsub2 ( A9 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>pshladd2 ( A10 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>pshradd2 ( A10 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>pcmp2.eq ( A9 )</td>
<td>pcmp2.gt ( A9 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

*Volume 3: Instruction Formats*
Table 4-15. Multimedia ALU Size 4 4-bit+2-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_{2a}$ Bits 35:34</th>
<th>$z_a$ Bit 36</th>
<th>$z_b$ Bit 33</th>
<th>$x_{4}$ Bits 32:29</th>
<th>$x_{2b}$ Bits 28:27</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>padd4 A9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>psub4 A9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>pcmp4.eq A9</td>
<td>pcmp4.gt A9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 4.2.3.1 Multimedia ALU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>padd1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>padd2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>padd4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>padd1.sss</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>padd2.sss</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>padd1.uuu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>padd2.uuu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>padd1.uus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>padd2.uus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psub1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psub2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psub4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psub1.sss</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psub2.sss</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psub1.uuu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psub2.uuu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psub1.uus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psub2.uus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pavg1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pavg2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pavg1.raz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pavg2.raz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pavgsub1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pavgsub2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pcmp1.eq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pcmp2.eq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pcmp4.eq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pcmp1.gt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pcmp2.gt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pcmp4.gt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example:**
- padd1: $r_1 = r_2 - r_3$

### 4.2.3.2 Multimedia Shift and Add

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>pshladd2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pshradd2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example:**
- pshladd2: $r_1 = r_2$, count, $r_3$
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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>z_a Bit 36</th>
<th>z_b Bit 33</th>
<th>v_e Bit 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Multimedia Size 1 (Table 4-17)

Multimedia Size 2 (Table 4-18)

Multimedia Size 4 (Table 4-19)

Variable Shift (Table 4-20)

Table 4-17. Multimedia Opcode 7 Size 1 2-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>z_a Bit 36</th>
<th>z_b Bit 33</th>
<th>v_e Bit 32</th>
<th>x_2a Bits 35:34</th>
<th>x_2b Bits 29:28</th>
<th>x_2c Bits 31:30</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

unpack1.h l2
mix1.r l2
pmin1.u l2
pmax1.u l2
unpack1.l l2
mix1.l l2
psad1 l2
mux1 l3
### Table 4-18. Multimedia Opcode 7 Size 2 2-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits</th>
<th>$z_a$ Bit</th>
<th>$z_b$ Bit</th>
<th>$v_a$ Bit</th>
<th>$x_{2a}$ Bits</th>
<th>$x_{2b}$ Bits</th>
<th>$x_{2c}$ Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pshr2.u – var I5</td>
<td>pshl2 – var I7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>mpyshr2.u I1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>pshr2 – var I5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>mpyshr2 I1</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits</th>
<th>$z_a$ Bit</th>
<th>$z_b$ Bit</th>
<th>$v_a$ Bit</th>
<th>$x_{2a}$ Bits</th>
<th>$x_{2b}$ Bits</th>
<th>$x_{2c}$ Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pshr4.u – var I5</td>
<td>pshl4 – var I7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>mpy4 I2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>pshr4 – var I5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>mpyshr4 I2</td>
<td></td>
</tr>
</tbody>
</table>

Volume 3: Instruction Formats
Table 4-20. Variable Shift Opcode 7 2-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits</th>
<th>za Bit 36</th>
<th>zb Bit 33</th>
<th>ve Bit 32</th>
<th>x2a Bits 35:34</th>
<th>x2b Bits 29:28</th>
<th>x2c Bits 31:30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3.1.1 Multimedia Multiply and Shift

Instruction Operands Opcode Extension

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>za</th>
<th>zb</th>
<th>ve</th>
<th>x2a</th>
<th>x2b</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmpyshr2</td>
<td>r1 = r2, r3, count2</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>pmpyshr2.u</td>
<td></td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Volume 3: Instruction Formats
### 4.3.1.2 Multimedia Multiply/Mix/Pack/Unpack

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpy4</td>
<td></td>
<td></td>
<td>za zb ve x2a x2b x2c</td>
</tr>
<tr>
<td>mpyshl4</td>
<td></td>
<td></td>
<td>za zb ve x2a x2b x2c</td>
</tr>
<tr>
<td>mpy2.r</td>
<td>0 1</td>
<td></td>
<td>1 1 3 3</td>
</tr>
<tr>
<td>mpy2.l</td>
<td>0 1</td>
<td></td>
<td>1 1 3 3</td>
</tr>
<tr>
<td>mix1.r</td>
<td>0 0</td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>mix2.r</td>
<td>0 1</td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>mix4.r</td>
<td>1 0</td>
<td></td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>mix1.l</td>
<td>0 0</td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>mix2.l</td>
<td>0 1</td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>mix4.l</td>
<td>1 0</td>
<td></td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>pack2.uss</td>
<td></td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>pack2.sss</td>
<td></td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>pack4.sss</td>
<td></td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>unpack1.h</td>
<td>0 0</td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>unpack2.h</td>
<td>0 1</td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>unpack4.h</td>
<td>1 0</td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>unpack1.l</td>
<td>0 0</td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>unpack2.l</td>
<td>0 1</td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>unpack4.l</td>
<td>1 0</td>
<td>7</td>
<td>0 0 0 2</td>
</tr>
<tr>
<td>pmin1.u</td>
<td>0 0</td>
<td>7</td>
<td>0 0 1 3</td>
</tr>
<tr>
<td>pmax1.u</td>
<td></td>
<td>7</td>
<td>0 0 1 3</td>
</tr>
<tr>
<td>pmin2</td>
<td>0 1</td>
<td></td>
<td>0 0 3 1</td>
</tr>
<tr>
<td>pmax2</td>
<td></td>
<td>7</td>
<td>0 0 3 1</td>
</tr>
<tr>
<td>psad1</td>
<td>0 0</td>
<td>7</td>
<td>0 0 3 1</td>
</tr>
</tbody>
</table>

### 4.3.1.3 Multimedia Mux1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mux1</td>
<td>r1 = r2, mbtype4</td>
<td>7</td>
<td>za zb ve x2a x2b x2c</td>
</tr>
</tbody>
</table>

### 4.3.1.4 Multimedia Mux2

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mux2</td>
<td>r1 = r2, mbtype8</td>
<td>7</td>
<td>za zb ve x2a x2b x2c</td>
</tr>
</tbody>
</table>
### 4.3.1.5 Shift Right – Variable

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>pshr2</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>pshr4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shr</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>pshr2.u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pshr4.u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shr.u</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.3.1.6 Multimedia Shift Right – Fixed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>pshr2</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>pshr4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shr</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>shru</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pshr2.u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pshr4.u</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.3.1.7 Shift Left – Variable

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>pshl2</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>pshl4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shl</td>
<td></td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

### 4.3.1.8 Multimedia Shift Left – Fixed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>pshl2</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>pshl4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Volume 3: Instruction Formats
4.3.1.9 Bit Strings

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_2$ Bits 35:34</th>
<th>$x$ Bit 33</th>
<th>$y$ Bit 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_2$ Bits 35:34</th>
<th>$x$ Bit 33</th>
<th>$y$ Bit 26</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_2$ Bits 35:34</th>
<th>$x$ Bit 33</th>
<th>$y$ Bit 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_2$ Bits 35:34</th>
<th>$x$ Bit 33</th>
<th>$y$ Bit 26</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3.2.1 Shift Right Pair

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_2$ Bits 35:34</th>
<th>$x$ Bit 33</th>
<th>$y$ Bit 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_2$ Bits 35:34</th>
<th>$x$ Bit 33</th>
<th>$y$ Bit 26</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3.2.1 Shift Right Pair

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_2$ Bits 35:34</th>
<th>$x$ Bit 33</th>
<th>$y$ Bit 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_2$ Bits 35:34</th>
<th>$x$ Bit 33</th>
<th>$y$ Bit 26</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 4.3.2.2 Extract

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>extr.u</td>
<td>r₁ = r₃, pos₆, len₆</td>
<td>5</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>extr</td>
<td>r₁ = r₃, pos₆, len₆</td>
<td>5</td>
<td>1 0 1 0</td>
</tr>
</tbody>
</table>

### 4.3.2.3 Zero and Deposit

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep.z</td>
<td>r₁ = r₂, pos₆, len₆</td>
<td>5</td>
<td>1 1 0 0</td>
</tr>
</tbody>
</table>

### 4.3.2.4 Zero and Deposit Immediate₈

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep.z</td>
<td>r₁ = imm₈, pos₆, len₆</td>
<td>5</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

### 4.3.2.5 Deposit Immediate₁₁

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep</td>
<td>r₁ = imm₁₁, r₃, pos₆, len₆</td>
<td>5</td>
<td>3 1</td>
</tr>
</tbody>
</table>

### 4.3.2.6 Deposit

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep</td>
<td>r₁ = r₂, r₃, pos₆, len₄</td>
<td>4</td>
</tr>
</tbody>
</table>

### 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 (x₂) plus five 1-bit opcode extension fields in bits 33 (t₃), 36 (t₆), 12 (c), 13 (y) and 19 (x). Table 4-23 summarizes these assignments.
### Table 4-23.  Test Bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>$x_2$ Bits 35:34</th>
<th>$t_a$ Bit 33</th>
<th>$t_b$ Bit 36</th>
<th>$c$ Bit 12</th>
<th>$y$ Bit 13</th>
<th>$x$ Bit 19</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### 4.3.3.1 Test Bit

<table>
<thead>
<tr>
<th>Instruction Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>tbit.z</td>
<td>5</td>
<td>$x_2$ 0  $t_a$ 0 $t_b$ 0 $y$ 0 $c$ 1</td>
</tr>
<tr>
<td>tbit.z.unc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tbit.z.and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tbit.nz.and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tbit.z.or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tbit.nz.or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tbit.z.or.andcm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tbit.nz.or.andcm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$p_1, p_2 = r_3, \text{pos}_6$
4.3.3.2 Test Nat

The miscellaneous I-unit instructions are encoded in major opcode 0 using a 3-bit opcode extension field (x₃) in bits 35:33. Some also have a 6-bit opcode extension field (x₆) 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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>tnat.z</td>
<td></td>
<td>5</td>
<td>x₂ tₐ tₐ b y c x</td>
</tr>
<tr>
<td>tnat.z.unc</td>
<td></td>
<td>5</td>
<td>0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>tnat.z.and</td>
<td>p₁, p₂ = r₃</td>
<td>5</td>
<td>1 0 1 0 0 1 1</td>
</tr>
<tr>
<td>tnat.nz.and</td>
<td></td>
<td>5</td>
<td>1 0 0 1 0 0 1</td>
</tr>
<tr>
<td>tnat.z.or</td>
<td></td>
<td>5</td>
<td>1 0 1 1 0 0 1</td>
</tr>
<tr>
<td>tnat.nz.or</td>
<td></td>
<td>5</td>
<td>1 0 1 1 0 0 1</td>
</tr>
<tr>
<td>tnat.z.or.andcm</td>
<td></td>
<td>5</td>
<td>1 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>tnat.z.or.andcm</td>
<td></td>
<td>5</td>
<td>0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

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 (x₃) in bits 35:33. Some also have a 6-bit opcode extension field (x₆) in bits 32:27. **Table 4-24** shows the 3-bit assignments and **Table 4-25** summarizes the 6-bit assignments.

**Table 4-25. Misc I-Unit 6-bit Opcode Extensions**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bits 40:37</th>
<th>x₃</th>
<th>Bits 35:33</th>
<th>6-bit Ext (Table 4-25)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>chk.s.i – int i20</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td>mov to pr rot – imm 44 i24</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
<td>mov to pr i23</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2</td>
<td></td>
<td>mov to pr i23</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
<td>mov to b i21</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 4-25. Misc I-Unit 6-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>( x_3 )</th>
<th>( x_6 )</th>
<th>( x_6 )</th>
<th>Bits 30:27</th>
<th>Bits 32:31</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_3 )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>break.i</td>
<td>119</td>
<td>zxt1</td>
<td>l29</td>
<td>mov from ip</td>
</tr>
<tr>
<td>1</td>
<td>1-bit Ext (Table 4-26)</td>
<td>zxt2</td>
<td>l29</td>
<td>mov from b</td>
<td>l22</td>
</tr>
<tr>
<td>2</td>
<td>zxt4</td>
<td>l29</td>
<td>mov.i from ar</td>
<td>l28</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>mov from pr</td>
<td>l25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>sxt1</td>
<td>l29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>sxt2</td>
<td>l29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>sxt4</td>
<td>l29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>czx1.i</td>
<td>l29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>czx2.i</td>
<td>l29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>mov.i to ar – imm8</td>
<td>l27</td>
<td>mov.i to ar</td>
<td>l26</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>czx1.r</td>
<td>l29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>czx2.r</td>
<td>l29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>mov.i to ar – imm8</td>
<td>l27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>( x_3 )</th>
<th>( x_6 )</th>
<th>( y )</th>
<th>Bit 26</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_3 ) ( x_6 ) ( y )</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nop.i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>hint.i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop.i</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>hint.i</td>
<td>imm21</td>
<td>0</td>
<td>01</td>
</tr>
</tbody>
</table>

Volume 3: Instruction Formats
4.3.4.2 Break (I-Unit)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>break.i</td>
<td>imm21</td>
<td>0</td>
<td>x3 x6</td>
</tr>
</tbody>
</table>

4.3.4.3 Integer Speculation Check (I-Unit)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>chk.s.i</td>
<td>r2, target25</td>
<td>0</td>
<td>x3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>wh</th>
<th>mwh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 21:20</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>sptk</td>
</tr>
<tr>
<td>1</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>dptk</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.mwh.ih</td>
<td>b1 = r2, tag13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mov.ret.mwh.ih</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
4.3.5.2 Move from BR

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 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

4.3.6.2 Move to Predicates – Immediate

4.3.6.3 Move from Predicates/IP

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)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.i</td>
<td>ar₃ = r₂</td>
<td>0</td>
<td>x₃ x₆</td>
</tr>
</tbody>
</table>

### 4.3.7.2 Move to AR – Immediate₈ (I-Unit)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.i</td>
<td>ar₃ = imm₈</td>
<td>0</td>
<td>x₃ x₆</td>
</tr>
</tbody>
</table>

### 4.3.7.3 Move from AR (I-Unit)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.i</td>
<td>r₁ = ar₃</td>
<td>0</td>
<td>x₃ x₆</td>
</tr>
</tbody>
</table>

### 4.3.8 Sign/Zero Extend/Compute Zero Index

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>zxt1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zxt2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zxt4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sx₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sx₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sx₄</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>czx₁.l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>czx₂.l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>czx₁.r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>czx₂.r</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operands Opcode Extension**

- x₃ x₆
- r₁
- Extension values:
  - 00: 10
  - 01: 11
  - 10: 12
  - 11: 14
  - 12: 15
  - 13: 16
  - 14: 18
  - 15: 19
  - 16: 1C
  - 17: 1D
4.3.9 Test Feature

323

4.4 M-Unit Instruction Encodings

4.4.1 Loads and Stores

All load and store instructions are encoded within major opcodes 4, 5, 6, and 7 using a 6-bit opcode extension field in bits 35:30 ($x_6$). Instructions in major opcode 4 (integer load/store, semaphores, and get FR) use two 1-bit opcode extension fields in bit 36 (m) and bit 27 (x) as shown in Table 4-28. 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>m Bit 36</th>
<th>x Bit 27</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>Load/Store (Table 4-30)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>Semaphore/get FR (Table 4-33)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>Load +Reg (Table 4-31)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-29. Floating-point Load/Store/Load Pair/Set FR 1-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>m Bit 36</th>
<th>x Bit 27</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>FP Load/Store (Table 4-34)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>FP Load Pair/set FR (Table 4-37)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>FP Load +Reg (Table 4-35)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>FP Load Pair +Imm (Table 4-38)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>m Bit 36</th>
<th>x Bit 27</th>
<th>x6 Bits 35:32</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>ld1 M2</td>
<td>ld2 M2</td>
<td>ld4 M2</td>
</tr>
<tr>
<td>1</td>
<td>ld1.s M2</td>
<td>ld2.s M2</td>
<td>ld4.s M2</td>
</tr>
<tr>
<td>2</td>
<td>ld1.a M2</td>
<td>ld2.a M2</td>
<td>ld4.a M2</td>
</tr>
<tr>
<td>3</td>
<td>ld1.sa M2</td>
<td>ld2.sa M2</td>
<td>ld4.sa M2</td>
</tr>
<tr>
<td>4</td>
<td>ld1.bias M2</td>
<td>ld2.bias M2</td>
<td>ld4.bias M2</td>
</tr>
<tr>
<td>5</td>
<td>ld1.acq M2</td>
<td>ld2.acq M2</td>
<td>ld4.acq M2</td>
</tr>
<tr>
<td>6</td>
<td>ld1.c.clr M2</td>
<td>ld2.c.clr M2</td>
<td>ld4.c.clr M2</td>
</tr>
<tr>
<td>7</td>
<td>ld1.acq M2</td>
<td>ld2.acq M2</td>
<td>ld4.acq M2</td>
</tr>
<tr>
<td>8</td>
<td>ld1.c.clr M2</td>
<td>ld2.c.clr M2</td>
<td>ld4.c.clr M2</td>
</tr>
<tr>
<td>9</td>
<td>ld1.c.nc M2</td>
<td>ld2.c.nc M2</td>
<td>ld4.c.nc M2</td>
</tr>
<tr>
<td>A</td>
<td>ld1.c.clr.acq M2</td>
<td>ld2.c.clr.acq M2</td>
<td>ld4.c.clr.acq M2</td>
</tr>
<tr>
<td>B</td>
<td>ld1.c.clr.acq M2</td>
<td>ld2.c.clr.acq M2</td>
<td>ld4.c.clr.acq M2</td>
</tr>
<tr>
<td>C</td>
<td>st1 M6</td>
<td>st2 M6</td>
<td>st4 M6</td>
</tr>
<tr>
<td>D</td>
<td>st1.rel M6</td>
<td>st2.rel M6</td>
<td>st4.rel M6</td>
</tr>
<tr>
<td>E</td>
<td>st1.spill M6</td>
<td>st2.spill M6</td>
<td>st4.rel M6</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4-31. Integer Load +Reg Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>m Bit 36</th>
<th>x Bit 27</th>
<th>x6 Bits 35:32</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>ld1 M2</td>
<td>ld2 M2</td>
<td>ld4 M2</td>
</tr>
<tr>
<td>1</td>
<td>ld1.s M2</td>
<td>ld2.s M2</td>
<td>ld4.s M2</td>
</tr>
<tr>
<td>2</td>
<td>ld1.a M2</td>
<td>ld2.a M2</td>
<td>ld4.a M2</td>
</tr>
<tr>
<td>3</td>
<td>ld1.sa M2</td>
<td>ld2.sa M2</td>
<td>ld4.sa M2</td>
</tr>
<tr>
<td>4</td>
<td>ld1.bias M2</td>
<td>ld2.bias M2</td>
<td>ld4.bias M2</td>
</tr>
<tr>
<td>5</td>
<td>ld1.acq M2</td>
<td>ld2.acq M2</td>
<td>ld4.acq M2</td>
</tr>
<tr>
<td>6</td>
<td>ld1.c.clr M2</td>
<td>ld2.c.clr M2</td>
<td>ld4.c.clr M2</td>
</tr>
<tr>
<td>7</td>
<td>ld1.acq M2</td>
<td>ld2.acq M2</td>
<td>ld4.acq M2</td>
</tr>
<tr>
<td>8</td>
<td>ld1.c.clr M2</td>
<td>ld2.c.clr M2</td>
<td>ld4.c.clr M2</td>
</tr>
<tr>
<td>9</td>
<td>ld1.c.nc M2</td>
<td>ld2.c.nc M2</td>
<td>ld4.c.nc M2</td>
</tr>
<tr>
<td>A</td>
<td>ld1.c.clr.acq M2</td>
<td>ld2.c.clr.acq M2</td>
<td>ld4.c.clr.acq M2</td>
</tr>
<tr>
<td>B</td>
<td>ld1.c.clr.acq M2</td>
<td>ld2.c.clr.acq M2</td>
<td>ld4.c.clr.acq M2</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 4-32. Integer Load/Store +Imm Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>Bits 35:32</th>
<th>Bits 31:30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ld1 M3</td>
<td>ld2 M3</td>
</tr>
<tr>
<td>1</td>
<td>ld1.s M3</td>
<td>ld2.s M3</td>
</tr>
<tr>
<td>2</td>
<td>ld1.a M3</td>
<td>ld2.a M3</td>
</tr>
<tr>
<td>3</td>
<td>ld1.sa M3</td>
<td>ld2.sa M3</td>
</tr>
<tr>
<td>4</td>
<td>ld1.bias M3</td>
<td>ld2.bias M3</td>
</tr>
<tr>
<td>5</td>
<td>ld1.acq M3</td>
<td>ld2.acq M3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ld1.c.clr M3</td>
<td>ld2.c.clr M3</td>
</tr>
<tr>
<td>9</td>
<td>ld1.c.nc M3</td>
<td>ld2.c.nc M3</td>
</tr>
<tr>
<td>A</td>
<td>ld1.c.clr.acq M3</td>
<td>ld2.c.clr.acq M3</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>st1 M5</td>
<td>st2 M5</td>
</tr>
<tr>
<td>D</td>
<td>st1.rel M5</td>
<td>st2.rel M5</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4-33. Semaphore/Get FR/16-Byte Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>m Bit 36</th>
<th>x Bits 27</th>
<th>x8 Bits 35:32</th>
<th>Bits 31:30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cmpxchg1.acq M16</td>
<td>cmpxchg2.acq M16</td>
<td>cmpxchg4.acq M16</td>
<td>cmpxchg8.acq M16</td>
</tr>
<tr>
<td>1</td>
<td>cmpxchg1.rel M16</td>
<td>cmpxchg2.rel M16</td>
<td>cmpxchg4.rel M16</td>
<td>cmpxchg8.rel M16</td>
</tr>
<tr>
<td>2</td>
<td>xchg1 M16</td>
<td>xchg2 M16</td>
<td>xchg4 M16</td>
<td>xchg8 M16</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>fetchadd4.acq M17</td>
<td>fetchadd8.acq M17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>fetchadd4.rel M17</td>
<td>fetchadd8.rel M17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>getf.sig M19</td>
<td>getf.exp M19</td>
<td>getf.s M19</td>
<td>getf.d M19</td>
</tr>
<tr>
<td>8</td>
<td>cmp8xchg16.acq M16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>cmp8xchg16.rel M16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>ld16 M2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ld16.acq M2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>st16 M6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>st16.rel M6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 4-34. Floating-point Load/Store/Lfetch Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>m Bit 36</th>
<th>x Bit 27</th>
<th>( x_6 )</th>
<th>( x_6 ) Bits 35:32</th>
<th>( x_6 ) Bits 31:30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>ldfe M9</td>
<td>ldfe M9</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>ldf8, s M9</td>
<td>ldf8, s M9</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
<td>ldfe.a M9</td>
<td>ldfe.a M9</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
<td>ldfe.sa M9</td>
<td>ldfe.sa M9</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>4</td>
<td>ldf.fill M9</td>
<td>ldf.fill M9</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td>ldfc.clr M9</td>
<td>ldfc.clr M9</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>6</td>
<td>ldfc.nc M9</td>
<td>ldfc.nc M9</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>7</td>
<td>lfetch M18</td>
<td>lfetch.excl M18</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>8</td>
<td>lfetchfault M18</td>
<td>lfetchfault.excl M18</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>9</td>
<td>lfetch M13</td>
<td>lfetch.excl M13</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td>A</td>
<td>stfe M13</td>
<td>stfe M13</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>B</td>
<td>stf8 M13</td>
<td>stf8 M13</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>C</td>
<td>stfs M13</td>
<td>stfs M13</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td>D</td>
<td>stfd M13</td>
<td>stfd M13</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>E</td>
<td>stf.spill M13</td>
<td>stf.spill M13</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4-35. Floating-point Load/Lfetch +Reg Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>m Bit 36</th>
<th>x Bit 27</th>
<th>( x_6 )</th>
<th>( x_6 ) Bits 35:32</th>
<th>( x_6 ) Bits 31:30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>ldfe M7</td>
<td>ldfe M7</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>ldf8, s M7</td>
<td>ldf8, s M7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
<td>ldfe.a M7</td>
<td>ldfe.a M7</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
<td>ldfc.sa M7</td>
<td>ldfc.sa M7</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>4</td>
<td>ldf.fill M7</td>
<td>ldf.fill M7</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td>ldfc.clr M7</td>
<td>ldfc.clr M7</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>6</td>
<td>ldfc.nc M7</td>
<td>ldfc.nc M7</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>7</td>
<td>lfetch M20</td>
<td>lfetch.excl M20</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>8</td>
<td>lfetchfault M20</td>
<td>lfetchfault.excl M20</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>9</td>
<td>lfetch M20</td>
<td>lfetch.excl M20</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td>A</td>
<td>stfe M20</td>
<td>stfe M20</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>B</td>
<td>stf8 M20</td>
<td>stf8 M20</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>C</td>
<td>stfs M20</td>
<td>stfs M20</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td>D</td>
<td>stfd M20</td>
<td>stfd M20</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>E</td>
<td>stf.spill M20</td>
<td>stf.spill M20</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 4-36. Floating-point Load/Store/Lfetch +Imm Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>Bits 35:32</th>
<th>Bits 31:30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x&lt;sub&gt;6&lt;/sub&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>ldfe M8</td>
<td>ldff M8</td>
</tr>
<tr>
<td>1</td>
<td>ldfe.s M8</td>
<td>ldff.s M8</td>
</tr>
<tr>
<td>2</td>
<td>ldfe.a M8</td>
<td>ldff.a M8</td>
</tr>
<tr>
<td>3</td>
<td>ldfe.sa M8</td>
<td>ldff.sa M8</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>ldff.fill M8</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ldfe.c.clr M8</td>
<td>ldff.c.clr M8</td>
</tr>
<tr>
<td>9</td>
<td>ldfe.c.nc M8</td>
<td>ldff.c.nc M8</td>
</tr>
<tr>
<td>A</td>
<td>lfetch M22</td>
<td>lfetch.excl M22</td>
</tr>
<tr>
<td>B</td>
<td>lfetch.fault M22</td>
<td>lfetch.fault.excl M22</td>
</tr>
<tr>
<td>C</td>
<td>stfe M10</td>
<td>stff M10</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4-37. Floating-point Load Pair/Set FR Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>m Bit 36</th>
<th>x Bit 27</th>
<th>Bits 35:32</th>
<th>Bits 31:30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x&lt;sub&gt;6&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>ldfp8 M11</td>
<td>ldfps M11</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>ldfp8.s M11</td>
<td>ldfps.s M11</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>ldfp8.a M11</td>
<td>ldfps.a M11</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>ldfp8.sa M11</td>
<td>ldfps.sa M11</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>setf.sig M18</td>
<td>setf.exp M18</td>
<td>setf.s M18</td>
<td>setf.d M18</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>ldfp8.c.clr M11</td>
<td>ldfps.c.clr M11</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>ldfp8.c.nc M11</td>
<td>ldfps.c.nc M11</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 4-38. Floating-point Load Pair +Imm Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>m Bit 36</th>
<th>x Bit 27</th>
<th>x6 Bits 35:32</th>
<th>x6 Bits 31:30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>ldfp8 M12</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>ldfp8.s M12</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
<td>ldfp8.a M12</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
<td>ldfp8.sa M12</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ldfp8 M12</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>1</td>
<td>ldfps M12</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>2</td>
<td>ldfps.s M12</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>3</td>
<td>ldfps.a M12</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>hint Bits 29:28</th>
<th>ldhint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>.nt1</td>
</tr>
<tr>
<td>2</td>
<td>.nta</td>
</tr>
</tbody>
</table>

### Table 4-40. Store Hint Completer

<table>
<thead>
<tr>
<th>hint Bits 29:28</th>
<th>sthint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.nta</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
### 4.4.1.1 Integer Load

#### Instruction Operands Opcode Extension

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld1.ldhint</td>
<td></td>
<td>mxx x</td>
<td>hint</td>
</tr>
<tr>
<td>ld2.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld4.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld8.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld1.s.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld2.s.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld4.s.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld8.s.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld1.a.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld2.a.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld4.a.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld8.a.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld1.sa.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld2.sa.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld4.sa.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld8.sa.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld1.bias.ldhint</td>
<td>r₁ = [r₃]</td>
<td>4  0  0</td>
<td></td>
</tr>
<tr>
<td>ld2.bias.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld4.bias.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld8.bias.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld1.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld2.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld4.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld8.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld8.fill.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld1.c.clr.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld2.c.clr.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld4.c.clr.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld8.c.clr.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld1.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld2.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld4.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld8.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld1.c.clr.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld2.c.clr.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld4.c.clr.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld8.c.clr.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld16.ldhint</td>
<td>r₁, ar.csd = [r₃]</td>
<td>0  1  28</td>
<td></td>
</tr>
<tr>
<td>ld16.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Table 4-39 on page 3:328.
### Integer Load – Increment by Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld1.ldhint</td>
<td></td>
<td></td>
<td>00</td>
</tr>
<tr>
<td>ld2.ldhint</td>
<td></td>
<td></td>
<td>01</td>
</tr>
<tr>
<td>ld4.ldhint</td>
<td></td>
<td></td>
<td>02</td>
</tr>
<tr>
<td>ld8.ldhint</td>
<td></td>
<td></td>
<td>03</td>
</tr>
<tr>
<td>ld1.s.ldhint</td>
<td></td>
<td></td>
<td>04</td>
</tr>
<tr>
<td>ld2.s.ldhint</td>
<td></td>
<td></td>
<td>05</td>
</tr>
<tr>
<td>ld4.s.ldhint</td>
<td></td>
<td></td>
<td>06</td>
</tr>
<tr>
<td>ld8.s.ldhint</td>
<td></td>
<td></td>
<td>07</td>
</tr>
<tr>
<td>ld1.a.ldhint</td>
<td></td>
<td>r1 = [r3], r2 4 1 0</td>
<td>08</td>
</tr>
<tr>
<td>ld2.a.ldhint</td>
<td></td>
<td></td>
<td>09</td>
</tr>
<tr>
<td>ld4.a.ldhint</td>
<td></td>
<td></td>
<td>0A</td>
</tr>
<tr>
<td>ld8.a.ldhint</td>
<td></td>
<td></td>
<td>0B</td>
</tr>
<tr>
<td>ld1.sa.ldhint</td>
<td></td>
<td></td>
<td>0C</td>
</tr>
<tr>
<td>ld2.sa.ldhint</td>
<td></td>
<td></td>
<td>0D</td>
</tr>
<tr>
<td>ld4.sa.ldhint</td>
<td></td>
<td></td>
<td>0E</td>
</tr>
<tr>
<td>ld8.sa.ldhint</td>
<td></td>
<td></td>
<td>0F</td>
</tr>
<tr>
<td>ld1.bias.ldhint</td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>ld2.bias.ldhint</td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>ld4.bias.ldhint</td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>ld8.bias.ldhint</td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>ld1.acq.ldhint</td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>ld2.acq.ldhint</td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>ld4.acq.ldhint</td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>ld8.acq.ldhint</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>ld8.fill.ldhint</td>
<td></td>
<td></td>
<td>1B</td>
</tr>
<tr>
<td>ld1.c.clr.ldhint</td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>ld2.c.clr.ldhint</td>
<td></td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>ld4.c.clr.ldhint</td>
<td></td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>ld8.c.clr.ldhint</td>
<td></td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>ld1.c.nc.ldhint</td>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>ld2.c.nc.ldhint</td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>ld4.c.nc.ldhint</td>
<td></td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>ld8.c.nc.ldhint</td>
<td></td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>ld1.c.clr.acq.ldhint</td>
<td></td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>ld2.c.clr.acq.ldhint</td>
<td></td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>ld4.c.clr.acq.ldhint</td>
<td></td>
<td></td>
<td>2A</td>
</tr>
<tr>
<td>ld8.c.clr.acq.ldhint</td>
<td></td>
<td></td>
<td>2B</td>
</tr>
</tbody>
</table>

See Table 4-39 on page 3:328
### 4.4.1.3 Integer Load – Increment by Immediate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld1.ldhint</td>
<td></td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>ld2.ldhint</td>
<td></td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>ld4.ldhint</td>
<td></td>
<td>02</td>
<td></td>
</tr>
<tr>
<td>ld8.ldhint</td>
<td></td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>ld1.s.ldhint</td>
<td></td>
<td>04</td>
<td></td>
</tr>
<tr>
<td>ld2.s.ldhint</td>
<td></td>
<td>05</td>
<td></td>
</tr>
<tr>
<td>ld4.s.ldhint</td>
<td></td>
<td>06</td>
<td></td>
</tr>
<tr>
<td>ld8.s.ldhint</td>
<td></td>
<td>07</td>
<td></td>
</tr>
<tr>
<td>ld1.a.ldhint</td>
<td></td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>ld2.a.ldhint</td>
<td></td>
<td>09</td>
<td></td>
</tr>
<tr>
<td>ld4.a.ldhint</td>
<td></td>
<td>0A</td>
<td></td>
</tr>
<tr>
<td>ld8.a.ldhint</td>
<td></td>
<td>0B</td>
<td></td>
</tr>
<tr>
<td>ld1.sa.ldhint</td>
<td></td>
<td>0C</td>
<td></td>
</tr>
<tr>
<td>ld2.sa.ldhint</td>
<td></td>
<td>0D</td>
<td></td>
</tr>
<tr>
<td>ld4.sa.ldhint</td>
<td></td>
<td>0E</td>
<td></td>
</tr>
<tr>
<td>ld8.sa.ldhint</td>
<td></td>
<td>0F</td>
<td></td>
</tr>
<tr>
<td>ld1.bias.ldhint</td>
<td></td>
<td>10</td>
<td>See Table 4-39 on page 3:328</td>
</tr>
<tr>
<td>ld2.bias.ldhint</td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>ld4.bias.ldhint</td>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>ld8.bias.ldhint</td>
<td></td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>ld1.acq.ldhint</td>
<td></td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>ld2.acq.ldhint</td>
<td></td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>ld4.acq.ldhint</td>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>ld8.acq.ldhint</td>
<td></td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>ld8.fill.ldhint</td>
<td></td>
<td>1B</td>
<td></td>
</tr>
<tr>
<td>ld1.c.clr.ldhint</td>
<td></td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>ld2.c.clr.ldhint</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>ld4.c.clr.ldhint</td>
<td></td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>ld8.c.clr.ldhint</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>ld1.c.nc.ldhint</td>
<td></td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>ld2.c.nc.ldhint</td>
<td></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>ld4.c.nc.ldhint</td>
<td></td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>ld8.c.nc.ldhint</td>
<td></td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>ld1.c.clr.acq.ldhint</td>
<td></td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>ld2.c.clr.acq.ldhint</td>
<td></td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>ld4.c.clr.acq.ldhint</td>
<td></td>
<td>2A</td>
<td></td>
</tr>
<tr>
<td>ld8.c.clr.acq.ldhint</td>
<td></td>
<td>2B</td>
<td></td>
</tr>
</tbody>
</table>
### 4.4.1.4 Integer Store

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>st1.s hintText</td>
<td>([r_3] = r_2)</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>st2.s hintText</td>
<td>([r_3] = r_2)</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>st16.s hintText</td>
<td>([r_3] = r_2, \text{ar.cs}d)</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### 4.4.1.5 Integer Store – Increment by Immediate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>st1.s hintText</td>
<td>([r_3] = r_2, \text{imm}_g)</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>st2.s hintText</td>
<td>([r_3] = r_2, \text{imm}_g)</td>
<td>5</td>
<td>34</td>
</tr>
</tbody>
</table>
## 4.4.1.6 Floating-point Load

### Table 4-39

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldfe.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.a.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.sa.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldf.fill.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfs.c.clr.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.c.clr.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfs.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldfe.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfs.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ f_1 = [r_3] \]

See Table 4-39 on page 3:328

### Instruction Format

<table>
<thead>
<tr>
<th>m</th>
<th>x</th>
<th>x_6</th>
<th>hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### Instruction Operands

- ldfe.ldhint
- ldfe.a.ldhint
- ldfe.sa.ldhint
- ldf.fill.ldhint
- ldfe.c.clr.ldhint
- ldfe.c.nc.ldhint
- ldfe.c.nc.ldhint
- ldfe.c.nc.ldhint
### 4.4.1.7 Floating-point Load – Increment by Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldafs.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfd.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfl.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldafs.s.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfd.s.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfl.s.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.s.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldafs.a.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfd.a.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfl.a.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.a.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldafs.sa.0dhint</td>
<td>$f_1 = [r_3], r_2$</td>
<td>6 1 0</td>
<td>OE, 0F, 0D, 0C</td>
</tr>
<tr>
<td>ldfd.sa.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfl.sa.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.sa.0dhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfl.fill.0dhint</td>
<td></td>
<td></td>
<td>1B</td>
</tr>
<tr>
<td>ldafs.c.clr.0dhint</td>
<td></td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>ldfd.c.clr.0dhint</td>
<td></td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>ldfl.c.clr.0dhint</td>
<td></td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>ldfe.c.clr.0dhint</td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>ldafs.c.nc.0dhint</td>
<td></td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>ldfd.c.nc.0dhint</td>
<td></td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>ldfl.c.nc.0dhint</td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>ldfe.c.nc.0dhint</td>
<td></td>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>

*See Table 4-39 on page 3:328*
### 4.4.1.8 Floating-point Load – Increment by Immediate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldfs.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfd.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfs.s.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfd.s.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.s.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfs.a.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfd.a.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.a.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfs.sa.ldhint</td>
<td>$f_1 = [r_3], imm_9$</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>ldfd.sa.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.sa.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldff.fill.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfs.c.clr.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfd.c.clr.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.c.clr.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfs.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfd.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldfe.c.nc.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### See Table 4-39 on page 3:328

### 4.4.1.9 Floating-point Store

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>stfs.sthint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stdf.sthint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stdfe.sthint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stdf.fill.sthint</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>stfs.sthint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stdf.sthint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stdfe.sthint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stdf.fill.sthint</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| $[r_3] = f_2$ | 6 | 0 | 0 |

<table>
<thead>
<tr>
<th>$x_6$</th>
<th>hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>02</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td></td>
</tr>
<tr>
<td>0A</td>
<td></td>
</tr>
<tr>
<td>0B</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td></td>
</tr>
<tr>
<td>0E</td>
<td></td>
</tr>
<tr>
<td>0F</td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td></td>
</tr>
<tr>
<td>1B</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>3B</td>
<td></td>
</tr>
</tbody>
</table>

See Table 4-40 on page 3:328
### 4.4.1.10 Floating-point Store – Increment by Immediate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>stfs.sthint</td>
<td></td>
<td>7</td>
<td>x6</td>
</tr>
<tr>
<td>std. sthint</td>
<td></td>
<td>32</td>
<td>hint</td>
</tr>
<tr>
<td>stf8. sthint</td>
<td>r3 = f2, imm9</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>stfe. sthint</td>
<td></td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>stf.spill. sthint</td>
<td></td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

See Table 4-40 on page 3:328.

### 4.4.1.11 Floating-point Load Pair

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldfsps.ldhint</td>
<td></td>
<td>6</td>
<td>m</td>
</tr>
<tr>
<td>ldfpsp.ldhint</td>
<td></td>
<td>02</td>
<td>x6</td>
</tr>
<tr>
<td>ldfpsp8.ldhint</td>
<td></td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>ldfsps.s.ldhint</td>
<td></td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>ldfpsp.s.ldhint</td>
<td></td>
<td>06</td>
<td></td>
</tr>
<tr>
<td>ldfpsp8.s.ldhint</td>
<td></td>
<td>07</td>
<td></td>
</tr>
<tr>
<td>ldfpsp.s.a.ldhint</td>
<td></td>
<td>05</td>
<td></td>
</tr>
<tr>
<td>ldfpsp8.s.a.ldhint</td>
<td></td>
<td>0A</td>
<td></td>
</tr>
<tr>
<td>ldfpsp.sa.ldhint</td>
<td></td>
<td>09</td>
<td></td>
</tr>
<tr>
<td>ldfpsp8.sa.ldhint</td>
<td></td>
<td>0B</td>
<td></td>
</tr>
<tr>
<td>ldfpsp.c.clr.ldhint</td>
<td></td>
<td>0E</td>
<td></td>
</tr>
<tr>
<td>ldfpsp8.c.clr.ldhint</td>
<td></td>
<td>0F</td>
<td></td>
</tr>
<tr>
<td>ldfpsp.c.nc.ldhint</td>
<td></td>
<td>0D</td>
<td></td>
</tr>
<tr>
<td>ldfpsp8.c.nc.ldhint</td>
<td></td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>ldfpsp.c.clr.ldhint</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>ldfpsp8.c.clr.ldhint</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>ldfpsp.c.nc.ldhint</td>
<td></td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>ldfpsp8.c.nc.ldhint</td>
<td></td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>ldfpsp8.c.nc.ldhint</td>
<td></td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

See Table 4-39 on page 3:328.
4.4.1.12 Floating-point Load Pair – Increment by Immediate

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

<table>
<thead>
<tr>
<th>hint</th>
<th>ifhint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>n1</td>
</tr>
<tr>
<td>2</td>
<td>n2</td>
</tr>
<tr>
<td>3</td>
<td>nta</td>
</tr>
</tbody>
</table>
4.4.2.1 Line Prefetch

The instruction format is as follows:

- Instruction: `ifetch.excl.lfhint`
- Operands: `[r3]`
- Opcode: 6
- Extension: 0

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ifetch.excl.lfhint</code></td>
<td><code>[r3]</code></td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

4.4.2.2 Line Prefetch – Increment by Register

The instruction format is as follows:

- Instruction: `ifetch.lfhint`
- Operands: `[r3], r2`
- Opcode: 6
- Extension: 1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ifetch.lfhint</code></td>
<td><code>[r3], r2</code></td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

4.4.2.3 Line Prefetch – Increment by Immediate

The instruction format is as follows:

- Instruction: `ifetch.lfhint`
- Operands: `[r3], imm9`
- Opcode: 7

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ifetch.lfhint</code></td>
<td><code>[r3], imm9</code></td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpxchg1.acq.ldhint</td>
<td>$r_1 = [r_3], r_2, ar.ccv$</td>
<td>4 0 1</td>
<td>00 01 02 03 04 05 06 07 08 09 0A 0B</td>
</tr>
<tr>
<td>cmpxchg2.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpxchg4.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpxchg8.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpxchg1.rel.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpxchg2.rel.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpxchg4.rel.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpxchg8.rel.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp8xchg16.acq.ldhint</td>
<td>$r_1 = [r_3], r_2, ar.csd, ar.ccv$</td>
<td>4 0 1</td>
<td>20 24 08 09 0A 0B</td>
</tr>
<tr>
<td>cmp8xchg16.rel.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xchg1.ldhint</td>
<td>$r_1 = [r_3], r_2$</td>
<td>4 0 1</td>
<td>08 09 0A 0B</td>
</tr>
<tr>
<td>xchg2.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xchg4.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xchg8.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.3.2 Fetch and Add – Immediate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetchadd4.acq.ldhint</td>
<td>$r_1 = [r_3], inc_3$</td>
<td>4 0 1</td>
<td>12 13 16 17</td>
</tr>
<tr>
<td>fetchadd8.acq.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fetchadd4.rel.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fetchadd8.rel.ldhint</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>self.sig</td>
<td>m x6</td>
<td>6</td>
<td>m x6</td>
</tr>
<tr>
<td>self.exp</td>
<td>f1 = r2</td>
<td>0</td>
<td>1C</td>
</tr>
<tr>
<td>self.s</td>
<td></td>
<td></td>
<td>1D</td>
</tr>
<tr>
<td>self.d</td>
<td></td>
<td></td>
<td>1E</td>
</tr>
</tbody>
</table>

### 4.4.4.2 Get FR

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>getf.sig</td>
<td>r1 = f2</td>
<td>4</td>
<td>m x6</td>
</tr>
<tr>
<td>getf.exp</td>
<td></td>
<td></td>
<td>1C</td>
</tr>
<tr>
<td>getf.s</td>
<td></td>
<td></td>
<td>1D</td>
</tr>
<tr>
<td>getf.d</td>
<td></td>
<td></td>
<td>1E</td>
</tr>
</tbody>
</table>

### 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)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>chk.s.m</td>
<td>r2, target25</td>
<td>1</td>
<td>x3</td>
</tr>
</tbody>
</table>

#### 4.4.5.2 Floating-point Speculation Check

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>chk.s</td>
<td>f2, target25</td>
<td>1</td>
<td>x3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
4.4.5.3 **Integer Advanced Load Check**

M22

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>chk.a.nc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chk.a.clc</td>
<td>r1, target25</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

4.4.5.4 **Floating-point Advanced Load Check**

M23

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>chk.a.nc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chk.a.clc</td>
<td>f1, target25</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

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**

M24

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>invala</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fwb</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>mf</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>mf.a</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>srlz.d</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>srlz.i</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>sync.i</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Volume 3: Instruction Formats
### 4.4.6.2 RSE Control

<table>
<thead>
<tr>
<th>M25</th>
<th>40 373635 33323130 2726</th>
<th>6 5 0</th>
</tr>
</thead>
</table>

**Instruction** | **Opcode** | **Extension**
--- | --- | ---
flushrs | x₃ x₄ x₂ | 
loadrs | 0 | C A 0 |

### 4.4.6.3 Integer ALAT Entry Invalidate

<table>
<thead>
<tr>
<th>M26</th>
<th>40 373635 33323130 2726</th>
<th>1312 6 5 0</th>
</tr>
</thead>
</table>

**Instruction** | **Operands** | **Opcode** | **Extension**
--- | --- | --- | ---
invala.e | r₁ | x₃ x₄ x₂ |

### 4.4.6.4 Floating-point ALAT Entry Invalidate

<table>
<thead>
<tr>
<th>M27</th>
<th>40 373635 33323130 2726</th>
<th>1312 6 5 0</th>
</tr>
</thead>
</table>

**Instruction** | **Operands** | **Opcode** | **Extension**
--- | --- | --- | ---
invala.e | f₁ | x₃ x₄ x₂ |

### 4.4.6.5 Flush Cache

<table>
<thead>
<tr>
<th>M28</th>
<th>40 373635 3332 2726</th>
<th>2019 6 5 0</th>
</tr>
</thead>
</table>

**Instruction** | **Operands** | **Opcode** | **Extension**
--- | --- | --- | ---
fc | r₃ | x₃ x₆ x |

### 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)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.m</td>
<td>ar&lt;sub&gt;3&lt;/sub&gt; = r&lt;sub&gt;2&lt;/sub&gt;</td>
<td>1</td>
<td>&lt;i&gt;x&lt;sub&gt;3&lt;/sub&gt;&lt;/i&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.p</td>
<td>cr&lt;sub&gt;3&lt;/sub&gt; = r&lt;sub&gt;2&lt;/sub&gt;</td>
<td>1</td>
<td>&lt;i&gt;x&lt;sub&gt;3&lt;/sub&gt;&lt;/i&gt;</td>
</tr>
</tbody>
</table>

### 4.4.7.2 Move to AR – Immediate (M-Unit)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.m</td>
<td>ar&lt;sub&gt;3&lt;/sub&gt; = imm&lt;sub&gt;8&lt;/sub&gt;</td>
<td>0</td>
<td>&lt;i&gt;x&lt;sub&gt;3&lt;/sub&gt;&lt;/i&gt;</td>
</tr>
</tbody>
</table>

### 4.4.7.3 Move from AR (M-Unit)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.m</td>
<td>r&lt;sub&gt;1&lt;/sub&gt; = ar&lt;sub&gt;3&lt;/sub&gt;</td>
<td>1</td>
<td>&lt;i&gt;x&lt;sub&gt;3&lt;/sub&gt;&lt;/i&gt;</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.p</td>
<td>cr&lt;sub&gt;3&lt;/sub&gt; = r&lt;sub&gt;2&lt;/sub&gt;</td>
<td>1</td>
<td>&lt;i&gt;x&lt;sub&gt;3&lt;/sub&gt;&lt;/i&gt;</td>
</tr>
</tbody>
</table>

### 4.4.8.2 Move from CR

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov.p</td>
<td>r&lt;sub&gt;1&lt;/sub&gt; = cr&lt;sub&gt;3&lt;/sub&gt;</td>
<td>1</td>
<td>&lt;i&gt;x&lt;sub&gt;3&lt;/sub&gt;&lt;/i&gt;</td>
</tr>
</tbody>
</table>
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

### Instruction: Allocate Register Stack Frame

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc^{f}</td>
<td>r1 = ar.pfs, i, l, o, r</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

**Note:** The three immediates in the instruction encoding are formed from the operands as follows:

- sof = i + l + o
- sol = i + l
- sor = r >> 3

### Move to PSR

4.4.9.2 Move to PSR

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov^{p}</td>
<td>psr.l = r2</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>mov</td>
<td>psr.um = r2</td>
<td>0</td>
<td>29</td>
</tr>
</tbody>
</table>

### Move from PSR

4.4.9.3 Move from PSR

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov^{p}</td>
<td>r1 = psr</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>mov</td>
<td>r1 = psr.um</td>
<td>0</td>
<td>21</td>
</tr>
</tbody>
</table>

### Break (M-Unit)

4.4.9.4 Break (M-Unit)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>break.m</td>
<td>imm21</td>
<td>0</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>( x_3 ) Bits 35:33</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>System/Memory Management 4-bit+2-bit Ext (Table 4-43)</td>
</tr>
<tr>
<td>0</td>
<td>System/Memory Management 4-bit+2-bit Ext (Table 4-43)</td>
</tr>
<tr>
<td>1</td>
<td>System/Memory Management 4-bit+2-bit Ext (Table 4-43)</td>
</tr>
<tr>
<td>2</td>
<td>System/Memory Management 4-bit+2-bit Ext (Table 4-43)</td>
</tr>
<tr>
<td>3</td>
<td>System/Memory Management 4-bit+2-bit Ext (Table 4-43)</td>
</tr>
<tr>
<td>4</td>
<td>chk.a.nc – int M22</td>
</tr>
<tr>
<td>5</td>
<td>chk.a.clr – int M22</td>
</tr>
<tr>
<td>6</td>
<td>chk.a.nc – fp M23</td>
</tr>
<tr>
<td>7</td>
<td>chk.a.clr – fp M23</td>
</tr>
</tbody>
</table>

Table 4-43. Opcode 0 System/Memory Management 4-bit+2-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>( x_3 ) Bits 35:33</th>
<th>( x_4 ) Bits 30:27</th>
<th>( x_2 ) Bits 32:31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>break.m M37</td>
<td>invala M24</td>
<td>fwb M24</td>
</tr>
<tr>
<td>1</td>
<td>1-bit Ext         (Table 4-46)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>invala.e – int M26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>invala.e – fp M27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 4-44. Opcode 1 System/Memory Management 3-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x₃ Bits 35:33</th>
<th>Bits 30:27</th>
<th>x₆ Bits 32:31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction:** System/Memory Management 6-bit Ext (Table 4-45)

- chk.s.m – int M20
- chk.s – fp M21
- alloc M34

### Table 4-45. Opcode 1 System/Memory Management 6-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x₃ Bits 35:33</th>
<th>Bits 30:27</th>
<th>x₆ Bits 32:31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction:**

- mov to rr M42
- mov from rr M43
- mov to dbr M42
- mov from dbr M43
- mov from psr.um M36
- probe.rw.fault – imm₂ M40
- mov to ibr M42
- mov from ibr M43
- mov from ar M31
- probe.rfault – imm₂ M40
- mov to pkr M42
- mov from pkr M43
- probe.w.fault – imm₂ M40
- mov to pmc M42
- mov from pmc M43
- mov from cr M33
- ptc.e M47
- mov to pmd M42
- mov from pmd M43
- mov from psr M36
- probe.r – imm₂ M39
- probe.w – imm₂ M39
- mov to psr.um M35
- probe.w M38
- ptc.g M45
- thash M46
- mov.m to ar M29
- ptc.w M38
- ptr.d M45
- ttag M46
- mov to cr M32
- ptc.i M45
- mov to psr.i M35
- ptr.i M45
- mov to psr.i M35
- ptr.g M45
- mov to cr M32
- ptr.m M45
- mov to cr M32
- ptr.m M45
- mov to cr M32

### 4.4.10.1 Probe – Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>probe.r</td>
<td>r₁ = r₃, r₂</td>
<td>1</td>
<td>38</td>
</tr>
<tr>
<td>probe.w</td>
<td></td>
<td>1</td>
<td>39</td>
</tr>
</tbody>
</table>

**Diagram:**

```
M38
```

```
probe.r
probe.w
```

```
probes
```

```
r₁ = r₃, r₂
```

```
probe
```

```
1
```

```
x₃ x₆
```

```
1 38
```

```
0 39
```

```
4 1 3 6 7 7 7 6 0
```

```
40 37 36 35 33 32 27 26 20 19 13 12 6 5 0
```
### 4.4.10.2 Probe – Immediate

**Instruction**: probe.r, probe.w

**Operands**: \( r_1 = r_3, \text{imm}_2 \)

**Opcode**: 1

**Extension**: 0, 18, 19

### 4.4.10.3 Probe Fault – Immediate

**Instruction**: probe.rw.fault, probe.r.fault, probe.w.fault

**Operands**: \( r_3, \text{imm}_2 \)

**Opcode**: 1

**Extension**: 0, 31, 32, 33

### 4.4.10.4 Translation Cache Insert

**Instruction**: itc.d, itc.i

**Operands**: \( r_2 \)

**Opcode**: 1

**Extension**: 0, 2E, 2F

### 4.4.10.5 Move to Indirect Register/Translation Register Insert

**Instruction**: mov, itr.d, itr.i

**Operands**: \( \text{rr}[r_3] = r_2, \text{dbr}[r_3] = r_2, \text{ibr}[r_3] = r_2, \text{pk}[r_3] = r_2, \text{pm}[r_3] = r_2, \text{pmd}[r_3] = r_2, \text{itr}[r_3] = r_2, \text{ltr}[r_3] = r_2 \)

**Opcode**: 1

**Extension**: 0, 00, 01, 02, 03, 04, 05, 0E, 0F
## 4.4.10.6 Move from Indirect Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov<code>p</code></td>
<td><code>r1 = rr[r3]</code></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td><code>r1 = dbr[r3]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>r1 = ibr[r3]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>r1 = pkf[r3]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>r1 = pmc[r3]</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

mov  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>r1 = pmd[r3]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>r1 = cpuid[r3]</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## 4.4.10.7 Set/Reset User/System Mask

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td><code>imm_{24}</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rum</td>
<td></td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>ssm<code>p</code></td>
<td></td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>rsm<code>p</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## 4.4.10.8 Translation Purge

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptc.l<code>p</code></td>
<td><code>r3, r2</code></td>
<td>1</td>
<td>09</td>
</tr>
<tr>
<td>ptc.g<code>p</code></td>
<td></td>
<td>0A</td>
<td></td>
</tr>
<tr>
<td>ptc.ga<code>p</code></td>
<td></td>
<td>0B</td>
<td></td>
</tr>
<tr>
<td>ptr.d<code>p</code></td>
<td></td>
<td>0C</td>
<td></td>
</tr>
<tr>
<td>ptr.i<code>p</code></td>
<td></td>
<td>0D</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
4.4.10.9  Translation Access

---

4.4.10.10  Purge Translation Cache Entry

---

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 (x₃), a 2-bit opcode extension field in bits 32:31 (x₂), a 4-bit opcode extension field in bits 30:27 (x₄), 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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x₃ Bits 35:33</th>
<th>x₄ Bits 30:27</th>
<th>x₂ Bits 32:31</th>
<th>y Bit 26</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>nop.m</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>hint.m</td>
</tr>
</tbody>
</table>

---

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>btype</th>
<th>Bits 8:6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>br.cond B1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>br.wexit B1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>br.wtop B1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>br.cloop B2</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>br.cexit B2</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>br.ctop B2</td>
</tr>
</tbody>
</table>

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 (x6). Table 4-48 summarizes these assignments.

Table 4-48. Indirect/Miscellaneous Branch Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bits 40:37</th>
<th>x6</th>
<th>Bits 32:31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>break.b B9</td>
<td>e</td>
<td>Indirect Branch (Table 4-49)</td>
</tr>
<tr>
<td>1</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>cover B8</td>
<td>e</td>
<td>Indirect Return (Table 4-50)</td>
</tr>
<tr>
<td>3</td>
<td>clrrb B8</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>clrrb.pr B8</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>8</td>
<td>rfi B8</td>
<td>e</td>
<td>vmsw.0 B8</td>
</tr>
<tr>
<td>9</td>
<td>vmsw.1 B8</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>A</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>B</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>C</td>
<td>bsw.0 B8</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>D</td>
<td>bsw.1 B8</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>E</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>F</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x6 Bits 32:27</th>
<th>btype Bits 8:6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>br.cond B4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>br.ia B4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>e</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x6 Bits 32:27</th>
<th>btype Bits 8:6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>br.ret B4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>p Bit 12</th>
<th>ph</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.few</td>
</tr>
<tr>
<td>1</td>
<td>.many</td>
</tr>
</tbody>
</table>

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.
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

<table>
<thead>
<tr>
<th>d</th>
<th>Bit 35</th>
<th>dh</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>.clr</td>
<td></td>
</tr>
</tbody>
</table>

4.5.1.1 IP-Relative Branch
4.5.1.2 IP-Relative Counted Branch

```
Instruction | Operands | Opcode | Extension
-------------|----------|--------|----------
br.cloop.bwh.ph.dh | s,d,wh | 40 3736 3534 3332 | imm32b | btype p blype
```

4.5.1.3 IP-Relative Call

```
Instruction | Operands | Opcode | Extension
-------------|----------|--------|----------
br.call.bwh.ph.dh | s,d,wh | 40 3736 3534 3332 | imm32b | p btype
```

4.5.1.4 Indirect Branch

```
Instruction | Operands | Opcode | Extension
-------------|----------|--------|----------
br.cond.bwh.ph.dh | s,d,wh | 40 3736 3534 3332 | imm32b | x6 btype
```

4.5.1.5 Indirect Call

```
Instruction | Operands | Opcode | Extension
-------------|----------|--------|----------
br.call.bwh.ph.dh | s,d,wh | 40 3736 3534 32 31 | b2 | p
```

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.
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 4-55. Indirect Predict/Nop/Hint Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>Bits 30:27</th>
<th>Bits 32:31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>nop.b B9</td>
<td>brp B7</td>
</tr>
<tr>
<td>1</td>
<td>hint.b B9</td>
<td>brp.ret B7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 4-56. Branch Importance Hint Completer

<table>
<thead>
<tr>
<th>ih Bit 23 or Bit 35</th>
<th>ih</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>.imp</td>
</tr>
</tbody>
</table>

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.
4.5.2.1 IP-Relative Predict

Table 4-58. Indirect Predict Whether Hint Completer

<table>
<thead>
<tr>
<th>wh</th>
<th>indwh</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sptk</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>dptk</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

### 4.5.2.2 Indirect Predict

#### Instruction Operands Opcode Extension

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>brp.ipwh.hh</td>
<td>target25, tag13</td>
<td>7</td>
<td>See Table 4-56 on page 3:354, See Table 4-57 on page 3:354</td>
</tr>
<tr>
<td>brp.indwh.hh</td>
<td>b2, tag13</td>
<td>2</td>
<td>10, 11</td>
</tr>
</tbody>
</table>

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 (x6) as described in Table 4-48 on page 3:350.

4.5.3.1 Miscellaneous (B-Unit)

#### Instruction Opcode Extension

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>cover^l</td>
<td></td>
<td>x6</td>
</tr>
<tr>
<td>clrrb</td>
<td></td>
<td>02</td>
</tr>
<tr>
<td>clrrb.pr</td>
<td></td>
<td>04</td>
</tr>
<tr>
<td>rfilp</td>
<td></td>
<td>05</td>
</tr>
<tr>
<td>bsw.0lp</td>
<td></td>
<td>06</td>
</tr>
<tr>
<td>bsw.1lp</td>
<td></td>
<td>08</td>
</tr>
<tr>
<td>epc</td>
<td></td>
<td>0C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
4.5.3.2 Break/Nop/Hint (B-Unit)

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 (x6) 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 x6 assignments.

Table 4-59. Miscellaneous Floating-point 1-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>break.b (^a)</td>
<td></td>
<td>0</td>
<td>x6</td>
</tr>
<tr>
<td>nop.b</td>
<td></td>
<td>2</td>
<td>00</td>
</tr>
<tr>
<td>hint.b</td>
<td>(imm_{21})</td>
<td></td>
<td>01</td>
</tr>
</tbody>
</table>

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.
### Table 4-60. Opcode 0 Miscellaneous Floating-point 6-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x Bit 33</th>
<th>Bits 30:27</th>
<th>x6</th>
<th>Bits 32:31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>break.f F15</td>
<td>fmerge.s F9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1-bit Ext (Table 4-68)</td>
<td>fmerge.ns F9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>fmerge.se F9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>fsetc F12</td>
<td>fmin F8</td>
<td>fswap F9</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>fcirl F13</td>
<td>fmax F8</td>
<td>fswap.nl F9</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>famin F8</td>
<td>fswap nr F9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>famax F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>fchkf F14</td>
<td>fcvt.fx F10</td>
<td>fpack F9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>fcvt.fxu F10</td>
<td></td>
<td>fmix lr F9</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>fcvt.fx trunc F10</td>
<td>fmix r F9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>fcvt.fxu trunc F10</td>
<td>fmix l F9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>fcvt.xf F11</td>
<td>fand F9</td>
<td>fsxt r F9</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>fandcm F9</td>
<td>fsgt I F9</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>for F9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>fxor F9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4-61. Opcode 1 Miscellaneous Floating-point 6-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x Bit 33</th>
<th>Bits 30:27</th>
<th>x6</th>
<th>Bits 32:31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>fmerge.s F9</td>
<td></td>
<td>fpcmp.eq F8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>fmerge.ns F9</td>
<td></td>
<td>fpcmp.lt F8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>fmerge.se F9</td>
<td></td>
<td>fpcmp.le F8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>fpcmp.unord F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>fmin F8</td>
<td>fpcmp.neq F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>fmax F8</td>
<td>fpcmp.nl F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>fparen F8</td>
<td>fpcmp.nle F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>fpmx F8</td>
<td>fpcmp.ord F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>fpcvt.fx F10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>fpcvt.fxu F10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>fpcvt.fx trunc F10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>fpcvt.fxu trunc F10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>sf</th>
<th>Bits 35:34</th>
<th>sf</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.s0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>.s1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.s2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.s3</td>
<td></td>
</tr>
</tbody>
</table>

### 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**

<table>
<thead>
<tr>
<th>x Bit 36</th>
<th>Opcode Bits 40:37</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fma F1 fma.d F1 fms F1 fms.d F1 fnma F1 fnma.d F1</td>
</tr>
<tr>
<td>1</td>
<td>fma.s F1 fpma F1 fms.s F1 fpms F1 fnma.s F1 fpnma F1</td>
</tr>
</tbody>
</table>

The fixed-point arithmetic and parallel floating-point select instructions are encoded within major opcode E using a 1-bit opcode extension field (x) in bit 36. The fixed-point arithmetic instructions also have a 2-bit opcode extension field ($x_2$) in bits 35:34. These assignments are shown in Table 4-65.

**Table 4-65. Fixed-point Multiply Add and Select Opcode Extensions**

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x Bit 36</th>
<th>$x_2$ Bits 35:34</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>0</td>
<td>fselect F3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>xma.I F2 xma.hu F2 xma.h F2</td>
</tr>
</tbody>
</table>
4.6.1.1 Floating-point Multiply Add

4.6.1.2 Fixed-point Multiply Add

4.6.2 Parallel Floating-point Select

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 (ra), 36 (rb), and 12 (ta), and a 2-bit opcode extension field (sf) in bits 35:34. The opcode, ra, rb, and ta 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.
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 4-66. Floating-point Compare Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>(t_a)</th>
<th>Bit 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bits 40:37</th>
<th>(t_a)</th>
<th>Bit 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fcmp.eq F4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>fcmp.eq.unc F4</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bits 40:37</th>
<th>(t_a)</th>
<th>Bit 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fcmp.le F4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>fcmp.le.unc F4</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bits 40:37</th>
<th>(t_a)</th>
<th>Bit 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fcmp.unord F4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>fcmp.unord.unc F4</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 4-67. Floating-point Class 1-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>(t_a)</th>
<th>Bit 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>fclass.m F5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>fclass.m.unc F5</td>
</tr>
</tbody>
</table>

### 4.6.3.1 Floating-point Compare

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcmp.eq.sf</td>
<td>(p_1, p_2 = f_2, f_3)</td>
<td>4</td>
<td>0 0 1 0 1</td>
</tr>
<tr>
<td>fcmp.lt.sf</td>
<td>(p_1, p_2 = f_2, f_3)</td>
<td>4</td>
<td>1 0 0 1 1</td>
</tr>
<tr>
<td>fcmp.le.sf</td>
<td>(p_1, p_2 = f_2, f_3)</td>
<td>4</td>
<td>1 1 0 1 1</td>
</tr>
<tr>
<td>fcmp.unord.sf</td>
<td>(p_1, p_2 = f_2, f_3)</td>
<td>4</td>
<td>1 1 0 1 1</td>
</tr>
</tbody>
</table>

See Table 4-63 on page 3:358

### 4.6.3.2 Floating-point Class

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>fclass.m</td>
<td>(p_1, p_2 = f_2, fclass_{\text{sf}})</td>
<td>5</td>
<td>0 0</td>
</tr>
<tr>
<td>fclass.m.unc</td>
<td>(p_1, p_2 = f_2, fclass_{\text{sf}})</td>
<td>5</td>
<td>0 0</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>frcpa.sf</td>
<td>$f_1, p_2 = f_2, f_3$</td>
<td>0, 1</td>
<td>0, 1</td>
</tr>
<tr>
<td>fprcpa.sf</td>
<td>$f_1, p_2 = f_2, f_3$</td>
<td>0, 1</td>
<td>0, 1</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>frsqta.sf</td>
<td>$f_1, p_2 = f_3$</td>
<td>0, 1</td>
<td>1, 1</td>
</tr>
<tr>
<td>fprsqta.sf</td>
<td>$f_1, p_2 = f_3$</td>
<td>0, 1</td>
<td>1, 1</td>
</tr>
</tbody>
</table>
4.6.5 Minimum/Maximum and Parallel Compare

There are two groups of Minimum/Maximum instructions. The first group, in major op 0, encodes the full register variants. The second group, in major op 1, encodes the parallel variants. The parallel compare instructions are all encoded in major op 1.

### Instruction Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmin.sf</td>
<td>x</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>fmax.sf</td>
<td>sf</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>famin.sf</td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>famax.sf</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>fpmin.sf</td>
<td>f1 = f2, f3</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>fpmax.sf</td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>fpamin.sf</td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>fpamax.sf</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>fpcmp.eq.sf</td>
<td></td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>fpcmp.lt.sf</td>
<td></td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>fpcmp.le.sf</td>
<td></td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>fpcmp.unord.sf</td>
<td></td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>fpcmp.neq.sf</td>
<td></td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>fpcmp.nlt.sf</td>
<td></td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>fpcmp.nle.sf</td>
<td></td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>fpcmp.ord.sf</td>
<td></td>
<td></td>
<td>37</td>
</tr>
</tbody>
</table>

See Table 4-63 on page 3:358
### 4.6.6 Merge and Logical

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fmerge.s</code></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td><code>fmerge.ns</code></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td><code>fmerge.se</code></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td><code>fmix.lr</code></td>
<td></td>
<td></td>
<td>39</td>
</tr>
<tr>
<td><code>fmix.r</code></td>
<td></td>
<td></td>
<td>3A</td>
</tr>
<tr>
<td><code>fmix.l</code></td>
<td></td>
<td></td>
<td>3B</td>
</tr>
<tr>
<td><code>fsxt.r</code></td>
<td></td>
<td></td>
<td>3C</td>
</tr>
<tr>
<td><code>fsxt.l</code></td>
<td></td>
<td></td>
<td>3D</td>
</tr>
<tr>
<td><code>fpack</code></td>
<td></td>
<td></td>
<td>28</td>
</tr>
<tr>
<td><code>fswap</code></td>
<td></td>
<td></td>
<td>34</td>
</tr>
<tr>
<td><code>fswap.nl</code></td>
<td></td>
<td></td>
<td>35</td>
</tr>
<tr>
<td><code>fswap.nr</code></td>
<td></td>
<td></td>
<td>36</td>
</tr>
<tr>
<td><code>fand</code></td>
<td></td>
<td></td>
<td>2C</td>
</tr>
<tr>
<td><code>fandcm</code></td>
<td></td>
<td></td>
<td>2D</td>
</tr>
<tr>
<td><code>for</code></td>
<td></td>
<td></td>
<td>2E</td>
</tr>
<tr>
<td><code>fxor</code></td>
<td></td>
<td></td>
<td>2F</td>
</tr>
<tr>
<td><code>fpmerge.s</code></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td><code>fpmerge.ns</code></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td><code>fpmerge.se</code></td>
<td></td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

### 4.6.7 Conversion

#### 4.6.7.1 Convert Floating-point to Fixed-point

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fcvt.fx.sf</code></td>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td><code>fcvt.fxu.sf</code></td>
<td></td>
<td></td>
<td>19</td>
</tr>
<tr>
<td><code>fcvt.fx.trunc.sf</code></td>
<td></td>
<td></td>
<td>1A</td>
</tr>
<tr>
<td><code>fcvt.fxu.trunc.sf</code></td>
<td></td>
<td></td>
<td>1B</td>
</tr>
<tr>
<td><code>fpfcvt.fx.sf</code></td>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td><code>fpfcvt.fxu.sf</code></td>
<td></td>
<td></td>
<td>19</td>
</tr>
<tr>
<td><code>fpfcvt.fx.trunc.sf</code></td>
<td></td>
<td></td>
<td>1A</td>
</tr>
<tr>
<td><code>fpfcvt.fxu.trunc.sf</code></td>
<td></td>
<td></td>
<td>1B</td>
</tr>
</tbody>
</table>
### 4.6.7.2 Convert Fixed-point to Floating-point

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcvt.xf</td>
<td>$f_1 = f_2$</td>
<td>0</td>
<td>0 1C</td>
</tr>
</tbody>
</table>

### 4.6.8 Status Field Manipulation

#### 4.6.8.1 Floating-point Set Controls

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsetc.sf</td>
<td>amask$_7$, omask$_7$</td>
<td>0 0 04</td>
<td>See Table 4-63 on page 3:358</td>
</tr>
</tbody>
</table>

#### 4.6.8.2 Floating-point Clear Flags

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcclrf.sf</td>
<td>0 0 05</td>
<td>See Table 4-63 on page 3:358</td>
</tr>
</tbody>
</table>

#### 4.6.8.3 Floating-point Check Flags

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>fchkf.sf</td>
<td>target$_{25}$</td>
<td>0 0 08</td>
<td>See Table 4-63 on page 3:358</td>
</tr>
</tbody>
</table>
4.6.9 Miscellaneous F-Unit Instructions

4.6.9.1 Break (F-Unit)

F15

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>break.f</td>
<td>imm21</td>
<td>0</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

Table 4-68. Misc F-Unit 1-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x Bit 33</th>
<th>x6 Bits 32:27</th>
<th>y Bit 26</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F16

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop.f</td>
<td>imm21</td>
<td>0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>hint.f</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 (x3), a 6-bit opcode extension field in bits 32:27 (x6), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-46.

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.

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 imm41 field occupies the L instruction slot. For brl, the imm39 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 (x3) in bits 35:33 and a 6-bit opcode extension field (x6) 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.
4.7.1.1 Break (X-Unit)

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 4-69. Misc X-Unit 3-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>(x_3) Bits 35:33</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6-bit Ext (Table 4-70)</td>
</tr>
<tr>
<td>1</td>
<td>(x_3)</td>
</tr>
<tr>
<td>2</td>
<td>(x_3)</td>
</tr>
<tr>
<td>3</td>
<td>(x_3)</td>
</tr>
<tr>
<td>4</td>
<td>(x_3)</td>
</tr>
<tr>
<td>5</td>
<td>(x_3)</td>
</tr>
<tr>
<td>6</td>
<td>(x_3)</td>
</tr>
<tr>
<td>7</td>
<td>(x_3)</td>
</tr>
</tbody>
</table>

### Table 4-70. Misc X-Unit 6-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>(x_3) Bits 35:33</th>
<th>(x_6) Bits 30:27</th>
<th>(x_6) Bits 32:31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>break.x X1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1-bit Ext</td>
<td>(Table 4-73)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>(x_3)</td>
<td>(x_6)</td>
</tr>
</tbody>
</table>

**4.7.2 Move Long Immediate\(_{64}\)**

The move long immediate instruction is encoded within major opcode 6 using a 1-bit reserved opcode extension in bit 20 (\(v_c\)) as shown in Table 4-71. This instruction is executed by an I-unit.
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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>btype</th>
<th>Bit 8:6</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>brl.cond X3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>btype</th>
<th>wh</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>brl.cond.bwh.ph.dh</td>
<td>target64</td>
<td>C</td>
<td>0</td>
<td>See Table 4-51 on page 3:351</td>
<td>See Table 4-52 on page 3:352</td>
</tr>
</tbody>
</table>

---

X2

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl X2</td>
<td>r1 = imm64</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

---

Table 4-71. Move Long 1-bit Opcode Extensions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bits 40:37</th>
<th>v_c</th>
<th>Bit 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

---

Row: 6

X2
4.7.3.2 Long Call

X4

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Opcode</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>brl.call.bwh.ph.dh</td>
<td>b1 = target64</td>
<td>D</td>
<td>See Table 4-51 on page 3:351</td>
</tr>
</tbody>
</table>

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 (x3), a 6-bit opcode extension field in bits 32:27 (x6), 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

<table>
<thead>
<tr>
<th>Opcode Bits 40:37</th>
<th>x3 Bits 35:33</th>
<th>x6 Bits 32:27</th>
<th>y Bit 26</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

X5

Table 4-74. 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>
<thead>
<tr>
<th>Instruction Format</th>
<th>Immediate Formation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>count2 = ct2d + 1</td>
</tr>
<tr>
<td>A3 A8 I27 M30</td>
<td>imm8 = sign_ext(s &lt;&lt; 7</td>
</tr>
<tr>
<td>A4</td>
<td>imm3d = sign_ext(s &lt;&lt; 13</td>
</tr>
<tr>
<td>A5</td>
<td>imm2g = sign_ext(s &lt; 21</td>
</tr>
<tr>
<td>A10</td>
<td>count2 = (ct2d &gt; 2) ? reservedQPb : ct2d + 1</td>
</tr>
<tr>
<td>I1</td>
<td>count2 = (ct2d == 0) ? 0 : (ct2d == 1) ? 7 : (ct2d == 2) ? 15 : 16</td>
</tr>
<tr>
<td>Instruction Format</td>
<td>Immediate Formation</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>I3</td>
<td>mbtype4 = (mbt4c == 0) ? @brct : (mbt4c == 8) ? @mix : (mbt4c == 9) ? @shuf : (mbt4c == 0xA) ? @alt : (mbt4c == 0xB) ? @rev : reservedQP</td>
</tr>
<tr>
<td>I4</td>
<td>mhtype8 = mht8c</td>
</tr>
<tr>
<td>I6</td>
<td>count5 = count5b</td>
</tr>
<tr>
<td>I8</td>
<td>count3 = 31 – ccount5c</td>
</tr>
<tr>
<td>I10</td>
<td>count5 = count5d</td>
</tr>
<tr>
<td>I11</td>
<td>len8 = len6d + 1</td>
</tr>
<tr>
<td></td>
<td>pos8 = pos6b</td>
</tr>
<tr>
<td>I12</td>
<td>len8 = len6d + 1</td>
</tr>
<tr>
<td></td>
<td>pos8 = 63 – cpos5c</td>
</tr>
<tr>
<td>I13</td>
<td>len8 = len6d + 1</td>
</tr>
<tr>
<td></td>
<td>pos8 = 63 – cpos5c</td>
</tr>
<tr>
<td></td>
<td>imm8 = sign_ext(s &lt;&lt; 7</td>
</tr>
<tr>
<td>I14</td>
<td>len8 = len6d + 1</td>
</tr>
<tr>
<td></td>
<td>pos8 = 63 – cpos6b</td>
</tr>
<tr>
<td></td>
<td>imm1 = sign_ext(s, 1)</td>
</tr>
<tr>
<td>I15</td>
<td>len4 = len6d + 1</td>
</tr>
<tr>
<td></td>
<td>pos6 = 63 – cpos6d</td>
</tr>
<tr>
<td>I16</td>
<td>pos8 = pos6b</td>
</tr>
<tr>
<td>I18 I19 M37 M48</td>
<td>imm21 = i &lt;&lt; 20</td>
</tr>
<tr>
<td>I21</td>
<td>tag13 = IP + (sign_ext(timm9c, 9) &lt;&lt; 4)</td>
</tr>
<tr>
<td>I23</td>
<td>mask17 = sign_ext(s &lt;&lt; 16</td>
</tr>
<tr>
<td>I24</td>
<td>imm44 = sign_ext(s &lt;&lt; 43</td>
</tr>
<tr>
<td>I30</td>
<td>imm9 = imm5b + 32</td>
</tr>
<tr>
<td>M3 M8 M22</td>
<td>imm9 = sign_ext(s &lt;&lt; 8</td>
</tr>
<tr>
<td>M5 M10</td>
<td>imm9 = sign_ext(s &lt;&lt; 8</td>
</tr>
<tr>
<td>M17</td>
<td>inc3 = sign_ext(((s) ? –1 : 1) * ((i2b == 3) ? 1 : 1 &lt;&lt; (4 – i2b)), 6)</td>
</tr>
<tr>
<td>I20 M20 M21</td>
<td>target25 = IP + (sign_ext(s &lt;&lt; 20</td>
</tr>
<tr>
<td>M22 M23</td>
<td>target25 = IP + (sign_ext(s &lt;&lt; 20</td>
</tr>
<tr>
<td>M34</td>
<td>il = sol</td>
</tr>
<tr>
<td></td>
<td>o = sof – sol</td>
</tr>
<tr>
<td></td>
<td>r = sor &lt;&lt; 3</td>
</tr>
<tr>
<td>M39 M40</td>
<td>imm2 = i2b</td>
</tr>
<tr>
<td>M44</td>
<td>imm24 = i &lt;&lt; 23</td>
</tr>
<tr>
<td>B1 B2 B3</td>
<td>target25 = IP + (sign_ext(s &lt;&lt; 20</td>
</tr>
<tr>
<td>B6</td>
<td>target25 = IP + (sign_ext(s &lt;&lt; 20</td>
</tr>
<tr>
<td></td>
<td>tag13 = IP + (sign_ext(t2a &lt;&lt; 7</td>
</tr>
<tr>
<td>B7</td>
<td>tag13 = IP + (sign_ext(t2a &lt;&lt; 7</td>
</tr>
<tr>
<td>B9</td>
<td>imm21 = i &lt;&lt; 20</td>
</tr>
<tr>
<td>F5</td>
<td>fclass9 = fclass7c &lt;&lt; 2</td>
</tr>
<tr>
<td>F12</td>
<td>amask7 = amask7b</td>
</tr>
<tr>
<td></td>
<td>omask7 = omask7c</td>
</tr>
<tr>
<td>F14</td>
<td>target25 = IP + (sign_ext(s &lt;&lt; 20</td>
</tr>
<tr>
<td>F15 F16</td>
<td>imm21 = i &lt;&lt; 20</td>
</tr>
<tr>
<td>X1 X5</td>
<td>imm62 = imm41 &lt;&lt; 21</td>
</tr>
<tr>
<td>X2</td>
<td>imm64 = i &lt;&lt; 63</td>
</tr>
<tr>
<td>X3 X4</td>
<td>target64 = IP + ((i &lt;&lt; 59</td>
</tr>
</tbody>
</table>
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 implied 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>
<thead>
<tr>
<th>Table 5-1. Semantics of Dependency Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Semantics of Dependency Code</strong></td>
</tr>
<tr>
<td>--------------------------------------</td>
</tr>
<tr>
<td>instr</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>implied</td>
</tr>
</tbody>
</table>
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 unpredictatable-instructions class have no qualifying predicate and thus always read or write their resources (except as noted).
- An instruction of type \texttt{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 \texttt{mov-to-PR} writes only those PRs as indicated by the immediate mask encoded in the instruction.
- A \texttt{st8.spill} only writes AR[UNAT]{\textit{X}} where \textit{X} equals the value in bits 8:3 of the store’s data address. A \texttt{id8.fill} instruction only reads AR[UNAT]{\textit{Y}} where \textit{Y} equals the value in bits 8:3 of the load’s data address.
- Instructions of type \texttt{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 \texttt{mod-sched-brs-counted} always read and write AR[LC], even if they do not change its value.
- For instructions of type \texttt{pr-or-writers} or \texttt{pr-and-writers}, if their completer is \texttt{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 \texttt{and.orcm}, then only the second target predicate is an or-compare and the first target predicate is an and-compare.
- \texttt{rum} and \texttt{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

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Readers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALAT</td>
<td>chk.a.clr, mem-readers-alat, mem-writers, invala-all</td>
<td>mem-readers-alat, mem-writers, chk-a, invala.e</td>
<td>none</td>
</tr>
<tr>
<td>AR[BSP]</td>
<td>br.call, brl.call, br.ret, cover, mov-to-AR-BSPSTORE, rfi</td>
<td>br.call, brl.call, br.ia, br.ret, cover, flushrs, loadsrs, mov-from-AR-BSP, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[BSPSTORE]</td>
<td>alloc, loadrs, flushrs, mov-to-AR-BSPSTORE</td>
<td>alloc, br.ia, flushrs, mov-from-AR-BSPSTORE</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[CCV]</td>
<td>mov-to-AR-CCV</td>
<td>br.ia, cmpxchg, mov-from-AR-CCV</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[CFLG]</td>
<td>mov-to-AR-CFLG</td>
<td>br.ia, mov-from-AR-CFLG</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[CSD]</td>
<td>ld16, mov-to-AR-CSD</td>
<td>br.ia, cmp8xchg16, mov-from-AR-CSD, st16</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[EC]</td>
<td>mod-sched-brs, br.ret, mov-to-AR-EC</td>
<td>br.call, brl.call, br.ia, mod-sched-brs, mov-from-AR-EC</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[EFLAG]</td>
<td>mov-to-AR-EFLAG</td>
<td>br.ia, mov-from-AR-EFLAG</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FCR]</td>
<td>mov-to-AR-FCR</td>
<td>br.ia, mov-from-AR-FCR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FDR]</td>
<td>mov-to-AR-FDR</td>
<td>br.ia, mov-from-AR-FDR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FI R]</td>
<td>mov-to-AR-FIR</td>
<td>br.ia, mov-from-AR-FIR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf0.controls</td>
<td>mov-to-AR-FPSR, fsetc.s0</td>
<td>br.ia, fp-arith-s0, fcmp-s0, fpcmp-s0, fsetc, mov-from-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf1.controls</td>
<td>mov-to-AR-FPSR, fsetc.s1</td>
<td>br.ia, fp-arith-s1, fcmp-s1, fpcmp-s1, mov-from-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf2.controls</td>
<td>mov-to-AR-FPSR, fsetc.s2</td>
<td>br.ia, fp-arith-s2, fcmp-s2, fpcmp-s2, mov-from-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf3.controls</td>
<td>mov-to-AR-FPSR, fsetc.s3</td>
<td>br.ia, fp-arith-s3, fcmp-s3, fpcmp-s3, mov-from-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf0.flags</td>
<td>fp-arith-s0, fclrf.s0, fcmp-s0, fpcmp-s0, mov-from-AR-FPSR</td>
<td>br.ia, fchkf, mov-from-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf1.flags</td>
<td>fp-arith-s1, fclrf.s1, fcmp-s1, fpcmp-s1, mov-from-AR-FPSR</td>
<td>br.ia, fchkf.s1, mov-from-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf2.flags</td>
<td>fp-arith-s2, fclrf.s2, fcmp-s2, fpcmp-s2, mov-from-AR-FPSR</td>
<td>br.ia, fchkf.s2, mov-from-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf3.flags</td>
<td>fp-arith-s3, fclrf.s3, fcmp-s3, fpcmp-s3, mov-from-AR-FPSR</td>
<td>br.ia, fchkf.s3, mov-from-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].flags</td>
<td>br.ia, fp-arith, fchkf, fcmp, fpcmp, mov-from-AR-FPSR</td>
<td>impliedF</td>
<td></td>
</tr>
<tr>
<td>AR[FPSR].rv</td>
<td>mov-to-AR-FPSR</td>
<td>br.ia, fp-arith, fchkf, fcmp, fpcmp, mov-from-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FSR]</td>
<td>mov-to-AR-FSR</td>
<td>br.ia, mov-from-AR-FSR</td>
<td>impliedF</td>
</tr>
</tbody>
</table>
### Table 5-2. RAW Dependencies Organized by Resource (Continued)

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Readers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR[ITC]</td>
<td>mov-to-AR-ITC</td>
<td>br.ia, mov-from-AR-ITC</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[K%], % in 0 - 7</td>
<td>mov-to-AR-K(^1)</td>
<td>br.ia, mov-from-AR-K(^1)</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[PFS]</td>
<td>br.call, brl.call</td>
<td>alloc, br.ia, br.ret, epc, mov-from-AR-PFS</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>mov-to-AR-PFS</td>
<td>alloc, br.ia, epc, mov-from-AR-PFS</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>br.ret</td>
<td>none</td>
</tr>
<tr>
<td>AR[RNAT]</td>
<td>alloc, flushrs, loadrs, mov-to-AR-RNAT, mov-to-AR-BSPSTORE</td>
<td>alloc, br.ia, flushrs, loadrs, mov-from-AR-RNAT</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[RUC]</td>
<td>mov-to-AR-RUC</td>
<td>br.ia, mov-from-AR-RUC</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[SSD]</td>
<td>mov-to-AR-SSD</td>
<td>br.ia, mov-from-AR-SSD</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[UNAT]%, % in 0 - 63</td>
<td>mov-to-AR-UNAT, st8.spill</td>
<td>br.ia, ld8.fill, mov-from-AR-UNAT</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR%, % in 48-63, 112-127</td>
<td>mov-to-AR-ig(^1)</td>
<td>br.ia, mov-from-AR-ig(^1)</td>
<td>impliedF</td>
</tr>
<tr>
<td>BR%, % in 0 - 7</td>
<td>br.call(^1), brl.call(^1)</td>
<td>indirect-brs(^1), indirect-brp(^1), mov-from-BR(^1)</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>mov-to-BR(^1)</td>
<td>indirect-brs(^1), indirect-brp(^1), mov-from-BR(^1)</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>none</td>
<td>impliedF</td>
</tr>
<tr>
<td>CFM</td>
<td>mod-sched-brs</td>
<td>mod-sched-brs</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cover, alloc, rfi, loadrs, br.ret, br.call, brl.call</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cfm-readers(^2)</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>br.call, brl.call, br.ret, clrmb, cover, rfi</td>
<td>cfm-readers</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>alloc</td>
<td>cfm-readers</td>
<td>none</td>
</tr>
<tr>
<td>CPUID#</td>
<td>none</td>
<td>mov-from-IND-CPUID(^4)</td>
<td>specific</td>
</tr>
<tr>
<td>Resource Name</td>
<td>Writers</td>
<td>Readers</td>
<td>Semantics of Dependency</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>---------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>CR[IFA]</td>
<td>mov-to-CR-IFA</td>
<td>itc.i, itc.d,iltr.i,ltr.d</td>
<td>implied</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-CR-IFA</td>
<td>data</td>
</tr>
<tr>
<td>CR[IFS]</td>
<td>mov-to-CR-IFS</td>
<td>mov-from-CR-IFS</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rfi</td>
<td>implied</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cover rfi, mov-from-CR-IFS</td>
<td>implied</td>
</tr>
<tr>
<td>CR[IIB%], % in 0 - 1</td>
<td>mov-to-CR-IIB</td>
<td>mov-from-CR-IIB</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rfi</td>
<td>implied</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rfi</td>
<td>implied</td>
</tr>
<tr>
<td>CR[IRR%], % in 0 - 3</td>
<td>mov-from-CR-IVR</td>
<td>mov-from-CR-IRR</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR[LRR%], % in 0 - 1</td>
<td>mov-to-CR-LRR</td>
<td>mov-from-CR-LRR</td>
<td>data</td>
</tr>
</tbody>
</table>
### Table 5-2. RAW Dependencies Organized by Resource (Continued)

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Readers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mov-to-PSR-1(^\text{17}), ssm(^\text{17})</td>
<td>SC Section 5.8.3.3, &quot;Task Priority Register (TPR – CR66)&quot; on page 2:123</td>
</tr>
<tr>
<td>CR%, % in 3, 5-7, 10-15, 18, 28-63, 75-79, 82-127</td>
<td>none</td>
<td>mov-from-CR-rv(^1)</td>
<td>none</td>
</tr>
<tr>
<td>DBR#</td>
<td>mov-to-IND-DBR(^3)</td>
<td>mov-from-IND-DBR(^3)</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>probe-all, lfetch-all, mem-readers, mem-writers</td>
<td>data</td>
</tr>
<tr>
<td>DTC</td>
<td>ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d</td>
<td>mem-readers, mem-writers, non-access</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>itc.i, itc.d, itr.i, itr.d</td>
<td>ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d</td>
<td>ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>itc.i, itc.d, itr.i, itr.d</td>
<td>itc.i, itc.d, itr.i, itr.d</td>
<td>impliedF</td>
</tr>
<tr>
<td>DTC_LIMIT*</td>
<td>ptc.g, ptc.ga</td>
<td>ptc.g, ptc.ga</td>
<td>impliedF</td>
</tr>
<tr>
<td>DTR</td>
<td>itr.d</td>
<td>mem-readers, mem-writers, non-access</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>ptr.d</td>
<td>mem-readers, mem-writers, non-access</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ptc.g, ptc.ga, ptc.l, ptr.d, itr.d</td>
<td>impliedF</td>
</tr>
<tr>
<td>FR%, % in 0 - 1</td>
<td>none</td>
<td>fr-readers(^1)</td>
<td>none</td>
</tr>
<tr>
<td>FR%, % in 2 - 127</td>
<td>fr-writers(^5), ld-c(^1), ldtp-c(^1)</td>
<td>fr-readers(^1)</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>ld-c(^1), ldtp-c(^1)</td>
<td>fr-readers(^1)</td>
<td>none</td>
</tr>
<tr>
<td>GR0</td>
<td>none</td>
<td>gr-readers(^1)</td>
<td>none</td>
</tr>
<tr>
<td>GR%, % in 1 - 127</td>
<td>id-c(^1),(^13)</td>
<td>gr-readers(^1)</td>
<td>none</td>
</tr>
<tr>
<td>IBR#</td>
<td>mov-to-IND-IBR(^3)</td>
<td>mov-from-IND-IBR(^3)</td>
<td>impliedF</td>
</tr>
<tr>
<td>InService*</td>
<td>mov-to-CR-EOI</td>
<td>mov-from-CR-IVR</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>mov-from-CR-IVR</td>
<td>mov-to-CR-EOI</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>mov-to-CR-EOI</td>
<td>mov-to-CR-EOI</td>
<td>impliedF</td>
</tr>
<tr>
<td>IP</td>
<td>all</td>
<td>all</td>
<td>none</td>
</tr>
<tr>
<td>ITC</td>
<td>ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d</td>
<td>epc, vmsw</td>
<td>instr</td>
</tr>
<tr>
<td></td>
<td>itc.i, itc.d, itr.i, itr.d</td>
<td>itc.i, ptr.d, ptr.e, ptc.g, ptc.ga, ptc.l</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>ptr.i, ptr.d, ptc.e, ptc.g, ptc.ga, ptc.l</td>
<td>epc, vmsw</td>
<td>instr</td>
</tr>
<tr>
<td></td>
<td>itc.i, itc.d, itr.i, itr.d</td>
<td>itc.d, itc.i, itr.d, ptr.i, ptr.d, ptr.i, ptc.g, ptc.ga, ptc.l</td>
<td>impliedF</td>
</tr>
<tr>
<td>ITC_LIMIT*</td>
<td>ptc.g, ptc.ga</td>
<td>ptc.g, ptc.ga</td>
<td>impliedF</td>
</tr>
</tbody>
</table>
Table 5-2. RAW Dependencies Organized by Resource (Continued)

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Readers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITR</td>
<td>itr.i, ptr.i</td>
<td>itr.i, ptr.i, ptc.g, ptc.ga, ptc.l, ptr.i</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>epc, vmsw</td>
<td>instr</td>
</tr>
<tr>
<td></td>
<td>ptr.i</td>
<td>ptr.i</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ptc.g, ptc.ga, ptc.l, ptr.i</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>epc, vmsw</td>
<td>instr</td>
</tr>
<tr>
<td>memory</td>
<td>mem-writers</td>
<td>mem-readers</td>
<td>none</td>
</tr>
<tr>
<td>PKR#</td>
<td>mov-to-IND-PKR</td>
<td>mem-readers, mem-writers,</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-IND-PKR, probe-all</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-to-IND-PKR</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-IND-PKR</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-to-IND-PKR</td>
<td>impliedF</td>
</tr>
<tr>
<td>PMC#</td>
<td>mov-to-IND-PMC</td>
<td>mov-from-IND-PMC</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-IND-PMD</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SC Section 7.2.1, “Generic</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Performance Counter</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Registers” for</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PMC[0].fr on page 2:156</td>
<td></td>
</tr>
<tr>
<td>PMD#</td>
<td>mov-to-IND-PMD</td>
<td>mov-from-IND-PMD</td>
<td>impliedF</td>
</tr>
<tr>
<td>PRO</td>
<td>pr-writers</td>
<td>pr-writers, pr-readers-br</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pr-readers-nobr-nomovpr,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-PR, mov-to-PR</td>
<td></td>
</tr>
<tr>
<td>PR%, % in 1 - 15</td>
<td>pr-writers,</td>
<td>pr-readers-nobr-nomovpr,</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>mov-to-PR-allreg</td>
<td>mov-from-PR, mov-to-PR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pr-writers-fp</td>
<td>pr-readers-br</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>pr-writers-int</td>
<td>pr-readers-br</td>
<td>none</td>
</tr>
<tr>
<td>PR%, % in 16 - 62</td>
<td>pr-writers,</td>
<td>pr-readers-nobr-nomovpr,</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>mov-to-PR-allreg,</td>
<td>mov-from-PR, mov-to-PR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov-to-PR-rotreg</td>
<td>pr-readers-br</td>
<td>none</td>
</tr>
<tr>
<td>PR63</td>
<td>mod-sched-brs,</td>
<td>pr-readers-nobr-nomovpr,</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>pr-writers,</td>
<td>mov-from-PR, mov-to-PR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov-to-PR-allreg,</td>
<td>pr-readers-br</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>mov-to-PR-rotreg</td>
<td>pr-readers-br</td>
<td></td>
</tr>
<tr>
<td>Resource Name</td>
<td>Writers</td>
<td>Readers</td>
<td>Semantics of Dependency</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>---------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>PSR.ac</td>
<td>user-mask-writers-partial, mov-to-PSR-um</td>
<td>mem-readers, mem-writers</td>
<td>implied</td>
</tr>
<tr>
<td></td>
<td>sys-mask-writers-partial, mov-to-PSR-l</td>
<td>mem-readers, mem-writers</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>user-mask-writers-partial, mov-to-PSR-um, sys-mask-writers-partial, mov-to-PSR-l</td>
<td>mov-from-PSR, mov-from-PSR-um</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>rfi</td>
<td>mem-readers, mem-writers, mov-from-PSR, mov-from-PSR-um</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.be</td>
<td>user-mask-writers-partial, mov-to-PSR-um</td>
<td>mem-readers, mem-writers</td>
<td>implied</td>
</tr>
<tr>
<td></td>
<td>sys-mask-writers-partial, mov-to-PSR-l</td>
<td>mem-readers, mem-writers</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>user-mask-writers-partial, mov-to-PSR-um, sys-mask-writers-partial, mov-to-PSR-l</td>
<td>mov-from-PSR, mov-from-PSR-um</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>rfi</td>
<td>mem-readers, mem-writers, mov-from-PSR, mov-from-PSR-um</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.bn</td>
<td>bsw, rfi</td>
<td>gr-readers, gr-writers</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.da</td>
<td>rfi</td>
<td>mem-readers, lfetch-all, mem-writers, probe-fault</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.db</td>
<td>mov-to-PSR-l</td>
<td>lfetch-all, mem-readers, mem-writers, probe-fault</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>rfi</td>
<td>lfetch-all, mem-readers, mem-writers, probe-fault</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.dd</td>
<td>rfi</td>
<td>lfetch-all, mem-readers, probe-fault, mem-writers</td>
<td>impliedF</td>
</tr>
</tbody>
</table>
## Table 5-2. RAW Dependencies Organized by Resource (Continued)

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Readers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSR.dfh</td>
<td>sys-mask-writers-partial(^7), mov-to-PSR-I</td>
<td>fr-readers(^5), fr-writers(^5)</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>rfi</td>
<td></td>
<td>mov-from-PSR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fr-readers(^5), fr-writers(^5), mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.dfl</td>
<td>sys-mask-writers-partial(^7), mov-to-PSR-I</td>
<td>fr-writers(^5), fr-readers(^5)</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>rfi</td>
<td></td>
<td>mov-from-PSR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fr-writers(^5), fr-readers(^5), mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.di</td>
<td>sys-mask-writers-partial(^7), mov-to-PSR-I</td>
<td>br.ia</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>rfi</td>
<td></td>
<td>mov-from-PSR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>br.ia, mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.dt</td>
<td>sys-mask-writers-partial(^7), mov-to-PSR-I</td>
<td>mem-readers, mem-writers, non-access</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>rfi</td>
<td></td>
<td>mov-from-PSR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem-readers, mem-writers, non-access, mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.ed</td>
<td>rfi</td>
<td>ifetch-all, mem-readers-spec</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.i</td>
<td>sys-mask-writers-partial(^7), mov-to-PSR-I, rfi</td>
<td>mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.la</td>
<td>rfi</td>
<td>all</td>
<td>none</td>
</tr>
<tr>
<td>PSR.ic</td>
<td>sys-mask-writers-partial(^7), mov-to-PSR-I</td>
<td>mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>rfi</td>
<td>cover, ltc.i, ltc.d, ltr.i, ltr.d, mov-from-interruption-CR, mov-to-interruption-CR</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-PSR, cover, ltc.i, ltc.d, ltr.i, ltr.d, mov-from-interruption-CR, mov-to-interruption-CR</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.id</td>
<td>rfi</td>
<td>all</td>
<td>none</td>
</tr>
<tr>
<td>PSR.is</td>
<td>br.ia, rfi</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>PSR.it</td>
<td>rfi</td>
<td>branches, mov-from-PSR, chk, epc, fchkf, vmsw</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.lp</td>
<td>mov-to-PSR-I</td>
<td>mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td>rfi</td>
<td>br.ret</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-PSR, br.ret</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.mc</td>
<td>rfi</td>
<td>mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.mfh</td>
<td>fr-writers(^5), user-mask-writers-partial(^7), mov-to-PSR-um, sys-mask-writers-partial(^7), mov-to-PSR-I, rfi</td>
<td>mov-from-PSR-um, mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.mfl</td>
<td>fr-writers(^5), user-mask-writers-partial(^7), mov-to-PSR-um, sys-mask-writers-partial(^7), mov-to-PSR-I, rfi</td>
<td>mov-from-PSR-um, mov-from-PSR</td>
<td>impliedF</td>
</tr>
</tbody>
</table>
Table 5-2. RAW Dependencies Organized by Resource (Continued)

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Readers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSR.pk</td>
<td>sys-mask-writers-partial, mov-to-PSR-l, rfi</td>
<td>Ifetch-all, mem-readers, mem-writers, probe-all, mov-from-PSR</td>
<td>impliedF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-PSR</td>
<td></td>
</tr>
<tr>
<td>PSR.pp</td>
<td>sys-mask-writers-partial, mov-to-PSR-l, rfi</td>
<td>mov-from-PSR</td>
<td></td>
</tr>
<tr>
<td>PSR.rf</td>
<td>rfi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSR.rt</td>
<td>mov-to-PSR-l</td>
<td>mov-from-PSR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>alloc, flushrs, loadrs</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-PSR, alloc, flushrs, loadrs</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.si</td>
<td>sys-mask-writers-partial, mov-to-PSR-l</td>
<td>mov-from-PSR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-AR-ITC, mov-from-AR-RUC</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rfi</td>
<td>mov-from-AR-ITC, mov-from-AR-RUC, mov-from-PSR</td>
</tr>
<tr>
<td>PSR.sp</td>
<td>sys-mask-writers-partial, mov-to-PSR-l</td>
<td>mov-from-PSR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-IND-PMD, mov-to-PSR-um, rum, sum</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rfi</td>
<td>mov-from-IND-PMD, mov-from-PSR, mov-to-PSR-um, rum, sum</td>
</tr>
<tr>
<td>PSR.ss</td>
<td>rfi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSR.tb</td>
<td>mov-to-PSR-l</td>
<td>branches, chk, fchkf</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-PSR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rfi</td>
<td>branches, chk, fchkf, mov-from-PSR</td>
</tr>
<tr>
<td>PSR.up</td>
<td>user-mask-writers-partial, mov-to-PSR-um, sys-mask-writers-partial, mov-to-PSR-l, rfi</td>
<td>mov-from-PSR-um, mov-from-PSR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rfi</td>
<td>mem-readers, mem-writers, mov-from-AR-RUC, mov-from-IND-CPUID, mov-to-AR-ITC, mov-to-AR-RUC, priv-ops vmsw, cover, thash, ttag</td>
</tr>
<tr>
<td>RR#</td>
<td>mov-to-IND-RR</td>
<td>mem-readers, mem-writers, iftc.i, iftc.d, !itr.i, !itr.d, non-access, ptc.g, ptc.ga, ptc.i, ptr.i, ptr.d, thash, tag</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov-from-IND-RR</td>
<td></td>
</tr>
<tr>
<td>RSE</td>
<td>rse-writers</td>
<td>rse-readers</td>
<td></td>
</tr>
</tbody>
</table>
### 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>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALAT</td>
<td>mem-readers-alat, mem-writers, chk.a.clr, invala-all</td>
<td>none</td>
</tr>
<tr>
<td>AR[BSP]</td>
<td>br.call, brl.call, br.ret, cover, mov-to-AR-BSPSTORE, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[BSPSTORE]</td>
<td>alloc, loadrs, flushrs, mov-to-AR-BSPSTORE</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[CCV]</td>
<td>mov-to-AR-CCV</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[CFLG]</td>
<td>mov-to-AR-CFLG</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[CSD]</td>
<td>ld16, mov-to-AR-CSD</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[EC]</td>
<td>br.ret, mod-sched-brs, mov-to-AR-EC</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[EFLAG]</td>
<td>mov-to-AR-EFLAG</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FCR]</td>
<td>mov-to-AR-FCR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FDR]</td>
<td>mov-to-AR-FDR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FIR]</td>
<td>mov-to-AR-FIR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf0.controls</td>
<td>mov-to-AR-FPSR, fsetc.s0</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf1.controls</td>
<td>mov-to-AR-FPSR, fsetc.s1</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf2.controls</td>
<td>mov-to-AR-FPSR, fsetc.s2</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf3.controls</td>
<td>mov-to-AR-FPSR, fsetc.s3</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf0.flags</td>
<td>fp-arith-s0, fcmp-s0, fpcomp-s0</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>fcifr.s0, fcmp-s0, fp-arith-s0, fpcomp-s0, mov-to-AR-FPSR</td>
<td>fcifr.s0, mov-to-AR-FPSR</td>
</tr>
<tr>
<td></td>
<td>fcifr.s1, mov-to-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf1.flags</td>
<td>fp-arith-s1, fcmp-s1, fpcomp-s1</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>fcifr.s1, fcmp-s1, fp-arith-s1, fpcomp-s1, mov-to-AR-FPSR</td>
<td>fcifr.s1, mov-to-AR-FPSR</td>
</tr>
<tr>
<td></td>
<td>fcifr.s2, mov-to-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf2.flags</td>
<td>fp-arith-s2, fcmp-s2, fpcomp-s2</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>fcifr.s2, fcmp-s2, fp-arith-s2, fpcomp-s2, mov-to-AR-FPSR</td>
<td>fcifr.s2, mov-to-AR-FPSR</td>
</tr>
<tr>
<td></td>
<td>fcifr.s3, mov-to-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].sf3.flags</td>
<td>fp-arith-s3, fcmp-s3, fpcomp-s3</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>fcifr.s3, fcmp-s3, fp-arith-s3, fpcomp-s3, mov-to-AR-FPSR</td>
<td>fcifr.s3, mov-to-AR-FPSR</td>
</tr>
<tr>
<td></td>
<td>fcifr.s4, mov-to-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].rv</td>
<td>mov-to-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FPSR].traps</td>
<td>mov-to-AR-FPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[FIR]</td>
<td>mov-to-AR-FIR</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[ITC]</td>
<td>mov-to-AR-ITC</td>
<td>impliedF</td>
</tr>
</tbody>
</table>
### Table 5-3. WAW Dependencies Organized by Resource (Continued)

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR[K%], % in 0 - 7</td>
<td>mov-to-AR-K&lt;sup&gt;1&lt;/sup&gt;</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[LC]</td>
<td>mod-sched-brs-counted, mov-to-AR-LC</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[PFS]</td>
<td>br.call, brl.call</td>
<td>none</td>
</tr>
<tr>
<td>AR[RSC]</td>
<td>mov-to-AR-RSC</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[RUC]</td>
<td>mov-to-AR-RUC</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[SSD]</td>
<td>mov-to-AR-SSD</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[UNAT]%&lt;sup&gt;2&lt;/sup&gt;, % in 0 - 63</td>
<td>mov-to-AR-UNAT, st8.spill</td>
<td>impliedF</td>
</tr>
<tr>
<td>BR%, % in 0 - 7</td>
<td>br.call&lt;sup&gt;1&lt;/sup&gt;, brl.call&lt;sup&gt;1&lt;/sup&gt;, mov-to-BR&lt;sup&gt;1&lt;/sup&gt;</td>
<td>impliedF</td>
</tr>
<tr>
<td>CFM</td>
<td>mod-sched-brs, br.call, brl.call, br.ret, alloc, crrrb, cover, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>CPUID#</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>CR[CMCV]</td>
<td>mov-to-CR-CMCV</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[DCR]</td>
<td>mov-to-CR-DCR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IFA]</td>
<td>mov-to-CR-IFA</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IFS]</td>
<td>mov-to-CR-IFS, cover</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IHA]</td>
<td>mov-to-CR-IHA</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IIB%], % in 0 - 1</td>
<td>mov-to-CR-IIB</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IIM]</td>
<td>mov-to-CR-IIM</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IIP]</td>
<td>mov-to-CR-IIP</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IIPA]</td>
<td>mov-to-CR-IIPA</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IPSR]</td>
<td>mov-to-CR-IPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IRR%], % in 0 - 3</td>
<td>mov-from-CR-IVR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[ISR]</td>
<td>mov-to-CR-ISR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[TIR]</td>
<td>mov-to-CR-TIR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[ITM]</td>
<td>mov-to-CR-ITM</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[ITO]</td>
<td>mov-to-CR-ITO</td>
<td>impliedF</td>
</tr>
</tbody>
</table>

<sup>1</sup>Schedulersられます。すなわち、動的スケジューリング

<sup>2</sup>Resource Name Writers Semantics of Dependency

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR[K%], % in 0 - 7</td>
<td>mov-to-AR-K&lt;sup&gt;1&lt;/sup&gt;</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[LC]</td>
<td>mod-sched-brs-counted, mov-to-AR-LC</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[PFS]</td>
<td>br.call, brl.call</td>
<td>none</td>
</tr>
<tr>
<td>AR[RSC]</td>
<td>mov-to-AR-RSC</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[RUC]</td>
<td>mov-to-AR-RUC</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[SSD]</td>
<td>mov-to-AR-SSD</td>
<td>impliedF</td>
</tr>
<tr>
<td>AR[UNAT]%&lt;sup&gt;2&lt;/sup&gt;, % in 0 - 63</td>
<td>mov-to-AR-UNAT, st8.spill</td>
<td>impliedF</td>
</tr>
<tr>
<td>BR%, % in 0 - 7</td>
<td>br.call&lt;sup&gt;1&lt;/sup&gt;, brl.call&lt;sup&gt;1&lt;/sup&gt;, mov-to-BR&lt;sup&gt;1&lt;/sup&gt;</td>
<td>impliedF</td>
</tr>
<tr>
<td>CFM</td>
<td>mod-sched-brs, br.call, brl.call, br.ret, alloc, crrrb, cover, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>CPUID#</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>CR[CMCV]</td>
<td>mov-to-CR-CMCV</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[DCR]</td>
<td>mov-to-CR-DCR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IFA]</td>
<td>mov-to-CR-IFA</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IFS]</td>
<td>mov-to-CR-IFS, cover</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IHA]</td>
<td>mov-to-CR-IHA</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IIB%], % in 0 - 1</td>
<td>mov-to-CR-IIB</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IIM]</td>
<td>mov-to-CR-IIM</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IIP]</td>
<td>mov-to-CR-IIP</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IIPA]</td>
<td>mov-to-CR-IIPA</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IPSR]</td>
<td>mov-to-CR-IPSR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IRR%], % in 0 - 3</td>
<td>mov-from-CR-IVR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[ISR]</td>
<td>mov-to-CR-ISR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[TIR]</td>
<td>mov-to-CR-TIR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[ITM]</td>
<td>mov-to-CR-ITM</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[ITO]</td>
<td>mov-to-CR-ITO</td>
<td>impliedF</td>
</tr>
</tbody>
</table>
Table 5-3.  WAW Dependencies Organized by Resource (Continued)

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR[ITV]</td>
<td>mov-to-CR-ITV</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IVA]</td>
<td>mov-to-CR-IVA</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[IVR]</td>
<td>none</td>
<td>SC</td>
</tr>
<tr>
<td>CR[LID]</td>
<td>mov-to-CR-LID</td>
<td>SC</td>
</tr>
<tr>
<td>CR[LRR%], % in 0 - 1</td>
<td>mov-to-CR-LRR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[PMV]</td>
<td>mov-to-CR-PMV</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[PTA]</td>
<td>mov-to-CR-PTA</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR[TPR]</td>
<td>mov-to-CR-TPR</td>
<td>impliedF</td>
</tr>
<tr>
<td>CR%, % in 3, 5-7, 10-15, 18, 28-63, 75-79, 82-127</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>DBR#</td>
<td>mov-to-IND-DBR³</td>
<td>impliedF</td>
</tr>
<tr>
<td>DTC</td>
<td>ptc.e, ptc.g, ptc.ga, ptr.i, ptr.d</td>
<td>none</td>
</tr>
<tr>
<td>DTC_LIMIT*</td>
<td>ptc.g, ptc.ga</td>
<td>impliedF</td>
</tr>
<tr>
<td>DTR</td>
<td>itc.i, itc.d, itr.i, itr.d</td>
<td>impliedF</td>
</tr>
<tr>
<td>FR%, % in 0 - 1</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>FR%, % in 2 - 127</td>
<td>fr-writers, ldf-c, ldfp-c</td>
<td>impliedF</td>
</tr>
<tr>
<td>GR0</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>GR%, % in 1 - 127</td>
<td>ld-c, gr-writers</td>
<td>impliedF</td>
</tr>
<tr>
<td>IBR#</td>
<td>mov-to-IND-IBR³</td>
<td>impliedF</td>
</tr>
<tr>
<td>InService*</td>
<td>mov-to-CR-EOI, mov-from-CR-IVR</td>
<td>SC</td>
</tr>
<tr>
<td>IP</td>
<td>all</td>
<td>none</td>
</tr>
<tr>
<td>ITC</td>
<td>ptc.e, ptc.g, ptc.ga, ptr.i, ptr.d</td>
<td>none</td>
</tr>
<tr>
<td>ITR</td>
<td>itr.i, ptr.i</td>
<td>impliedF</td>
</tr>
<tr>
<td>memory</td>
<td>mem-writers</td>
<td>none</td>
</tr>
<tr>
<td>PKR#</td>
<td>mov-to-IND-PKR³</td>
<td>none</td>
</tr>
<tr>
<td>PMC#</td>
<td>mov-to-IND-PMC³</td>
<td>impliedF</td>
</tr>
<tr>
<td>PMD#</td>
<td>mov-to-IND-PMD³</td>
<td>impliedF</td>
</tr>
<tr>
<td>PR0</td>
<td>None</td>
<td>none</td>
</tr>
</tbody>
</table>
Table 5-3. WAW Dependencies Organized by Resource (Continued)

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Writers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR%, % in 1 - 15</td>
<td>pr-and-writers, pr-or-writers</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>pr-unc-writers-fp, pr-unc-writers-int, pr-norm-writers-fp, pr-norm-writers-int, pr-and-writers, mov-to-PR-allreg</td>
<td>impliedF</td>
</tr>
<tr>
<td>PR%, % in 16 - 62</td>
<td>pr-and-writers, pr-or-writers</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>pr-unc-writers-fp, pr-unc-writers-int, pr-norm-writers-fp, pr-norm-writers-int, pr-and-writers, mov-to-PR-allreg, mov-to-PR-rotreg</td>
<td>impliedF</td>
</tr>
<tr>
<td>PR63</td>
<td>pr-and-writers, pr-or-writers</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>mod-sched-brs, pr-unc-writers-fp, pr-unc-writers-int, pr-norm-writers-fp, pr-norm-writers-int, pr-and-writers, mov-to-PR-allreg, mov-to-PR-rotreg</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.ac</td>
<td>user-mask-writers-partial, mov-to-PSR-um, sys-mask-writers-partial, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.be</td>
<td>user-mask-writers-partial, mov-to-PSR-um, sys-mask-writers-partial, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.bn</td>
<td>bsw, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.cpl</td>
<td>epc, br.ret, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.da</td>
<td>rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.db</td>
<td>mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.dd</td>
<td>rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.dfh</td>
<td>sys-mask-writers-partial, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.dll</td>
<td>sys-mask-writers-partial, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.di</td>
<td>sys-mask-writers-partial, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.dt</td>
<td>sys-mask-writers-partial, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.ed</td>
<td>rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.ia</td>
<td>rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.ic</td>
<td>sys-mask-writers-partial, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.id</td>
<td>rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.is</td>
<td>br.ia, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.it</td>
<td>rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.lp</td>
<td>mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.mc</td>
<td>rfi</td>
<td>impliedF</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Readers</th>
<th>Writers</th>
<th>Semantics of Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSR.mfh</td>
<td></td>
<td>fr-writers³</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>user-mask-writers-partial⁷, mov-to-PSR-um, fr-writers⁹, sys-mask-writers-partial⁷, mov-to-PSR-l, rfi</td>
<td>user-mask-writers-partial⁷, mov-to-PSR-um, sys-mask-writers-partial⁷, mov-to-PSR-l, rfi</td>
</tr>
<tr>
<td></td>
<td></td>
<td>impliedF</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.mfl</td>
<td></td>
<td>fr-writers⁷</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>user-mask-writers-partial⁴, mov-to-PSR-um, fr-writers⁹, sys-mask-writers-partial⁰, mov-to-PSR-l, rfi</td>
<td>user-mask-writers-partial⁴, mov-to-PSR-um, sys-mask-writers-partial⁰, mov-to-PSR-l, rfi</td>
</tr>
<tr>
<td></td>
<td></td>
<td>impliedF</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.pk</td>
<td></td>
<td>sys-mask-writers-partial⁴, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.pp</td>
<td></td>
<td>sys-mask-writers-partial⁴, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.ri</td>
<td></td>
<td>rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.rt</td>
<td></td>
<td>mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.si</td>
<td></td>
<td>sys-mask-writers-partial⁴, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.sp</td>
<td></td>
<td>sys-mask-writers-partial⁴, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.ss</td>
<td></td>
<td>rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.tb</td>
<td></td>
<td>mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.up</td>
<td></td>
<td>user-mask-writers-partial⁴, mov-to-PSR-um, sys-mask-writers-partial⁴, mov-to-PSR-l, rfi</td>
<td>impliedF</td>
</tr>
<tr>
<td>PSR.vm</td>
<td></td>
<td>rfi, vmsw</td>
<td>impliedF</td>
</tr>
<tr>
<td>RR⁸</td>
<td></td>
<td>mov-to-IND-RR⁹</td>
<td>impliedF</td>
</tr>
<tr>
<td>RSE</td>
<td></td>
<td>rse-writers⁴</td>
<td>impliedF</td>
</tr>
</tbody>
</table>

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 ld-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 ld-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

<table>
<thead>
<tr>
<th>Class</th>
<th>Events/Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>predicatable-instructions, unpredicatable-instructions</td>
</tr>
<tr>
<td>branches</td>
<td>indirect-brs, ip-rel-brs</td>
</tr>
<tr>
<td>cfms-readers</td>
<td>fr-readers, fr-writers, gr-readers, pr-writers, mod-sched-brs, predicatable-instructions, pr-writers, alloc, br.call, brl.call, br.ret, cover, loadrs, rfi, chk-a, invalae</td>
</tr>
<tr>
<td>chk-a</td>
<td>chk.a.clr, chk.a.nc</td>
</tr>
<tr>
<td>cmpxchg</td>
<td>cmpxchg1, cmpxchg2, cmpxchg4, cmpxchg8, cmp8xchg16</td>
</tr>
<tr>
<td>cpxz</td>
<td>cpxz1, cpxz2</td>
</tr>
<tr>
<td>fcms-s0</td>
<td>fcmp[Field(sf)==s0]</td>
</tr>
<tr>
<td>fcms-s1</td>
<td>fcmp[Field(sf)==s1]</td>
</tr>
<tr>
<td>fcms-s2</td>
<td>fcmp[Field(sf)==s2]</td>
</tr>
<tr>
<td>fcms-s3</td>
<td>fcmp[Field(sf)==s3]</td>
</tr>
<tr>
<td>fetchadd</td>
<td>fetchadd4, fetchadd8</td>
</tr>
<tr>
<td>fp-arith</td>
<td>fadd, famax, famin, fctv.fx, fctv.fux, fctv.xuf, fma, fmax, fmin, fmpy, fms, fnma, fnmop, fnorm, fpmx, fpmn, fpoperator, fpmax, fpval, fpr, fpset, fสิทธิ, fsub</td>
</tr>
<tr>
<td>fp-arith-s0</td>
<td>fp-arith[Field(sf)==s0]</td>
</tr>
<tr>
<td>fp-arith-s1</td>
<td>fp-arith[Field(sf)==s1]</td>
</tr>
<tr>
<td>fp-arith-s2</td>
<td>fp-arith[Field(sf)==s2]</td>
</tr>
<tr>
<td>fp-arith-s3</td>
<td>fp-arith[Field(sf)==s3]</td>
</tr>
<tr>
<td>fp-non-arith</td>
<td>fabs, fand, fandcm, fclass, fctv.xf, fmerge, fmx, fneg, fnegabs, for, fpabs, fpmerge, fpack, fneg, fnegabs, fselect, fswap, fsxt, fxor, xma, xmop</td>
</tr>
<tr>
<td>fpcomp-s0</td>
<td>fpcomp[Field(sf)==s0]</td>
</tr>
<tr>
<td>fpcomp-s1</td>
<td>fpcomp[Field(sf)==s1]</td>
</tr>
<tr>
<td>fpcomp-s2</td>
<td>fpcomp[Field(sf)==s2]</td>
</tr>
<tr>
<td>fpcomp-s3</td>
<td>fpcomp[Field(sf)==s3]</td>
</tr>
<tr>
<td>fr-readers</td>
<td>fp-arith, fp-non-arith, mem-readers-fp, pr-writers-fp, chk.s[Format in (M21)], getf</td>
</tr>
<tr>
<td>fr-writers</td>
<td>fp-arith, fp-non-arith, mem-readers-fp, setf</td>
</tr>
<tr>
<td>gr-readers</td>
<td>gr-readers-writers, mem-readers, mem-readers, chk.s, cmp, cmp4, fc, fctv.fx, fctv.fux, fctv.xuf, fma, fmax, fmin, fmpy, fms, fnma, fnmop, fnorm, fpmx, fpmn, fpoperator, fpmax, fpval, fpr, fpset, fall, fneg, fnegabs, fselect, fswap, fsxt, fxor, xma, xmop</td>
</tr>
<tr>
<td>gr-readers-writers</td>
<td>mov-from-IND, add, addi, addip4, adds, and, andcm, ciz, cpxz, dep, dep[Format in (I13)], extr, mem-readers-int, ld-all-postinc, ld-fetch-postinc, mix, mux, or, pack, padd, pavg, pavgsub, pcmp, pmax, pmin, pmpy, pmpyshr, popcnt, probe-regular, psad, pshl, pshladd, pshr, pshradddq, psb, shl, shladd, shiladd4p, shr, shrp, st-postinc, sub, sxt, tak, thash, tpa, ttag, unpack, xor, xztt</td>
</tr>
<tr>
<td>gr-writers</td>
<td>alloc, dep, getf, gr-readers-writers, mem-readers-int, mov-from-AR, mov-from-PR, mov-from-PSR, mov-from-PSR-um, mov-ip, movl</td>
</tr>
<tr>
<td>indirect-brp</td>
<td>brp[Format in (B7)]</td>
</tr>
<tr>
<td>indirect-brs</td>
<td>br.call[Format in (B5)], br.cond[Format in (B4)], br.ia, br.ret</td>
</tr>
<tr>
<td>invala-all</td>
<td>invala[Format in (M24)], invalae</td>
</tr>
<tr>
<td>ip-rel-brs</td>
<td>mod-sched-brs, br.call[Format in (B3)], brl.call, br.lcond, br.cond[Format in (B1)], br.cloop</td>
</tr>
<tr>
<td>ld</td>
<td>ld1, ld2, ld4, ld8, ld8.fill, ld16</td>
</tr>
<tr>
<td>ld-a</td>
<td>ld1.a, ld2.a, ld4.a, ld8.a</td>
</tr>
</tbody>
</table>

Volume 3: Resource and Dependency Semantics 3:389
### Table 5-5. Instruction Classes (Continued)

<table>
<thead>
<tr>
<th>Class</th>
<th>Events/Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld-all-postinc</td>
<td>ld[Format in {M2 M3}], ldfp[Format in {M12}], ldf[Format in {M7 M8}]</td>
</tr>
<tr>
<td>ld-c</td>
<td>ld-c-nc, ld-c-clr</td>
</tr>
<tr>
<td>ld-c-clr</td>
<td>ld1.c.clr, ld2.c.clr, ld4.c.clr, ld8.c.clr, ld-c-clr-acq</td>
</tr>
<tr>
<td>ld-c-clr-acq</td>
<td>ld1.c.clr.acq, ld2.c.clr.acq, ld4.c.clr.acq, ld8.c.clr.acq</td>
</tr>
<tr>
<td>ld-c-nc</td>
<td>ld1.c.nc, ld2.c.nc, ld4.c.nc, ld8.c.nc</td>
</tr>
<tr>
<td>ld-s</td>
<td>ld1.s, ld2.s, ld4.s, ld8.s</td>
</tr>
<tr>
<td>ld-sa</td>
<td>ld1.sa, ld2.s, ld4.s, ld8.s sa</td>
</tr>
<tr>
<td>ld</td>
<td>ldfs, ldfd, ldfe, ldf8, ldf.fill</td>
</tr>
<tr>
<td>ld-a</td>
<td>ldfs.a, ldfd.a, ldfe.a, ldf8.a</td>
</tr>
<tr>
<td>ldf-c</td>
<td>ldf-c-nc, ldf-c-clr</td>
</tr>
<tr>
<td>ldf-c-clr</td>
<td>ldfs.c.clr, ldfd.c.clr, ldfe.c.clr, ldf8.c.clr</td>
</tr>
<tr>
<td>ldf-c-nc</td>
<td>ldfs.c.nc, ldfd.c.nc, ldfe.c.nc, ldf8.c.nc</td>
</tr>
<tr>
<td>ldf-s</td>
<td>ldfs.s, ldfd.s, ldfe.s, ldf8.s</td>
</tr>
<tr>
<td>ldfp</td>
<td>ldfps, ldfpd, ldfp8</td>
</tr>
<tr>
<td>ldfp-a</td>
<td>ldfps.a, ldfpd.a, ldfp8.a</td>
</tr>
<tr>
<td>ldfp-c</td>
<td>ldfp-c-nc, ldfp-c-clr</td>
</tr>
<tr>
<td>ldfp-c-clr</td>
<td>ldfps.c.clr, ldfpd.c.clr, ldfp8.c.clr</td>
</tr>
<tr>
<td>ldfp-c-nc</td>
<td>ldfps.c.nc, ldfpd.c.nc, ldfp8.c.nc</td>
</tr>
<tr>
<td>ldfp-s</td>
<td>ldfps.s, ldfpd.s, ldfp8.s</td>
</tr>
<tr>
<td>ldfp-sa</td>
<td>ldfps.sa, ldfpd.sa, ldfp8.sa</td>
</tr>
<tr>
<td>lfetch-all</td>
<td>lfetch</td>
</tr>
<tr>
<td>lfetch-fault</td>
<td>lfetch[Field(lftype)==fault]</td>
</tr>
<tr>
<td>lfetch-nofault</td>
<td>lfetch[Field(lftype)==]</td>
</tr>
<tr>
<td>lfetch-postinc</td>
<td>lfetch[Format in {M20 M22}]</td>
</tr>
<tr>
<td>mem-readers</td>
<td>mem-readers-fp, mem-readers-int</td>
</tr>
<tr>
<td>mem-readers-alat</td>
<td>ld-a, ld-fd, ld-fe, ldf8, ldf.fill</td>
</tr>
<tr>
<td>mem-readers-fp</td>
<td>ld, ldfp</td>
</tr>
<tr>
<td>mem-readers-int</td>
<td>cmpxchg, fetchadd, xchg, ld</td>
</tr>
<tr>
<td>mem-readers-spec</td>
<td>ld-s, ld-sa, ldf-s, ldfp-s, ldfp-sa</td>
</tr>
<tr>
<td>mem-writers</td>
<td>mem-writers-fp, mem-writers-int</td>
</tr>
<tr>
<td>mem-writers-fp</td>
<td>stf</td>
</tr>
<tr>
<td>mod-sched-brs</td>
<td>br.cexit, br.clop, br.wexit, br.wtop</td>
</tr>
<tr>
<td>mod-sched-brs-counted</td>
<td>br.cexit, br.clop, br.clop</td>
</tr>
<tr>
<td>mov-from-AR-BSP</td>
<td>mov-from-AR-M[Field(ar3) == BSP]</td>
</tr>
<tr>
<td>mov-from-AR-BSPSTORE</td>
<td>mov-from-AR-M[Field(ar3) == BSPSTORE]</td>
</tr>
<tr>
<td>mov-from-AR-CCV</td>
<td>mov-from-AR-M[Field(ar3) == CCV]</td>
</tr>
<tr>
<td>mov-from-AR-CFLG</td>
<td>mov-from-AR-M[Field(ar3) == CFLG]</td>
</tr>
<tr>
<td>mov-from-AR-CSD</td>
<td>mov-from-AR-M[Field(ar3) == CSD]</td>
</tr>
<tr>
<td>mov-from-AR-EC</td>
<td>mov-from-AR-M[Field(ar3) == EC]</td>
</tr>
<tr>
<td>mov-from-AR-EFLAG</td>
<td>mov-from-AR-M[Field(ar3) == EFLAG]</td>
</tr>
<tr>
<td>mov-from-AR-FCR</td>
<td>mov-from-AR-M[Field(ar3) == FCR]</td>
</tr>
</tbody>
</table>
Table 5-5. Instruction Classes (Continued)

<table>
<thead>
<tr>
<th>Class</th>
<th>Events/Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov-from-AR-FDR</td>
<td>mov-from-AR-M[Field(ar3) == FDR]</td>
</tr>
<tr>
<td>mov-from-AR-FIR</td>
<td>mov-from-AR-M[Field(ar3) == FIR]</td>
</tr>
<tr>
<td>mov-from-AR-FPSR</td>
<td>mov-from-AR-M[Field(ar3) == FPSR]</td>
</tr>
<tr>
<td>mov-from-AR-FCR</td>
<td>mov-from-AR-M[Field(ar3) == FSR]</td>
</tr>
<tr>
<td>mov-from-AR-I</td>
<td>mov_ar[Format in {I28}]</td>
</tr>
<tr>
<td>mov-from-AR-ig</td>
<td>mov-from-AR-IM[Field(ar3) in (48-63 112-127)]</td>
</tr>
<tr>
<td>mov-from-AR-IM</td>
<td>mov_ar[Format in {I28 M31}]</td>
</tr>
<tr>
<td>mov-from-AR-ITC</td>
<td>mov-from-AR-M[Field(ar3) == ITC]</td>
</tr>
<tr>
<td>mov-from-AR-K</td>
<td>mov-from-AR-M[Field(ar3) in {K0 K1 K2 K4 K5 K7}]</td>
</tr>
<tr>
<td>mov-from-AR-LC</td>
<td>mov-from-AR-M[Field(ar3) == LC]</td>
</tr>
<tr>
<td>mov-from-AR-M</td>
<td>mov_ar[Format in {M31}]</td>
</tr>
<tr>
<td>mov-from-AR-PFS</td>
<td>mov-from-AR-M[Field(ar3) == PFS]</td>
</tr>
<tr>
<td>mov-from-AR-RNAT</td>
<td>mov-from-AR-M[Field(ar3) == RNAT]</td>
</tr>
<tr>
<td>mov-from-AR-RSC</td>
<td>mov-from-AR-M[Field(ar3) == RSC]</td>
</tr>
<tr>
<td>mov-from-AR-RUC</td>
<td>mov-from-AR-M[Field(ar3) == RUC]</td>
</tr>
<tr>
<td>mov-from-AR-rv</td>
<td>none</td>
</tr>
<tr>
<td>mov-from-AR-SSD</td>
<td>mov-from-AR-M[Field(ar3) == SSD]</td>
</tr>
<tr>
<td>mov-from-AR-UNAT</td>
<td>mov-from-AR-M[Field(ar3) == UNAT]</td>
</tr>
<tr>
<td>mov-from-BR</td>
<td>mov_br[Format in {I22}]</td>
</tr>
<tr>
<td>mov-from-CR</td>
<td>mov_cr[Format in {M33}]</td>
</tr>
<tr>
<td>mov-from-CR-CMCV</td>
<td>mov-from-CR-M[Field(cr3) == CMCV]</td>
</tr>
<tr>
<td>mov-from-CR-DCR</td>
<td>mov-from-CR-M[Field(cr3) == DCR]</td>
</tr>
<tr>
<td>mov-from-CR-EOI</td>
<td>mov-from-CR-M[Field(cr3) == EOI]</td>
</tr>
<tr>
<td>mov-from-CR-IFA</td>
<td>mov-from-CR-M[Field(cr3) == IFA]</td>
</tr>
<tr>
<td>mov-from-CR-IFS</td>
<td>mov-from-CR-M[Field(cr3) == IFS]</td>
</tr>
<tr>
<td>mov-from-CR-IHA</td>
<td>mov-from-CR-M[Field(cr3) == IHA]</td>
</tr>
<tr>
<td>mov-from-CR-IIB</td>
<td>mov-from-CR-M[Field(cr3) in {IIB0 IIB1}]</td>
</tr>
<tr>
<td>mov-from-CR-IIM</td>
<td>mov-from-CR-M[Field(cr3) == IIM]</td>
</tr>
<tr>
<td>mov-from-CR-IIP</td>
<td>mov-from-CR-M[Field(cr3) == IIP]</td>
</tr>
<tr>
<td>mov-from-CR-IIPA</td>
<td>mov-from-CR-M[Field(cr3) == IIPA]</td>
</tr>
<tr>
<td>mov-from-CR-IPSR</td>
<td>mov-from-CR-M[Field(cr3) == IPSR]</td>
</tr>
<tr>
<td>mov-from-CR-I RR</td>
<td>mov-from-CR-M[Field(cr3) in {IRR0 IRR1 IRR2 IRR3}]</td>
</tr>
<tr>
<td>mov-from-CR-IS R</td>
<td>mov-from-CR-M[Field(cr3) == ISR]</td>
</tr>
<tr>
<td>mov-from-CR-IT IR</td>
<td>mov-from-CR-M[Field(cr3) == ITIR]</td>
</tr>
<tr>
<td>mov-from-CR-IT M</td>
<td>mov-from-CR-M[Field(cr3) == ITM]</td>
</tr>
<tr>
<td>mov-from-CR-ITO</td>
<td>mov-from-CR-M[Field(cr3) == ITO]</td>
</tr>
<tr>
<td>mov-from-CR-ITV</td>
<td>mov-from-CR-M[Field(cr3) == ITV]</td>
</tr>
<tr>
<td>mov-from-CR-IVA</td>
<td>mov-from-CR-M[Field(cr3) == IVA]</td>
</tr>
<tr>
<td>mov-from-CR-IV R</td>
<td>mov-from-CR-M[Field(cr3) == IVR]</td>
</tr>
<tr>
<td>mov-from-CR-L I D</td>
<td>mov-from-CR-M[Field(cr3) == LID]</td>
</tr>
<tr>
<td>mov-from-CR-L RR</td>
<td>mov-from-CR-M[Field(cr3) in {LRR0 LRR1}]</td>
</tr>
<tr>
<td>mov-from-CR-PMV</td>
<td>mov-from-CR-M[Field(cr3) == PMV]</td>
</tr>
<tr>
<td>mov-from-CR-PT A</td>
<td>mov-from-CR-M[Field(cr3) == PTA]</td>
</tr>
<tr>
<td>mov-from-Cr-rv</td>
<td>none</td>
</tr>
<tr>
<td>mov-from-CR-TP R</td>
<td>mov-from-CR-M[Field(cr3) == TPR]</td>
</tr>
</tbody>
</table>
### Table 5-5. Instruction Classes (Continued)

<table>
<thead>
<tr>
<th>Class</th>
<th>Events/Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov-from-IND</td>
<td>mov_indirect[Format in {M43}]</td>
</tr>
<tr>
<td>mov-from-IND-CPUID</td>
<td>mov-from-IND[Field(reg) == cpuid]</td>
</tr>
<tr>
<td>mov-from-IND-DBR</td>
<td>mov-from-IND[Field(reg) == dbr]</td>
</tr>
<tr>
<td>mov-from-IND-IBR</td>
<td>mov-from-IND[Field(reg) == ibr]</td>
</tr>
<tr>
<td>mov-from-IND-PKR</td>
<td>mov-from-IND[Field(reg) == pkr]</td>
</tr>
<tr>
<td>mov-from-IND-PMC</td>
<td>mov-from-IND[Field(reg) == pmc]</td>
</tr>
<tr>
<td>mov-from-IND-PMD</td>
<td>mov-from-IND[Field(reg) == pmd]</td>
</tr>
<tr>
<td>mov-from-IND-priv</td>
<td>mov-from-IND[Field(reg) in {dbr ibr pkr pmc rr}]</td>
</tr>
<tr>
<td>mov-from-IND-RR</td>
<td>mov-from-IND[Field(reg) == rr]</td>
</tr>
<tr>
<td>mov-from-PR</td>
<td>mov_pr[Format in {I25}]</td>
</tr>
<tr>
<td>mov-from-PSR</td>
<td>mov_psr[Format in {M36}]</td>
</tr>
<tr>
<td>mov-from-PSR-um</td>
<td>mov_um[Format in {M36}]</td>
</tr>
<tr>
<td>mov-ip</td>
<td>mov_ip[Format in {I25}]</td>
</tr>
<tr>
<td>mov-to-AR</td>
<td>mov-to-AR-M, mov-to-AR-I</td>
</tr>
<tr>
<td>mov-to-AR-BSP</td>
<td>mov-to-AR-M[Field(ar3) == BSP]</td>
</tr>
<tr>
<td>mov-to-AR-BSPSTORE</td>
<td>mov-to-AR-M[Field(ar3) == BSPSTORE]</td>
</tr>
<tr>
<td>mov-to-AR-CCV</td>
<td>mov-to-AR-M[Field(ar3) == CCV]</td>
</tr>
<tr>
<td>mov-to-AR-CFLG</td>
<td>mov-to-AR-M[Field(ar3) == CFLG]</td>
</tr>
<tr>
<td>mov-to-AR-CSD</td>
<td>mov-to-AR-M[Field(ar3) == CSD]</td>
</tr>
<tr>
<td>mov-to-AR-EC</td>
<td>mov-to-AR-I[Field(ar3) == EC]</td>
</tr>
<tr>
<td>mov-to-AR-EFLAG</td>
<td>mov-to-AR-M[Field(ar3) == EFLAG]</td>
</tr>
<tr>
<td>mov-to-AR-FCR</td>
<td>mov-to-AR-M[Field(ar3) == FCR]</td>
</tr>
<tr>
<td>mov-to-AR-FDR</td>
<td>mov-to-AR-M[Field(ar3) == FDR]</td>
</tr>
<tr>
<td>mov-to-AR-FIR</td>
<td>mov-to-AR-M[Field(ar3) == FIR]</td>
</tr>
<tr>
<td>mov-to-AR-FPSR</td>
<td>mov-to-AR-M[Field(ar3) == FPSR]</td>
</tr>
<tr>
<td>mov-to-AR-FSR</td>
<td>mov-to-AR-M[Field(ar3) == FSR]</td>
</tr>
<tr>
<td>mov-to-AR-gr</td>
<td>mov-to-AR-M[Format in {M29}], mov-to-AR-I[Format in {I26}]</td>
</tr>
<tr>
<td>mov-to-AR-I</td>
<td>mov_ar[Format in {I26 I27}]</td>
</tr>
<tr>
<td>mov-to-AR-ig</td>
<td>mov-to-AR-I[Field(ar3) in {48-63 112-127}]</td>
</tr>
<tr>
<td>mov-to-AR-IM</td>
<td>mov_ar[Format in {I26 I27 M29 M30}]</td>
</tr>
<tr>
<td>mov-to-AR-ITC</td>
<td>mov-to-AR-M[Field(ar3) == ITC]</td>
</tr>
<tr>
<td>mov-to-AR-K</td>
<td>mov-to-AR-M[Field(ar3) in {K0 K1 K2 K3 K4 K5 K6 K7}]</td>
</tr>
<tr>
<td>mov-to-AR-LC</td>
<td>mov-to-AR-I[Field(ar3) == LC]</td>
</tr>
<tr>
<td>mov-to-AR-M</td>
<td>mov_ar[Format in {M29 M30}]</td>
</tr>
<tr>
<td>mov-to-AR-PFS</td>
<td>mov-to-AR-I[Field(ar3) == PFS]</td>
</tr>
<tr>
<td>mov-to-AR-RNAT</td>
<td>mov-to-AR-M[Field(ar3) == RNAT]</td>
</tr>
<tr>
<td>mov-to-AR-RSC</td>
<td>mov-to-AR-M[Field(ar3) == RSC]</td>
</tr>
<tr>
<td>mov-to-AR-RUC</td>
<td>mov-to-AR-M[Field(ar3) == RUC]</td>
</tr>
<tr>
<td>mov-to-AR-SSD</td>
<td>mov-to-AR-M[Field(ar3) == SSD]</td>
</tr>
<tr>
<td>mov-to-AR-UNAT</td>
<td>mov-to-AR-M[Field(ar3) == UNAT]</td>
</tr>
<tr>
<td>mov-to-AR-UNAT</td>
<td>mov-to-AR-M[Field(ar3) == UNAT]</td>
</tr>
<tr>
<td>mov-to-AR-UNAT</td>
<td>mov-to-AR-M[Field(ar3) == UNAT]</td>
</tr>
<tr>
<td>mov-to-AR-UNAT</td>
<td>mov-to-AR-M[Field(ar3) == UNAT]</td>
</tr>
<tr>
<td>mov-to-BR</td>
<td>mov_br[Format in {I21}]</td>
</tr>
<tr>
<td>mov-to-CR</td>
<td>mov_cr[Format in {M32}]</td>
</tr>
<tr>
<td>mov-to-CR-CMCV</td>
<td>mov-to-CR[Field(cr3) == CMCV]</td>
</tr>
</tbody>
</table>
Table 5-5. Instruction Classes (Continued)

<table>
<thead>
<tr>
<th>Class</th>
<th>Events/Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov-to-CR-DCR</td>
<td>mov-to-CR[field(cr3) == DCR]</td>
</tr>
<tr>
<td>mov-to-CR-EOI</td>
<td>mov-to-CR[field(cr3) == EOI]</td>
</tr>
<tr>
<td>mov-to-CR-IFIA</td>
<td>mov-to-CR[field(cr3) == IFA]</td>
</tr>
<tr>
<td>mov-to-CR-IFS</td>
<td>mov-to-CR[field(cr3) == IFS]</td>
</tr>
<tr>
<td>mov-to-CR-IHA</td>
<td>mov-to-CR[field(cr3) == IHA]</td>
</tr>
<tr>
<td>mov-to-CR-IIB</td>
<td>mov-to-CR[field(cr3) in {IIB0 IIB1}]</td>
</tr>
<tr>
<td>mov-to-CR-IIM</td>
<td>mov-to-CR[field(cr3) == IIM]</td>
</tr>
<tr>
<td>mov-to-CR-IIP</td>
<td>mov-to-CR[field(cr3) == IIP]</td>
</tr>
<tr>
<td>mov-to-CR-IIPA</td>
<td>mov-to-CR[field(cr3) == IIPA]</td>
</tr>
<tr>
<td>mov-to-CR-IPSR</td>
<td>mov-to-CR[field(cr3) == IPSR]</td>
</tr>
<tr>
<td>mov-to-CR-IRR</td>
<td>mov-to-CR[field(cr3) in {IRR0 IRR1 IRR2 IRR3}]</td>
</tr>
<tr>
<td>mov-to-CR-ISR</td>
<td>mov-to-CR[field(cr3) == ISR]</td>
</tr>
<tr>
<td>mov-to-CR-ITIR</td>
<td>mov-to-CR[field(cr3) == ITIR]</td>
</tr>
<tr>
<td>mov-to-CR-ITM</td>
<td>mov-to-CR[field(cr3) == ITM]</td>
</tr>
<tr>
<td>mov-to-CR-ITO</td>
<td>mov-to-CR[field(cr3) == ITO]</td>
</tr>
<tr>
<td>mov-to-CR-ITV</td>
<td>mov-to-CR[field(cr3) == ITV]</td>
</tr>
<tr>
<td>mov-to-CR-IVA</td>
<td>mov-to-CR[field(cr3) == IVA]</td>
</tr>
<tr>
<td>mov-to-CR-IVR</td>
<td>mov-to-CR[field(cr3) == IVR]</td>
</tr>
<tr>
<td>mov-to-CR-LID</td>
<td>mov-to-CR[field(cr3) == LID]</td>
</tr>
<tr>
<td>mov-to-CR-LRR</td>
<td>mov-to-CR[field(cr3) in {LRR0 LRR1}]</td>
</tr>
<tr>
<td>mov-to-CR-PMV</td>
<td>mov-to-CR[field(cr3) == PMV]</td>
</tr>
<tr>
<td>mov-to-CR-PTA</td>
<td>mov-to-CR[field(cr3) == PTA]</td>
</tr>
<tr>
<td>mov-to-CR-TPR</td>
<td>mov-to-CR[field(cr3) == TPR]</td>
</tr>
<tr>
<td>mov-to-IND</td>
<td>mov_indirect[Format in {M42}]</td>
</tr>
<tr>
<td>mov-to-IND-CPUID</td>
<td>mov-to-IND[field(reg) == cpuid]</td>
</tr>
<tr>
<td>mov-to-IND-DBR</td>
<td>mov-to-IND[field(reg) == dbr]</td>
</tr>
<tr>
<td>mov-to-IND-IBR</td>
<td>mov-to-IND[field(reg) == ibr]</td>
</tr>
<tr>
<td>mov-to-IND-PKR</td>
<td>mov-to-IND[field(reg) == pkr]</td>
</tr>
<tr>
<td>mov-to-IND-PMC</td>
<td>mov-to-IND[field(reg) == pmc]</td>
</tr>
<tr>
<td>mov-to-IND-PMD</td>
<td>mov-to-IND[field(reg) == pmd]</td>
</tr>
<tr>
<td>mov-to-IND-priv</td>
<td>mov-to-IND</td>
</tr>
<tr>
<td>mov-to-IND-RR</td>
<td>mov-to-IND[field(reg) == rr]</td>
</tr>
<tr>
<td>mov-to-PR</td>
<td>mov-to-PR-allreg, mov-to-PR-rotreg</td>
</tr>
<tr>
<td>mov-to-PR-allreg</td>
<td>mov_pr[Format in {I23}]</td>
</tr>
<tr>
<td>mov-to-PR-rotreg</td>
<td>mov_pr[Format in {I24}]</td>
</tr>
<tr>
<td>mov-to-PSR-I</td>
<td>mov_psr[Format in {M35}]</td>
</tr>
<tr>
<td>mov-to-PSR-um</td>
<td>mov_um[Format in {M35}]</td>
</tr>
<tr>
<td>mux</td>
<td>mux1, mux2</td>
</tr>
<tr>
<td>non-access</td>
<td>fc, lfetch, probe-all, tpa, tak</td>
</tr>
<tr>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>pack</td>
<td>pack2, pack4</td>
</tr>
<tr>
<td>padd</td>
<td>padd1, padd2, padd4</td>
</tr>
<tr>
<td>pavg</td>
<td>pavg1, pavg2</td>
</tr>
<tr>
<td>Class</td>
<td>Events/Instructions</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pavgsub</td>
<td>pavgsub1, pavgsub2</td>
</tr>
<tr>
<td>pcmp</td>
<td>pcmp1, pcmp2, pcmp4</td>
</tr>
<tr>
<td>pmax</td>
<td>pmax1, pmax2</td>
</tr>
<tr>
<td>pmin</td>
<td>pmin1, pmin2</td>
</tr>
<tr>
<td>pmpy</td>
<td>pmpy2</td>
</tr>
<tr>
<td>pmpyshr</td>
<td>pmpyshr2</td>
</tr>
<tr>
<td>pr-and-writers</td>
<td>pr-gen-writers-int[Field(ctype) in {and andcm}], pr-gen-writers-int[Field(ctype) in {or.andcm and.orcm}]</td>
</tr>
<tr>
<td>pr-gen-writers-fp</td>
<td>pr-gen-writers-fp[Field(ctype)==]</td>
</tr>
<tr>
<td>pr-gen-writers-int</td>
<td>pr-gen-writers-int[Field(ctype)==]</td>
</tr>
<tr>
<td>pr-or-writers</td>
<td>pr-gen-writers-int[Field(ctype) in {or orcm}], pr-gen-writers-int[Field(ctype) in {or orcm}]</td>
</tr>
<tr>
<td>pr-readers-br</td>
<td>br.call, br.cond, br.l.call, br.lCOND, br.ret, br.wexit, break.b, hint.b, nop.b, ReservedBQP</td>
</tr>
<tr>
<td>pr-readers-nobr-nomovpr</td>
<td>add, addl, addpd4, adds, and, andcm, break.f, break.i, break.m, break.x, chk.s, chk.a, cmp, cmp4, cmpxchg, ciz, czx, dep, extr, fp-arith, fp-non-arith, fc, fchif, fcif, fcmp, fetchadd, fcpmp, fselt, fwb, gelf, hint.f, hint.i, hint.m, hint.x, invala-all, itc.i, itc.d, itri.i, itrd.i, ld, ld.f, ld.fp, fetch-all, mf, mix, mov-from-AR-M, mov-from-AR-IM, mov-from-AR-I, mov-to-AR-M, mov-to-AR-IM, mov-to-AR-I, mov-to-BR, mov-from-BR, mov-to-CR, mov-from-CR, mov-to-IND, mov-from-IND, mov-ip, mov-to-PSR-um, mov-from-PSR, mov-from-PSR-um, movl, mux, nop.f, nop.i, nop.m, nop.x, or, pack, padd, pavg, pavgsub, pcmp, pmax, pmin, pmpy, pmpyshr, popcnt, probe-all, psad, pshl, pshladd, pshr, pshrad, psub, ptce, ptcg, ptcga, ptcl, ptrd, ptrl, ReservedQP, rsm, selfl, shl, shladd, shladdp4, shr, shrp, shrz.i, shrz.d, ssm, st, stf, sub, sum, sxt, sync, tak, tbit, tf, thash, tnat, tpa, tag, unpack, xchg, xma, xmpy, xor, zxt</td>
</tr>
<tr>
<td>pr-unc-writers-fp</td>
<td>pr-gen-writers-fp[Field(ctype)==unc], fprcpa11, fprsqrta11, frcpa11, frsqrta11</td>
</tr>
<tr>
<td>pr-unc-writers-int</td>
<td>pr-gen-writers-int[Field(ctype)==unc]</td>
</tr>
<tr>
<td>pr-writers</td>
<td>pr-gen-writers-int[Field(ctype)==unc], pr-gen-writers-int[Field(ctype)==unc]</td>
</tr>
<tr>
<td>pr-and-writers</td>
<td>pr-gen-writers-int[Field(ctype) in {and andcm}], pr-gen-writers-int[Field(ctype) in {or.andcm and.orcm}]</td>
</tr>
<tr>
<td>pr-or-writers</td>
<td>pr-gen-writers-int[Field(ctype) in {or orcm}], pr-gen-writers-int[Field(ctype) in {or orcm}]</td>
</tr>
<tr>
<td>pr-readers-br</td>
<td>br.call, br.cond, br.l.call, br.lCOND, br.ret, br.wexit, break.b, hint.b, nop.b, ReservedBQP</td>
</tr>
<tr>
<td>pr-readers-nobr-nomovpr</td>
<td>add, addl, addpd4, adds, and, andcm, break.f, break.i, break.m, break.x, chk.s, chk.a, cmp, cmp4, cmpxchg, ciz, czx, dep, extr, fp-arith, fp-non-arith, fc, fchif, fcif, fcmp, fetchadd, fcpmp, fselt, fwb, gelf, hint.f, hint.i, hint.m, hint.x, invala-all, itc.i, itc.d, itri.i, itrd.i, ld, ld.f, ld.fp, fetch-all, mf, mix, mov-from-AR-M, mov-from-AR-IM, mov-from-AR-I, mov-to-AR-M, mov-to-AR-IM, mov-to-AR-I, mov-to-BR, mov-from-BR, mov-to-CR, mov-from-CR, mov-to-IND, mov-from-IND, mov-ip, mov-to-PSR-um, mov-from-PSR, mov-from-PSR-um, movl, mux, nop.f, nop.i, nop.m, nop.x, or, pack, padd, pavg, pavgsub, pcmp, pmax, pmin, pmpy, pmpyshr, popcnt, probe-all, psad, pshl, pshladd, pshr, pshrad, psub, ptce, ptcg, ptcga, ptcl, ptrd, ptrl, ReservedQP, rsm, selfl, shl, shladd, shladdp4, shr, shrp, shrz.i, shrz.d, ssm, st, stf, sub, sum, sxt, sync, tak, tbit, tf, thash, tnat, tpa, tag, unpack, xchg, xma, xmpy, xor, zxt</td>
</tr>
<tr>
<td>pr-unc-writers-fp</td>
<td>pr-gen-writers-fp[Field(ctype)==unc], fprcpa11, fprsqrta11, frcpa11, frsqrta11</td>
</tr>
<tr>
<td>pr-unc-writers-int</td>
<td>pr-gen-writers-int[Field(ctype)==unc]</td>
</tr>
<tr>
<td>pr-writers</td>
<td>pr-gen-writers-int[Field(ctype)==unc], pr-gen-writers-int[Field(ctype)==unc]</td>
</tr>
<tr>
<td>pr-and-writers</td>
<td>pr-gen-writers-int[Field(ctype) in {and andcm}], pr-gen-writers-int[Field(ctype) in {or.andcm and.orcm}]</td>
</tr>
<tr>
<td>pr-or-writers</td>
<td>pr-gen-writers-int[Field(ctype) in {or orcm}], pr-gen-writers-int[Field(ctype) in {or orcm}]</td>
</tr>
<tr>
<td>predicable-instructions</td>
<td>mov-from-PR, mov-to-PR, pr-readers-br, pr-readers-nobr-nomovpr</td>
</tr>
<tr>
<td>probe-all</td>
<td>probe-fault, probe-regular</td>
</tr>
<tr>
<td>probe-fault</td>
<td>probe[Format in {M40}]</td>
</tr>
<tr>
<td>probe-regular</td>
<td>probe[Format in {M38 M39}]</td>
</tr>
<tr>
<td>psad</td>
<td>psad1</td>
</tr>
<tr>
<td>pshl</td>
<td>pshl2, pshl4</td>
</tr>
<tr>
<td>pshladd</td>
<td>pshladd2</td>
</tr>
<tr>
<td>pshr</td>
<td>pshr2, pshr4</td>
</tr>
<tr>
<td>pshrad</td>
<td>pshrad2</td>
</tr>
<tr>
<td>psub</td>
<td>psub1, psub2, psub4</td>
</tr>
<tr>
<td>ReservedBQP</td>
<td>-15</td>
</tr>
<tr>
<td>ReservedQP</td>
<td>-16</td>
</tr>
</tbody>
</table>

Table 5-5. Instruction Classes (Continued)
Table 5-5. Instruction Classes (Continued)

<table>
<thead>
<tr>
<th>Class</th>
<th>Events/Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>rse-readers</td>
<td>alloc, br.call, br.ia, br.ret, brl.call, cover, flushrs, loadrs, mov-from-AR-BSP,</td>
</tr>
<tr>
<td></td>
<td>mov-from-AR-BSPSTORE, mov-to-AR-BSPSTORE, mov-from-AR-RNAT, mov-to-AR-RNAT, rfi</td>
</tr>
<tr>
<td>rse-writers</td>
<td>alloc, br.call, br.ia, br.ret, brl.call, cover, flushrs, loadrs, mov-to-AR-BSPSTORE, rfi</td>
</tr>
<tr>
<td>st</td>
<td>st1, st2, st4, st8, st8.spill, st16</td>
</tr>
<tr>
<td>st-postinc</td>
<td>stf[Format in (M10)], stf[Format in (M5)]</td>
</tr>
<tr>
<td>stf</td>
<td>stfs, stfd, stfe, stf8, stf.spill</td>
</tr>
<tr>
<td>sxt</td>
<td>sx1, sx2, sx4</td>
</tr>
<tr>
<td>sys-mask-writers-partial</td>
<td>rsm, ssm</td>
</tr>
<tr>
<td>unpack</td>
<td>unpack1, unpack2, unpack4</td>
</tr>
<tr>
<td>unpredicatable-instructions</td>
<td>alloc, br.cloop, br.clop, br.cexit, br.ia, brp, bsw, clrrb, cover, ep, flushrs, loadrs, rfi, vmsw</td>
</tr>
<tr>
<td>user-mask-writers-partial</td>
<td>rum, sum</td>
</tr>
<tr>
<td>xchg</td>
<td>xchg1, xchg2, xchg4, xchg8</td>
</tr>
<tr>
<td>zxt</td>
<td>zxt1, zxt2, zxt4</td>
</tr>
</tbody>
</table>
Index
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
ADDPs 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
Acquire Semantics 2:507
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.la 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
brl Instruction 3:30
brrp Instruction 3:32
BSF Instruction 4:35
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
clrrb 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
CMPSQ Instruction 4:71
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,
INDEX

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
CVTS2SS Instruction 4:502
CVTTPS2PI Instruction 4:504
CVTTSS2SI Instruction 4:506
CWD Instruction 4:85
CWDE Instruction 4:57, 4:86
cxz 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, 1:579
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
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
fclog Instruction 3:64
FCLX Instruction 4:109
fclog Instruction 3:66
FCMOI Instruction 4:115
FCMVcc Instruction 4:110
fcmp Instruction 3:67
FCOM Instruction 4:112
FCOMIP Instruction 4:115
FCOMPP 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
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page References</th>
</tr>
</thead>
<tbody>
<tr>
<td>FICOM Instruction 4:128</td>
<td></td>
</tr>
<tr>
<td>FICOMP Instruction 4:128</td>
<td></td>
</tr>
<tr>
<td>FIDIV Instruction 4:121</td>
<td></td>
</tr>
<tr>
<td>FIDIVR Instruction 4:124</td>
<td></td>
</tr>
<tr>
<td>FILD Instruction 4:130</td>
<td></td>
</tr>
<tr>
<td>FIMUL Instruction 4:145</td>
<td></td>
</tr>
<tr>
<td>FINCSTP Instruction 4:132</td>
<td></td>
</tr>
<tr>
<td>Firmware 1:7, 2:623</td>
<td></td>
</tr>
<tr>
<td>Firmware Address Space 2:283</td>
<td></td>
</tr>
<tr>
<td>Firmware Entrypoint 2:281, 2:350</td>
<td></td>
</tr>
<tr>
<td>Firmware Interface Table (FIT) 2:287</td>
<td></td>
</tr>
<tr>
<td>FIST Instruction 4:134</td>
<td></td>
</tr>
<tr>
<td>FISTP Instruction 4:134</td>
<td></td>
</tr>
<tr>
<td>FISUB Instruction 4:182, 4:183</td>
<td></td>
</tr>
<tr>
<td>FISUBR Instruction 4:185</td>
<td></td>
</tr>
<tr>
<td>FIT (Firmware Interface Table) 2:287</td>
<td></td>
</tr>
<tr>
<td>FLD Instruction 4:137</td>
<td></td>
</tr>
<tr>
<td>FLD1 Instruction 4:139</td>
<td></td>
</tr>
<tr>
<td>FLDCW Instruction 4:141</td>
<td></td>
</tr>
<tr>
<td>FLDENV Instruction 4:143</td>
<td></td>
</tr>
<tr>
<td>FLDL2E Instruction 4:139</td>
<td></td>
</tr>
<tr>
<td>FLDL2T Instruction 4:139</td>
<td></td>
</tr>
<tr>
<td>FLDLG2 Instruction 4:139</td>
<td></td>
</tr>
<tr>
<td>FLDLN2 Instruction 4:139</td>
<td></td>
</tr>
<tr>
<td>FLDPI Instruction 4:139</td>
<td></td>
</tr>
<tr>
<td>FLDZ Instruction 4:139</td>
<td></td>
</tr>
<tr>
<td>Floating-point Architecture 1:19, 1:85, 1:205</td>
<td></td>
</tr>
<tr>
<td>Floating-point Exception Fault 1:102</td>
<td></td>
</tr>
<tr>
<td>Floating-point Instructions 1:91</td>
<td></td>
</tr>
<tr>
<td>Floating-point Register (FR) 1:139</td>
<td></td>
</tr>
<tr>
<td>Floating-point Software Assistance Exception</td>
<td></td>
</tr>
<tr>
<td>Handler (FPSWA) 2:587</td>
<td></td>
</tr>
<tr>
<td>Floating-point Status Register (FPSR) 1:31, 1:88</td>
<td></td>
</tr>
<tr>
<td>flushrs Instruction 3:76</td>
<td></td>
</tr>
<tr>
<td>fma Instruction 1:210, 3:77</td>
<td></td>
</tr>
<tr>
<td>fmax Instruction 3:79</td>
<td></td>
</tr>
<tr>
<td>fmerge Instruction 3:80</td>
<td></td>
</tr>
<tr>
<td>fmin Instruction 3:82</td>
<td></td>
</tr>
<tr>
<td>fmix Instruction 3:83</td>
<td></td>
</tr>
<tr>
<td>fmpy Instruction 3:85</td>
<td></td>
</tr>
<tr>
<td>fms Instruction 3:86</td>
<td></td>
</tr>
<tr>
<td>FMUL Instruction 4:145</td>
<td></td>
</tr>
<tr>
<td>FMULP Instruction 4:145</td>
<td></td>
</tr>
<tr>
<td>FNCLEX Instruction 4:109</td>
<td></td>
</tr>
<tr>
<td>fneg Instruction 3:88</td>
<td></td>
</tr>
<tr>
<td>fnegabs Instruction 3:89</td>
<td></td>
</tr>
<tr>
<td>FNINIT Instruction 4:133</td>
<td></td>
</tr>
<tr>
<td>fnma Instruction 3:90</td>
<td></td>
</tr>
<tr>
<td>fnmpy Instruction 3:92</td>
<td></td>
</tr>
<tr>
<td>FNOP Instruction 4:148</td>
<td></td>
</tr>
<tr>
<td>fnorm Instruction 3:93</td>
<td></td>
</tr>
<tr>
<td>FNSAVE Instruction 4:162</td>
<td></td>
</tr>
<tr>
<td>FNSTCW Instruction 4:176</td>
<td></td>
</tr>
<tr>
<td>FNSTENV Instruction 4:178</td>
<td></td>
</tr>
<tr>
<td>FNSTSW Instruction 4:180</td>
<td></td>
</tr>
</tbody>
</table>

Index: 3
INDEX

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 2:237
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 (interruption Faulting Address) 2:541
IFS (Interruption Function State) 2:541
IHA (Interuption Hash Address) 2:41, 2:541
IIB0 (Interuption Instruction Bundle 0) 2:541
IIB1 (Interuption Instruction Bundle 1) 2:541
IIM (Interuption Immediate) 2:541
IIP (Interuption Instruction Pointer) 2:541
IIPA (Interuption 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
Interrupt Control Registers (CR16-27) 2:36
Interrupt Handler 2:537
Interrupt Handling 2:543
Interrupt Hash Address 2:41
Interrupt Instruction Bundle Registers (IIB0-1) 2:42
Interrupt Processor Status Register (IPSR) 2:36
Interrupt Register State 2:540
Interrupt Registers 2:538
Interrupt Status Register (ISR) 2:36
Interrupt 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
Interrupt Vector Address 2:35, 2:538
Interrupt 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 (Interrupt Processor Status Register) 2:36, 2:541
IRET Instruction 4:231
IRETD Instruction 4:231
IRR (External Interrupt Request Registers) 2:125
ISR (Interrupt 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 (Interrupt 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 (Interrupt Vector Address) 2:95, 2:537
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
ldfp Instruction 3:161
LDMXCSR Instruction 4:516
LEA Instruction 4:255
LEAVE Instruction 4:260
LES Instruction 4:255
lfetch Instruction 3:164
LFS Instruction 4:255
LGDT Instruction 4:264
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
PADD Instruction 4:410
PADDDB Instruction 4:413
PADD指令 4:413
PADDW Instruction 4:416
PADDW指令 4:416
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
INDEX

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
pcmp Instruction 3:206
PCMPEQD Instruction 4:423
PCMPEQD Instruction 4:423
PCMPGTB Instruction 4:426
PCMPGD 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
PEXTRW 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 Register) 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
POF Instruction 4:311
POPA Instruction 4:315
POPAD Instruction 4:315
popcnt Instruction 3:216
POPF Instruction 4:317
POPF 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
pxr Instruction 3:224
pxhradd Instruction 3:226
PSHUFW Instruction 4:575
PSLLD Instruction 4:437
PSLQQ 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
<table>
<thead>
<tr>
<th>Index</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>10 Index for Volumes 1, 2, 3 and 4</td>
</tr>
<tr>
<td>SIDT Instruction</td>
<td>4:359</td>
</tr>
<tr>
<td>Single Step Trap</td>
<td>2:151</td>
</tr>
<tr>
<td>SDLT Instruction</td>
<td>4:367</td>
</tr>
<tr>
<td>SMSW Instruction</td>
<td>4:369</td>
</tr>
<tr>
<td>Software Pipelining</td>
<td>1:19, 1:75, 1:145, 1:181</td>
</tr>
<tr>
<td>Speculation</td>
<td>1:16, 1:142, 1:151</td>
</tr>
<tr>
<td>Control Speculation</td>
<td>1:16</td>
</tr>
<tr>
<td>Data Speculation</td>
<td>1:17</td>
</tr>
<tr>
<td>Recovery Code</td>
<td>1:17, 2:580</td>
</tr>
<tr>
<td>Speculation Check</td>
<td>1:156</td>
</tr>
<tr>
<td>SQRTPS Instruction</td>
<td>4:551</td>
</tr>
<tr>
<td>SQRTSS Instruction</td>
<td>4:552</td>
</tr>
<tr>
<td>srlz Instruction</td>
<td>3:249</td>
</tr>
<tr>
<td>SSE Instructions</td>
<td>4:463</td>
</tr>
<tr>
<td>ssm Instruction</td>
<td>3:250</td>
</tr>
<tr>
<td>st Instruction</td>
<td>3:251</td>
</tr>
<tr>
<td>Stacked Calling Convention</td>
<td>2:352</td>
</tr>
<tr>
<td>Stacked General Registers</td>
<td>2:550</td>
</tr>
<tr>
<td>Stacked Registers</td>
<td>1:144</td>
</tr>
<tr>
<td>Static Calling Convention</td>
<td>2:352</td>
</tr>
<tr>
<td>Static General Registers</td>
<td>2:550</td>
</tr>
<tr>
<td>STC Instruction</td>
<td>4:371</td>
</tr>
<tr>
<td>STD Instruction</td>
<td>4:372</td>
</tr>
<tr>
<td>stf Instruction</td>
<td>3:254</td>
</tr>
<tr>
<td>STI Instruction</td>
<td>4:373</td>
</tr>
<tr>
<td>STMXCSR Instruction</td>
<td>4:553</td>
</tr>
<tr>
<td>Stops</td>
<td>1:38</td>
</tr>
<tr>
<td>Store Instructions</td>
<td>1:59</td>
</tr>
<tr>
<td>Stores to Memory</td>
<td>1:147</td>
</tr>
<tr>
<td>STOS Instruction</td>
<td>4:376</td>
</tr>
<tr>
<td>STOSB Instruction</td>
<td>4:376</td>
</tr>
<tr>
<td>STOSD Instruction</td>
<td>4:376</td>
</tr>
<tr>
<td>STOSW Instruction</td>
<td>4:376</td>
</tr>
<tr>
<td>STR Instruction</td>
<td>4:378</td>
</tr>
<tr>
<td>SUB Instruction</td>
<td>4:379</td>
</tr>
<tr>
<td>sub Instruction</td>
<td>3:256</td>
</tr>
<tr>
<td>SUBPS Instruction</td>
<td>4:554</td>
</tr>
<tr>
<td>SUBSS Instruction</td>
<td>4:555</td>
</tr>
<tr>
<td>sum Instruction</td>
<td>3:257</td>
</tr>
<tr>
<td>sxt Instruction</td>
<td>3:258</td>
</tr>
<tr>
<td>sync Instruction</td>
<td>3:259</td>
</tr>
<tr>
<td>sync.i</td>
<td>2:526</td>
</tr>
<tr>
<td>System Abstraction Layer - See SAL (System Abstraction Layer)</td>
<td></td>
</tr>
<tr>
<td>System Architecture</td>
<td>1:20</td>
</tr>
<tr>
<td>System Environment</td>
<td>2:13</td>
</tr>
<tr>
<td>System Programmer’s Guide</td>
<td>2:501</td>
</tr>
<tr>
<td>System State</td>
<td>2:20</td>
</tr>
<tr>
<td>tak Instruction</td>
<td>3:260</td>
</tr>
<tr>
<td>Taken Branch trap</td>
<td>2:151</td>
</tr>
<tr>
<td>Task Priority Register (TPR)</td>
<td>2:123, 2:605</td>
</tr>
<tr>
<td>tbit Instruction</td>
<td>3:261</td>
</tr>
<tr>
<td>TC (Translation Cache)</td>
<td>2:49, 2:567</td>
</tr>
</tbody>
</table>

**U**

<table>
<thead>
<tr>
<th>Index</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCOMISS Instruction</td>
<td>4:556</td>
</tr>
<tr>
<td>UD2 Instruction</td>
<td>4:383</td>
</tr>
<tr>
<td>UEFI (Unified Extensible Firmware Interface)</td>
<td>2:630</td>
</tr>
<tr>
<td>UM (User Mask Register)</td>
<td>1:33</td>
</tr>
<tr>
<td>UNAT (User NaT Collection Register)</td>
<td>1:31, 1:156</td>
</tr>
<tr>
<td>Uncacheable Page</td>
<td>2:77</td>
</tr>
<tr>
<td>Unchanged Register</td>
<td>2:352</td>
</tr>
<tr>
<td>Unordered Semantics</td>
<td>2:507</td>
</tr>
<tr>
<td>unpack Instruction</td>
<td>3:270</td>
</tr>
<tr>
<td>UNPCKHPS Instruction</td>
<td>4:558</td>
</tr>
<tr>
<td>UNPCKLPS Instruction</td>
<td>4:560</td>
</tr>
<tr>
<td>User Mask (UM)</td>
<td>1:33</td>
</tr>
<tr>
<td>User NaT Collection Register (UNAT)</td>
<td>1:31, 1:156</td>
</tr>
</tbody>
</table>

**V**

<table>
<thead>
<tr>
<th>Index</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERR Instruction</td>
<td>4:384</td>
</tr>
<tr>
<td>VERW Instruction</td>
<td>4:384</td>
</tr>
<tr>
<td>VHPT (Virtual Hash Page Table)</td>
<td>2:61, 2:571</td>
</tr>
<tr>
<td>VHPT Translation Vector</td>
<td>2:173</td>
</tr>
<tr>
<td>Virtual Addressing</td>
<td>2:45</td>
</tr>
<tr>
<td>Virtual Hash Page Table (VHPT)</td>
<td>2:61, 2:571</td>
</tr>
<tr>
<td>Virtual Machine Monitor (VMM)</td>
<td>2:352</td>
</tr>
<tr>
<td>Virtual Processor Descriptor (VPD)</td>
<td>2:325, 2:352</td>
</tr>
<tr>
<td>Virtual Processor State</td>
<td>2:352</td>
</tr>
<tr>
<td>Virtual Processor Status Register (VPSR)</td>
<td>2:327</td>
</tr>
<tr>
<td>Virtual Region Number (VRN)</td>
<td>2:561</td>
</tr>
<tr>
<td>Virtualization</td>
<td>2:44, 2:324</td>
</tr>
<tr>
<td>Virtualization Acceleration Control (vac)</td>
<td>2:329</td>
</tr>
<tr>
<td>Virtualization Disable Control (vdc)</td>
<td>2:329</td>
</tr>
<tr>
<td>VMM (Virtual Machine Monitor)</td>
<td>2:352</td>
</tr>
<tr>
<td>vmsw Instruction</td>
<td>3:273</td>
</tr>
<tr>
<td>VPD (Virtual Processor Descriptor)</td>
<td>2:325, 2:352</td>
</tr>
<tr>
<td>VPSR (Virtual Processor Status Register)</td>
<td>2:327</td>
</tr>
<tr>
<td>VRN (Virtual Region Number)</td>
<td>2:561</td>
</tr>
</tbody>
</table>
INDEX

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
XT (External Task Priority Cycle) 2:130
XT (External Task Priority Register) 2:605

Z
zxt Instruction 3:280

Index for Volumes 1, 2, 3 and 4