Intel® Itanium® Architecture Software Developer's Manual

Revision 2.3

Volume 3: Intel® Itanium® Instruction Set

Intel® Itanium® Architecture Software Developer's Manual

Volume 3: Intel® Itanium® Instruction Set Reference

Revision 2.3

May 2010

Document Number: 323207

Contents

Figures

Tables

§

About this Manual 1

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

The *Intel® Itanium® Architecture Software Developer's Manual* provides a comprehensive description of the programming environment, resources, and instruction set visible to both the application and system programmer. In addition, it also describes how programmers can take advantage of the features of the Itanium architecture to help them optimize code.

1.1 Overview of Volume 1: Application Architecture

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

1.1.1 Part 1: Application Architecture Guide

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

Chapter 2, "Introduction to the Intel® Itanium® Architecture" provides an overview of the architecture.

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

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

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

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

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

Chapter 1, "About the Optimization Guide" gives an overview of the optimization guide.

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

Chapter 3, "Memory Reference" discusses features and optimizations related to control and data speculation.

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

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

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

1.2 Overview of Volume 2: System Architecture

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

1.2.1 Part 1: System Architecture Guide

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

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

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

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

Chapter 5, "Interruptions" describes all interruptions that can be generated by a processor based on the Itanium architecture.

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

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

Chapter 8, "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 1, "About this Manual" provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer's Manual*.

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

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

Chapter 4, "Instruction Formats" describes the encoding and instruction format instructions.

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

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

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

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

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

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

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

1.5 Terminology

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

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

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

IA-32 Architecture – The 32-bit and 16-bit Intel architecture as described in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.

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

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

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

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

1.6 Related Documents

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

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

- ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** – This set of manuals describes the Intel 32-bit architecture. They are available from the Intel Literature Department by calling 1-800-548-4725 and requesting Document Numbers 243190, 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

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

Date of Revision	Revision Number	Description
August 2005	2.2	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.l` 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 4.3.3, 4.4.1, 4.4.5, 4.4.6, 4.4.7, 5.5.2, and 7.1.2, Part I, Volume 2; Section 2.5, 2.5.1, 2.5.2, 2.5.3, and 4.5.2.1, Part II, Volume 2; Section 2.2, 3, 4.1, 4.4.6.5, and 4.4.10.10, Part I, Volume 3). 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).

Date of Revision	Revision Number	Description
		Volume 2: Class pr-writers-int clarification (Table A-5). PAL_MC_DRAIN clarification (Section 4.4.6.1). VHPT walk and forward progress change (Section 4.1.1.2). IA-32 IBR/DBR match clarification (Section 7.1.1). ISR figure changes (pp. 8-5, 8-26, 8-33 and 8-36). PAL_CACHE_FLUSH return argument change – added new status return argument (Section 11.8.3). PAL self-test Control and PAL_A procedure requirement change – added new arguments, figures, requirements (Section 11.2). PAL_CACHE_FLUSH clarifications (Chapter 11). Non-speculative reference clarification (Section 4.4.6). RID and Preferred Page Size usage clarification (Section 4.1). VHPT read atomicity clarification (Section 4.1). IIP and WC flush clarification (Section 4.4.5). Revised RSE and PMC typographical errors (Section 6.4). Revised DV table (Section A.4). Memory attribute transitions – added new requirements (Section 4.4). MCA for WC/UC aliasing change (Section 4.4.1). Bus lock deprecation – changed behavior of DCR 'lc' bit (Section 3.3.4.1, Section 10.6.8, Section 11.8.3). PAL_PROC_GET/SET_FEATURES changes – extend calls to allow implementation-specific feature control (Section 11.8.3). Split PAL_A architecture changes (Section 11.1.6). Simple barrier synchronization clarification (Section 13.4.2). Limited speculation clarification – added hardware-generated speculative references (Section 4.4.6). PAL memory accesses and restrictions clarification (Section 11.9). PSP validity on INITs from PAL_MC_ERROR_INFO clarification (Section 11.8.3). Speculation attributes clarification (Section 4.4.6). PAL_A FIT entry, PAL_VM_TR_READ, PSP, PAL_VERSION clarifications (Sections 11.8.3 and 11.3.2.1). TLB searching clarifications (Section 4.1). IA-32 related changes (Section 10.3, Section 10.3.2, Section 10.3.2, Section 10.3.3.1, Section 10.10.1). 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).
		Volume 3: IA-32 CPUID clarification (p. 5-71). Revised figures for extract, deposit, and alloc instructions (Section 2.2). RCPPS, RCPSS, RSQRTPS, and RSQRTSS clarification (Section 7.12). IA-32 related changes (Section 5.3). tak, tpa change (Section 2.2).
July 2000	1.1	Volume 1: Processor Serial Number feature removed (Chapter 3). Clarification on exceptions to instruction dependency (Section 3.4.3).

Date of Revision	Revision Number	Description
		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).
January 2000	1.0	Initial release of document.

§

Instruction Reference 2

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

2.1 Instruction Page Conventions

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

Table 2-1. Instruction Page Description

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

Table 2-2. Instruction Page Font Conventions

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

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

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

(64-bits not including the NaT bit) where the notation `GR[addr]` is used. The syntactical differences between the code found in the Operation section and ANSI C is listed in Table 2-4.

Table 2-3. Register File Notation

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

Table 2-4. C Syntax Differences

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

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

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

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

2.2 Instruction Descriptions

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

add — Add

Format:

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

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

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

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

Operation:

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

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

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

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

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

Interruptions: Illegal Operation fault

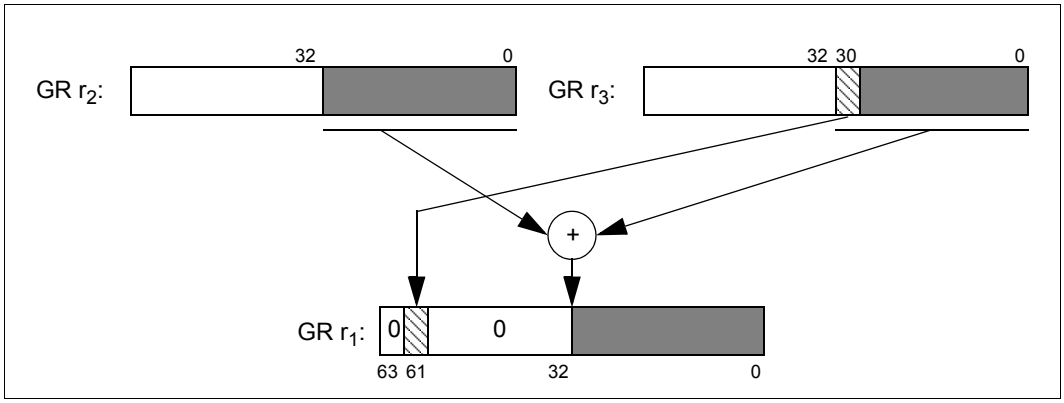
addp4 — Add Pointer

Format:

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

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

Figure 2-1. Add Pointer



Operation:

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

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

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

Interruptions: Illegal Operation fault

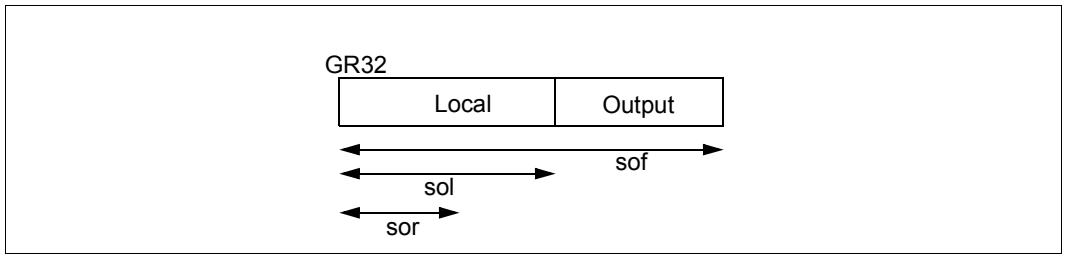
alloc — Allocate Stack Frame

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

Description: A new stack frame is allocated on the general register stack, and the Previous Function State register (PFS) is copied to GR r_1. The change of frame size is immediate. The write of GR r_1 and subsequent instructions in the same instruction group use the new frame.

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

Figure 2-2. Stack Frame



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

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

Although the assembler does not allow illegal combinations of operands for alloc, illegal combinations can be encoded in the instruction. Attempting to allocate a stack frame larger than 96 registers, or with the rotating region larger than the stack frame, or with the size of locals larger than the stack frame, or specifying a qualifying predicate other than PR 0, will cause an Illegal Operation fault.

This instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0; otherwise, the results are undefined.

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

Operation:

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

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

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

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

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

Operation:

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

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

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

Interruptions: Illegal Operation fault

andcm — And Complement

Format:

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

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

Operation:

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

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

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

Interruptions: Illegal Operation fault

br — Branch

Format:

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

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

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

Table 2-6. Branch Types

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

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

The branch type determines how the branch condition is calculated and whether the branch has other effects (such as writing a link register). For the basic branch types,

the branch condition is simply the value of the specified predicate register. These basic branch types are:

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

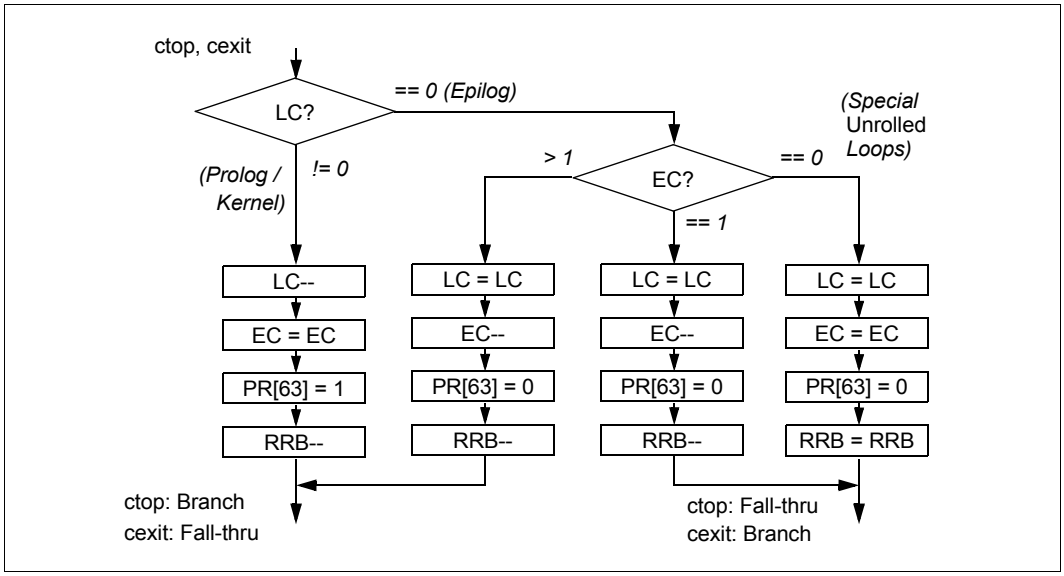
These branch types are of two categories: top and exit. The top types (ctop and wtop) are used when the loop decision is located at the bottom of the loop body and therefore a taken branch will continue the loop while a fall through branch will exit the loop. The exit types (cexit and wexit) are used 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.

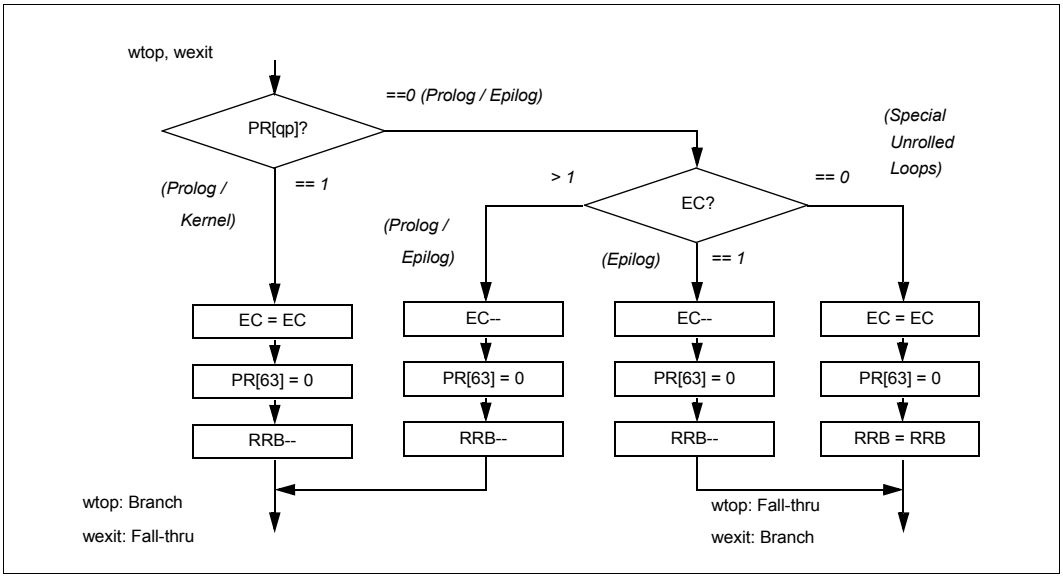
Figure 2-3. Operation of br.ctop and br.cexit



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.

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



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

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

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

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

Values for various branch hint completers are shown in the following tables. Whether Prediction Strategy hints are shown in Table 2-7. Sequential Prefetch hints are shown in Table 2-8. Branch Cache Deallocation hints are shown in Table 2-9. See Section 4.5.2, “Branch Prediction Hints” on page 1:78.

Table 2-7. Branch Whether Hint

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

Table 2-8. Sequential Prefetch Hint

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

Table 2-9. Branch Cache Deallocation Hint

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

Operation:

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

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

lower_priv_transition = 0;

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

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

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

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

    case 'ret':                             // return restores stack frame
```

```
    tmp_taken = PR[qp];
    if (tmp_taken) {
        // tmp_growth indicates the amount to move logical TOP *up*:
        // tmp_growth = sizeof(previous out) - sizeof(current frame)
        // a negative amount indicates a shrinking stack
        tmp_growth = (AR[PFS].pfm.sof - AR[PFS].pfm.sol) - CFM.sof;
        alat_frame_update(-AR[PFS].pfm.sol, 0);
        rse_fatal = rse_restore_frame(AR[PFS].pfm.sol,
                                      tmp_growth, CFM.sof);
        if (rse_fatal) {
        // See Section 6.4, "RSE Operation" on page 2:137
            CFM.sof = 0;
            CFM.sol = 0;
            CFM.sor = 0;
            CFM.rrb.gr = 0;
            CFM.rrb.fr = 0;
            CFM.rrb.pr = 0;
        } else // normal branch return
            CFM = AR[PFS].pfm;

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

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

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

case 'cloop':                           // simple counted loop
    if (slot != 2)
```

```
            illegal_operation_fault();
        tmp_taken = (AR[LC] != 0);
        if (AR[LC] != 0)
            AR[LC]--;
        break;

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

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

```
    taken_branch = 1;
    IP = tmp_IP;                          // set the new value for IP
    if (!impl_uia_fault_supported() &&
        ((PSR.it && unimplemented_virtual_address(tmp_IP, PSR.vm))
         || (!PSR.it && unimplemented_physical_address(tmp_IP))))
        unimplemented_instruction_address_trap(lower_priv_transition,
                                               tmp_IP);
    if (lower_priv_transition && PSR.lp)
        lower_privilege_transfer_trap();
    if (PSR.tb)
        taken_branch_trap();
}
```

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

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

break — Break

Format:

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

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

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

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

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

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

Operation:

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

    break_instruction_fault(immediate);
}
```

Interruptions: Break Instruction fault

brl — Branch Long

Format:

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

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

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

Table 2-10. Long Branch Types

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

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

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

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

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

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

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

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

brp — Branch Predict

Format:

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

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

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

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

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

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

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

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

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

Table 2-12. Indirect Branch Predict Whether Hint

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

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

Table 2-13. Importance Hint

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

Operation:

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

Interruptions: None

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:

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

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

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

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

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

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

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

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

Table 2-14. ALAT Clear Completer

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

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

Operation:

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

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

clrrrb — Clear RRB

Format:	clrrrb	all_form	B8
	clrrrb.pr	pred_form	B8

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

This instruction must be the last instruction in an instruction group; otherwise, operation is undefined.

This instruction cannot be predicated.

Operation:

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

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

Interruptions: None

clz — Count Leading Zeros

Format: (*qp*) clz $r_1 = r_3$ I9

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

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

Operation:

```
if (PR[qp])
    if (!instruction_implemented(CLZ))
        illegal_operation_fault();
    check_target_register(r1);

    tmp_val = 0;

    do {
        if (GR[r3]{63 - tmp_val} != 0) break;
    } while (tmp_val++ < 63);

    GR[r1] = tmp_val;
    GR[r1].nat = GR[r3].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 = imm_8, r_3	imm8_form	A8
(*qp*) cmp.*crel.ctype* p_1, p_2 = r0, r_3	parallel_inequality_form	A7
(*qp*) cmp.*crel.ctype* p_1, p_2 = r_3, r0	pseudo-op	

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

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

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

Table 2-15. Comparison Types

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

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

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

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

simply uses the negative relation with an implemented type. The implemented relations and how the pseudo-ops map onto them are shown in Table 2-16 (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

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

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

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

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

Operation:

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

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

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

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

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

Interruptions: Illegal Operation fault

cmp4 — Compare 4 Bytes

Format:

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

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

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

Operation:

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

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

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

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

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

```
                PR[p2] = 0;
            }
            break;
        case 'unc':                                     // unc-type compare
        default:                                        // normal compare
            if (tmp_nat) {
                PR[p1] = 0;
                PR[p2] = 0;
            } else {
                PR[p1] = tmp_rel;
                PR[p2] = !tmp_rel;
            }
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Interruptions: Illegal Operation fault

cmpxchg — Compare and Exchange

Format:

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

(*qp*) cmp8xchg16.*sem*.*ldhint* r_1 = [r_3], r_2, ar.csd, ar.ccv sixteen_byte_form M16

Description:

A value consisting of *sz* bytes (8 bytes for `cmp8xchg16`) is read from memory starting at the address specified by the value in GR r_3. The value is zero extended and compared with the contents of the `cmpxchg` Compare Value application register (AR[CCV]). If the two are equal, then the least significant *sz* bytes of the value in GR r_2 are written to memory starting at the address specified by the value in GR r_3. For `cmp8xchg16`, if the two are equal, then 8-bytes from GR r_2 are stored at the specified address ignoring bit 3 (GR r_3 & ~0x8), and 8 bytes from the Compare and Store Data application register (AR[CSD]) are stored at that address + 8 ((GR r_3 & ~0x8) + 8). The zero-extended value read from memory is placed in GR r_1 and the NaT bit corresponding to GR r_1 is cleared.

The values of the *sz* completer are given in Table 2-19. The *sem* completer specifies the type of semaphore operation. These operations are described in Table 2-20. See Section 4.4.7, "Sequentiality Attribute and Ordering" on page 2:82 for details on memory ordering.

Table 2-19. Memory Compare and Exchange Size

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

Table 2-20. Compare and Exchange Semaphore Types

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

If the address specified by the value in GR r_3 is not naturally aligned to the size of the value being accessed in memory, an Unaligned Data Reference fault is taken independent of the state of the User Mask alignment checking bit, UM.ac (PSR.ac in the Processor Status Register). For the `cmp8xchg16` instruction, the address specified must be 8-byte aligned.

The memory read and write are guaranteed to be atomic. For the `cmp8xchg16` instruction, the 8-byte memory read and the 16-byte memory write are guaranteed to be atomic.

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

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

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 2-34 on page 3:152. Locality hints do not

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

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

Operation:

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

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

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

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

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

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

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

Interruptions:

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

cover — Cover Stack Frame

Format: cover B8

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

A `cover` instruction must be the last instruction in an instruction group; otherwise, operation is undefined.

This instruction cannot be predicated.

Operation:

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

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

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

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

Interruptions: Virtualization fault

czx — Compute Zero Index

Format:

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

Description: GR r_3 is scanned for a zero element. The element is either an 8-bit aligned byte (one_byte_form) or a 16-bit aligned pair of bytes (two_byte_form). The index of the first zero element is placed in GR r_1. If there are no zero elements in GR r_3, a default value is placed in GR r_1. Table 2-21 gives the possible result values. In the left_form, the source is scanned from most significant element to least significant element, and in the right_form it is scanned from least significant element to most significant element.

Table 2-21. Result Ranges for czx

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

Operation:

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

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

```
            else if ((GR[r3] & 0x0000ffff00000000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0xffff000000000000) == 0) GR[r1] = 3;
            else GR[r1] = 4;
        }
    }
    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

dep — Deposit

Format:

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

Description:

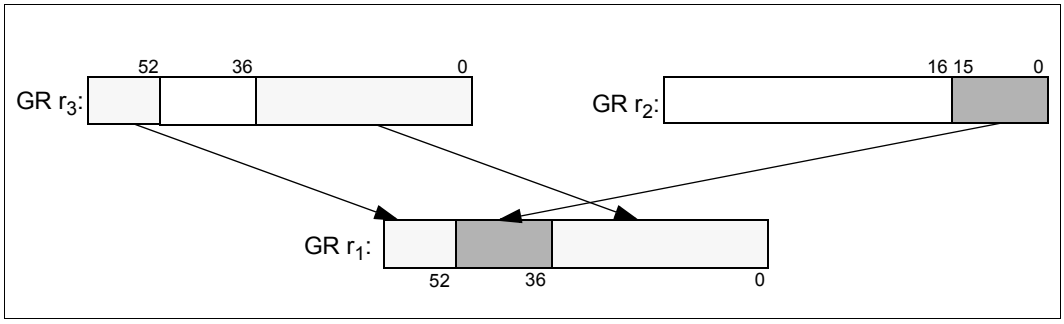
In the merge_form, a right justified bit field taken from the first source operand is deposited into the value in GR r_3 at an arbitrary bit position and the result is placed in GR r_1. In the register_form the first source operand is GR r_2; and in the imm_form it is the sign-extended value specified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

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

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

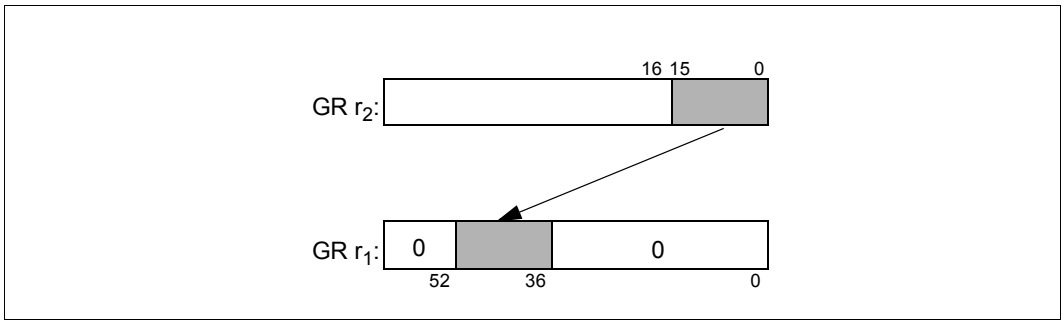
The operation of `dep r1 = r2, r3, 36, 16` is illustrated in Figure 2-5.

Figure 2-5. Deposit Example (merge_form)



The operation of `dep.z r1 = r2, 36, 16` is illustrated in Figure 2-6.

Figure 2-6. Deposit Example (zero_form)



Operation:

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

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

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

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

Interruptions: Illegal Operation fault

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

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

The operation of `extr 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 u> 64)
        tmp_len = 64 - pos6;

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

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

Interruptions: Illegal Operation fault

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.

Operation: See "fmerge — Floating-point Merge" on page 3:80.

fadd — Floating-point Add

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

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

The mnemonic values for the opcode's *pc* are given in Table 2-22. 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

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

Table 2-23. *sf* Mnemonic Values

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

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

famax — Floating-point Absolute Maximum

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

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

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

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

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

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

Operation:

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

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

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

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr(f1);
}
```

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

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

famin — Floating-point Absolute Minimum

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

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

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

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

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

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

Operation:

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

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

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

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr(f1);
}
```

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

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

fand — Floating-point Logical And

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

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

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

Operation:

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

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

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fandcm — Floating-point And Complement

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

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

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

Operation:

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

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

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fc — Flush Cache

Format:

(*qp*) fc r_3	invalidate_line_form	M28
(*qp*) fc.i r_3	instruction_cache_coherent_form	M28

Description: In the invalidate_line form, the cache line associated with the address specified by the value of GR r_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:

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

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

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

fchkf — Floating-point Check Flags

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

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

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

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

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

Operation:

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

FP Exceptions: None

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, $fclass_9$ F5

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

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

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

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

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

Operation:

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

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

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

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

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

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fclrf — Floating-point Clear Flags

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

Description: The status field's 6-bit flags field is reset to zero.
The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

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

FP Exceptions: None

Interruptions: None

fcmp — Floating-point Compare

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

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

Table 2-26. Floating-point Comparison Types

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

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

The relations are defined for each of the comparison types in Table 2-27. Of the twelve relations, not all are directly implemented in hardware. Some are actually pseudo-ops. For these, the assembler simply switches the source operand specifiers and/or switches the predicate target specifiers and uses an implemented relation.

Table 2-27. Floating-point Comparison Relations

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

Operation:

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

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

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

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

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

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

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

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

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

fcvt.fx — Convert Floating-point to Integer

Format:

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

Description:

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

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

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

Operation:

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

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

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

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

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

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

fcvt.xf — Convert Signed Integer to Floating-point

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

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

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

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

Operation:

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

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

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

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

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fcvt.xuf — Convert Unsigned Integer to Floating-point

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

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

Note: Multiplying FR f_3 with FR 1 (a 1.0) normalizes the canonical representation of an integer in the floating-point register file producing a normal floating-point value.

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

The mnemonic values for the opcode's *pc* are given in Table 2-22 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* r_1 = [r_3], inc_3	four_byte_form	M17
(*qp*) fetchadd8.*sem.ldhint* r_1 = [r_3], inc_3	eight_byte_form	M17

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

The *sem* completer specifies the type of semaphore operation. These operations are described in Table 2-28. See Section 4.4.7, "Sequentiality Attribute and Ordering" on page 2:82 for details on memory ordering.

Table 2-28. Fetch and Add Semaphore Types

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

The memory read and write are guaranteed to be atomic for accesses to pages with cacheable, writeback memory attribute. For accesses to other memory types, atomicity is platform dependent. Details on memory attributes are described in Section 4.4, "Memory Attributes" on page 2:75.

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

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

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

On a processor model that supports exported `fetchadd`, a `fetchadd` to a UCE page causes the fetch-and-add operation to be exported outside of the processor; if the platform does not support exported `fetchadd`, the operation is undefined. On a processor model that does not support exported `fetchadd`, a `fetchadd` to a UCE page causes an Unsupported Data Reference fault. See Section 4.4.9, "Effects of Memory Attributes on Memory Reference Instructions" on page 2:86.

The value of the *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_unused);
    if (!ma_supports_fetchadd(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

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

    alat_inval_multiple_entries(paddr, size);

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

Interruptions:

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

flushrs — Flush Register Stack

Format: flushrs M25

Description: All stacked general registers in the dirty partition of the register stack are written to the backing store before execution continues. The dirty partition contains registers from previous procedure frames that have not yet been saved to the backing store. For a description of the register stack partitions, refer to Chapter 6, "Register Stack Engine" in Volume 2. A pending external interrupt can interrupt the RSE store loop when enabled.

After this instruction completes execution BSPSTORE is equal to BSP.

This instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0; otherwise, the results are undefined. This instruction cannot be predicated.

Operation:

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

Interruptions:

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

fma — Floating-point Multiply Add

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

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

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

If f_2 is f0, an IEEE multiply operation is performed instead of a multiply and add. See "fmpy — Floating-point Multiply" on page 3:85.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation:

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

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

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

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

FP Exceptions:

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

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

fmax — Floating-point Maximum

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

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

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

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

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

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

Operation:

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

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

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

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

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

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

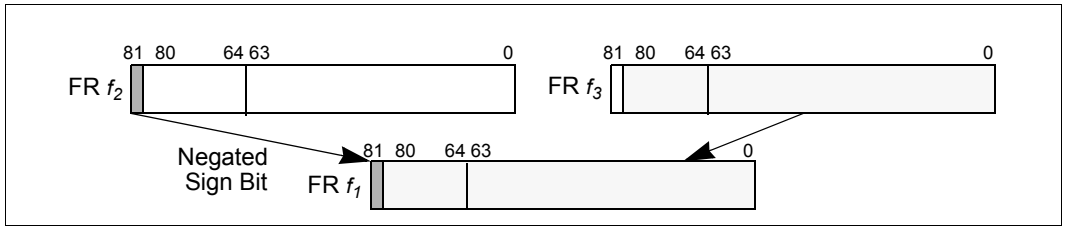


Figure 2-9. Floating-point Merge Sign Operation

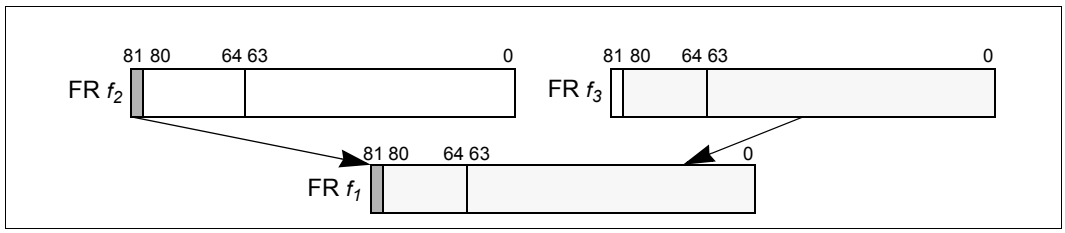
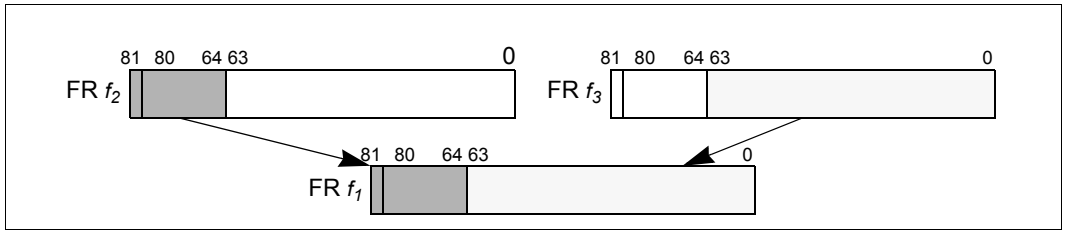


Figure 2-10. Floating-point Merge Sign and Exponent Operation



Operation:

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

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

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fmin — Floating-point Minimum

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

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

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

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

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

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

Operation:

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

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

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

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

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

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

fmix — Floating-point Mix

Format:

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

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

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

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

Figure 2-11. Floating-point Mix Left

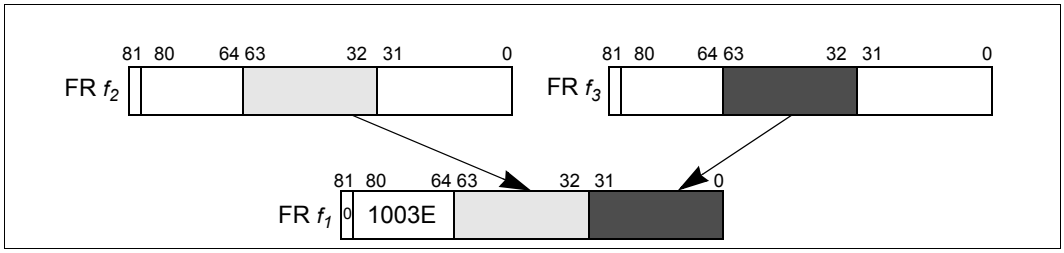


Figure 2-12. Floating-point Mix Right

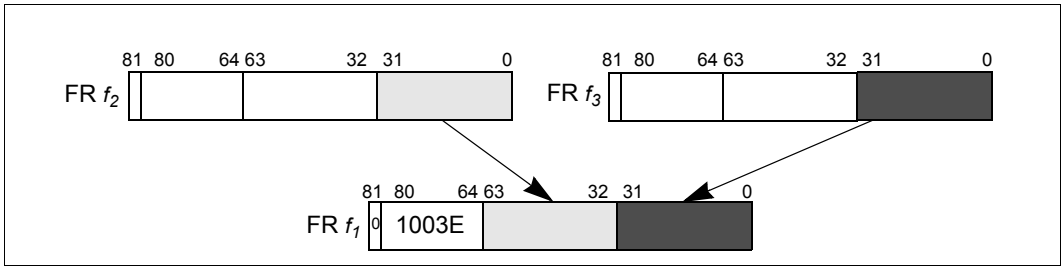
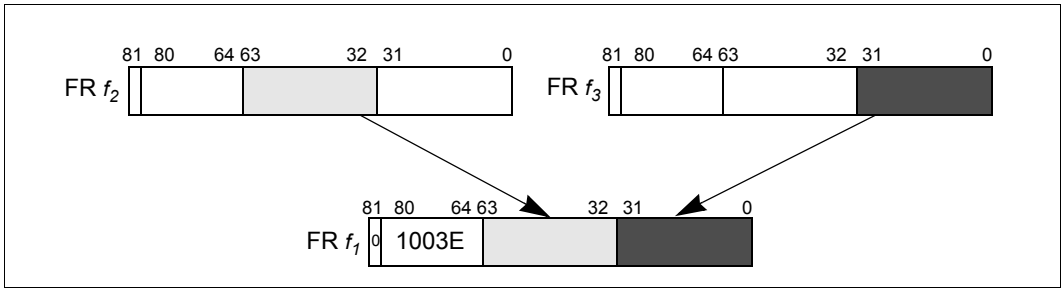


Figure 2-13. Floating-point Mix Left-Right



Operation:

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

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

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fmpy — Floating-point Multiply

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

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

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

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

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

fms — Floating-point Multiply Subtract

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

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

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

If f_2 is f0, an IEEE multiply operation is performed instead of a multiply and subtract. See "fmpy — Floating-point Multiply" on page 3:85.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation:

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

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

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

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

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

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

fneg — Floating-point Negate

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

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

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

Operation: See "fmerge — Floating-point Merge" on page 3:80.

fnegabs — Floating-point Negate Absolute Value

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

Description: The absolute value of the value in FR f_3 is computed, negated, and placed in FR f_1. If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See "fmerge — Floating-point Merge" on page 3:80.

fnma — Floating-point Negative Multiply Add

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

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

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

If f_2 is f0, an IEEE multiply operation is performed, followed by negation of the product. See "fnmpy — Floating-point Negative Multiply" on page 3:92.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation:

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

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

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

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

FP Exceptions:

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

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

fnmpy — Floating-point Negative Multiply

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

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

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

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation: See "fnma — Floating-point Negative Multiply Add" on page 3:90.

fnorm — Floating-point Normalize

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

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

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

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

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

for — Floating-point Logical Or

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

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

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

Operation:

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

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

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fpabs — Floating-point Parallel Absolute Value

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

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

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

Operation: See "fpmerge — Floating-point Parallel Merge" on page 3:111.

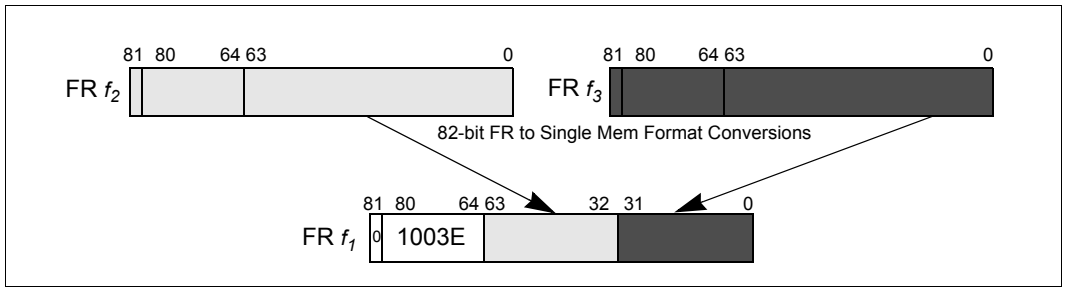
fpack — Floating-point Pack

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

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

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

Figure 2-14. Floating-point Pack



Operation:

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

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

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

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fpamax — Floating-point Parallel Absolute Maximum

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

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

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

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

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

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

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

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

Operation:

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

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

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

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

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

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

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

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

fpamin — Floating-point Parallel Absolute Minimum

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

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

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

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

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

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

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

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

Operation:

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

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

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

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

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

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

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

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

fpcmp — Floating-point Parallel Compare

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

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

Table 2-29. Floating-point Parallel Comparison Results

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

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

The relations are defined for each of the comparison types in Table 2-29. Of the twelve relations, not all are directly implemented in hardware. Some are actually pseudo-ops. For these, the assembler simply switches the source operand specifiers and/or switches the predicate type specifiers and uses an implemented relation.

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

Table 2-30. Floating-point Parallel Comparison Relations

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

Operation:

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

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

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

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

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

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

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

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

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

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

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

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

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

fpcvt.fx — Convert Parallel Floating-point to Integer

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

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

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

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

Operation:

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

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

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

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

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

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

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

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

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

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

fpma — Floating-point Parallel Multiply Add

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

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

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

Note: If f_2 is f0 in the fpma instruction, just the IEEE multiply operation is performed. (See "fpmpy — Floating-point Parallel Multiply" on page 3:115.) FR f1, as an operand, is not a packed pair of 1.0 values, it is just the register file format's 1.0 value.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 1:90.

Operation:

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

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

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

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

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

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

FP Exceptions:

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

Interruptions:

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

fpmax — Floating-point Parallel Maximum

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

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

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

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

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

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

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

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

Operation:

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

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

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

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

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

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

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

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

fpmerge — Floating-point Parallel Merge

Format:

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

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

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

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

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

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

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

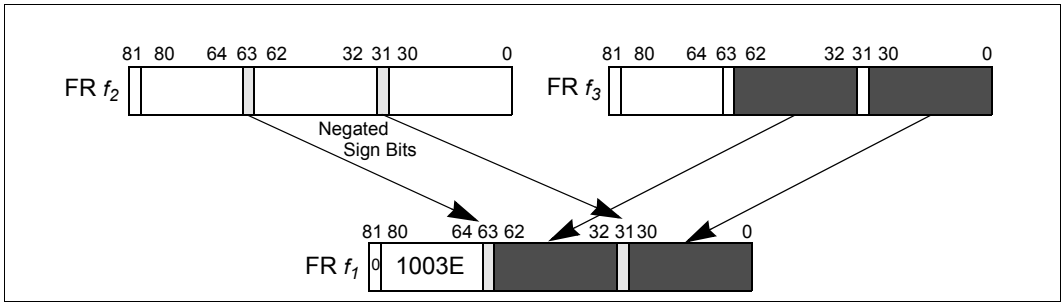


Figure 2-16. Floating-point Parallel Merge Sign Operation

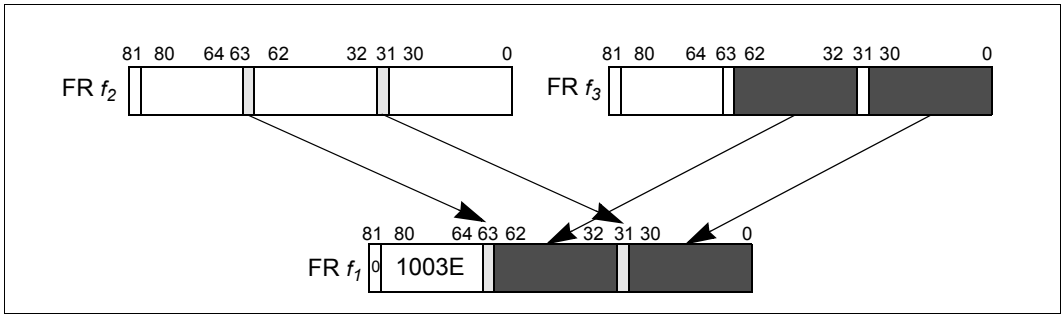
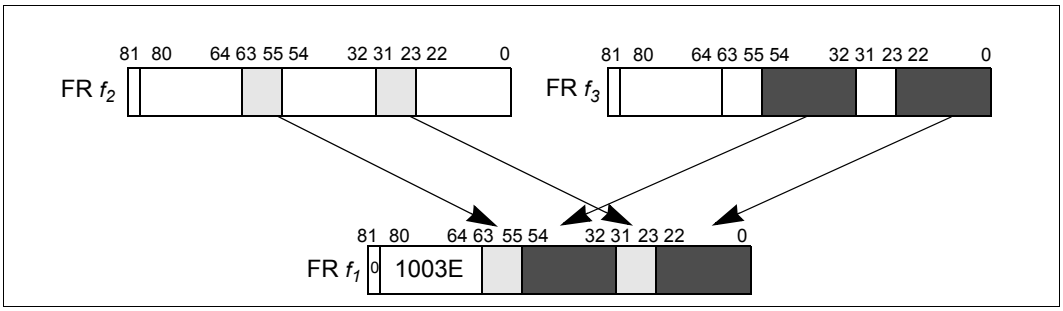


Figure 2-17. Floating-point Parallel Merge Sign and Exponent Operation



Operation:

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

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

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

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fpmin — Floating-point Parallel Minimum

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

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

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

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

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

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

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

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

Operation:

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

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

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

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

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

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

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

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

fpmpy — Floating-point Parallel Multiply

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

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

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

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 1:90.

Operation: See "fpma — Floating-point Parallel Multiply Add" on page 3:107.

fpms — Floating-point Parallel Multiply Subtract

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

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

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

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

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 1:90.

Operation:

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

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

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

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

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

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

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

FP Exceptions:

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

Interruptions:

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

fpneg — Floating-point Parallel Negate

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

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

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

Operation: See "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 $f_1 = f0, f_3$

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

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

Operation: See "fpmerge — Floating-point Parallel Merge" on page 3:111.

fpnma — Floating-point Parallel Negative Multiply Add

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

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

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

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

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 1:90.

Operation:

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

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

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

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

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

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

FP Exceptions:

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

Interruptions:

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

fpnmpy — Floating-point Parallel Negative Multiply

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

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

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

The mnemonic values for *sf* are given in Table 2-23 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 F6

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

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

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

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

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

Operation:

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

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

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

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

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

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

FP Exceptions: Invalid Operation (V)
Zero Divide (Z)

Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

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

fprsqrta — Floating-point Parallel Reciprocal Square Root Approximation

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

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

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

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

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

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

Operation:

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

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

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

```
                    if (limits_check.hi)
                        tmp_pred_hi = 0;
                    else
                        tmp_pred_hi = 1;
                }
                tmp_res_hi = fp_single(tmp_res);
            }

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

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

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

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

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

frcpa — Floating-point Reciprocal Approximation

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

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

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

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

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

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

Operation:

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

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

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

```
            } else {
                FR[f1] = fp_ieee_recip(den);
                PR[p2] = 1;
            }
        }
        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}

// fp_ieee_recip()

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

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

```
    return (tmp_res);
}
```

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

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

frsqrta — Floating-point Reciprocal Square Root Approximation

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

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

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

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

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

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

Operation:

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

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

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

```
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}

// fp_ieee_recip_sqrt()

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

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

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

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

fselect — Floating-point Select

Format: (*qp*) fselect $f_1 = f_3, f_4, f_2$ F3

Description: The significand field of FR f_3 is logically AND-ed with the significand field of FR f_2 and the significand field of FR f_4 is logically AND-ed with the one's complement of the significand field of FR f_2. The two results are logically OR-ed together. The result is placed in the significand field of FR f_1.

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

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

Operation:

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

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand   = (FR[f3].significand & FR[f2].significand)
                             | (FR[f4].significand & ~FR[f2].significand);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fsetc — Floating-point Set Controls

Format: (*qp*) fsetc.*sf* *amask*$_7$, *omask*$_7$ F12

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

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

Operation:

```
if (PR[qp]) {
    tmp_controls = (AR[FPSR].sf0.controls & amask7) | omask7;
    if (is_reserved_field(FSETC, sf, tmp_controls))
        reserved_register_field_fault();
    fp_set_sf_controls(sf, tmp_controls);
}
```

FP Exceptions: None

Interruptions: Reserved Register/Field fault

fsub — Floating-point Subtract

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

Description: FR f_2 is subtracted from FR f_3 (computed to infinite precision), rounded to the precision indicated by *pc* (and possibly FPSR.*sf.pc* and FPSR.*sf.wre*) using the rounding mode specified by FPSR.*sf.rc*, and placed in FR f_1.

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

The mnemonic values for the opcode's *pc* are given in Table 2-22 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*) fswap $f_1 = f_2, f_3$	swap_form	F9
(*qp*) fswap.nl $f_1 = f_2, f_3$	swap_nl_form	F9
(*qp*) fswap.nr $f_1 = f_2, f_3$	swap_nr_form	F9

Description: For the swap_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3. The concatenated pair is then swapped.

For the swap_nl_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3. The concatenated pair is then swapped, and the left single precision value is negated.

For the swap_nr_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3. The concatenated pair is then swapped, and the right single precision value is negated.

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

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

Figure 2-18. Floating-point Swap

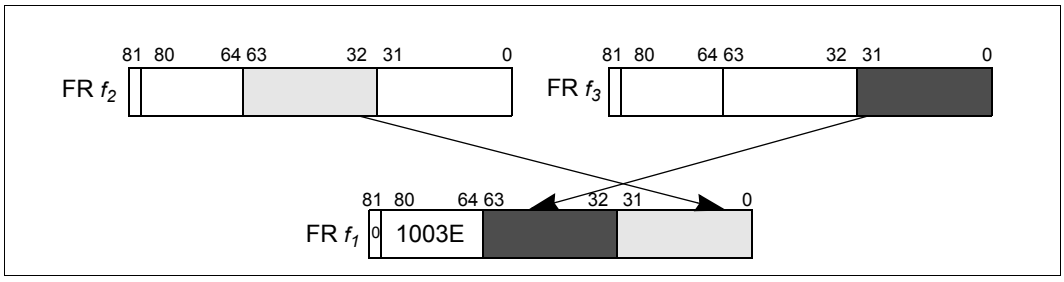


Figure 2-19. Floating-point Swap Negate Left

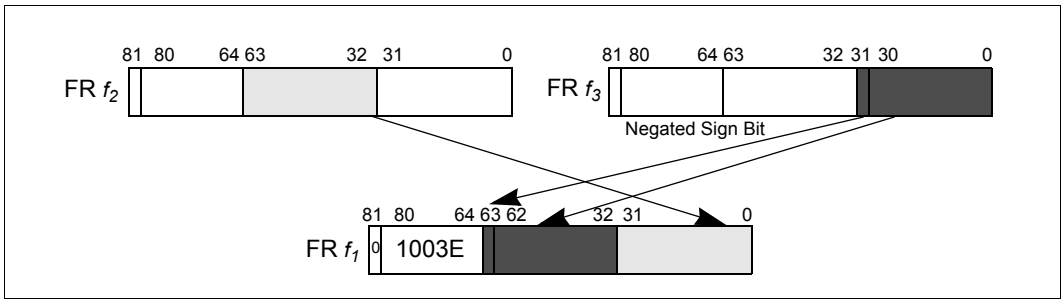
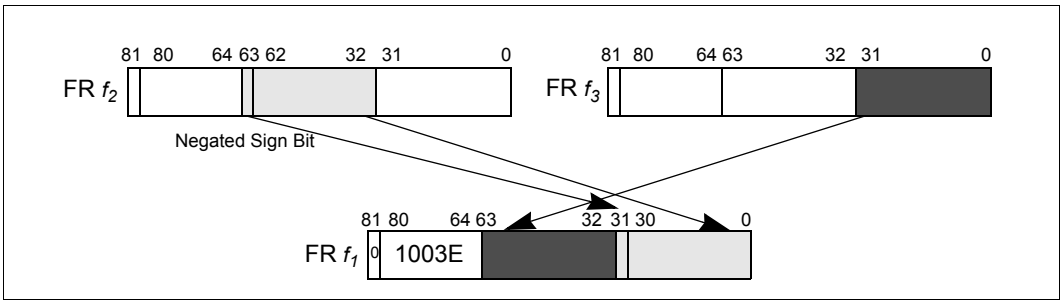


Figure 2-20. Floating-point Swap Negate Right



Operation:

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

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (swap_form) {
            tmp_res_hi = FR[f3].significand{31:0};
            tmp_res_lo = FR[f2].significand{63:32};
        } else if (swap_nl_form) {
            tmp_res_hi = (!FR[f3].significand{31} << 31)
                       | (FR[f3].significand{30:0});
            tmp_res_lo = FR[f2].significand{63:32};
        } else { // swap_nr_form
            tmp_res_hi = FR[f3].significand{31:0};
            tmp_res_lo = (!FR[f2].significand{63} << 31)
                       | (FR[f2].significand{62:32});
        }

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

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fsxt — Floating-point Sign Extend

Format:

(*qp*) fsxt.l $f_1 = f_2, f_3$	sxt_l_form	F9
(*qp*) fsxt.r $f_1 = f_2, f_3$	sxt_r_form	F9

Description: For the sxt_l_form (sxt_r_form), the sign of the left (right) single precision value in FR f_2 is extended to 32-bits and is concatenated with the left (right) single precision value in FR f_3.

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

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

Figure 2-21. Floating-point Sign Extend Left

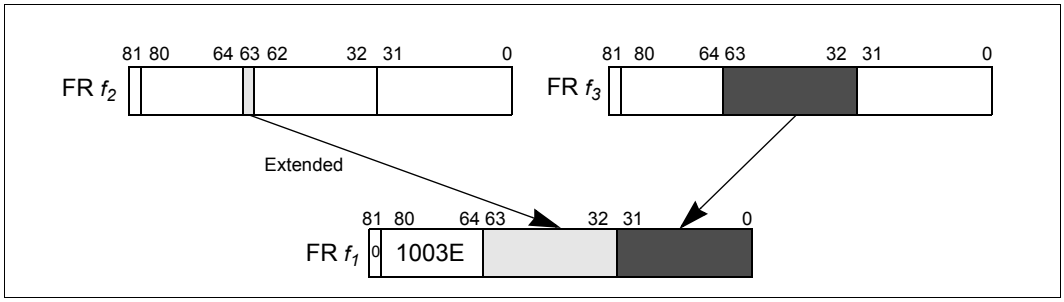
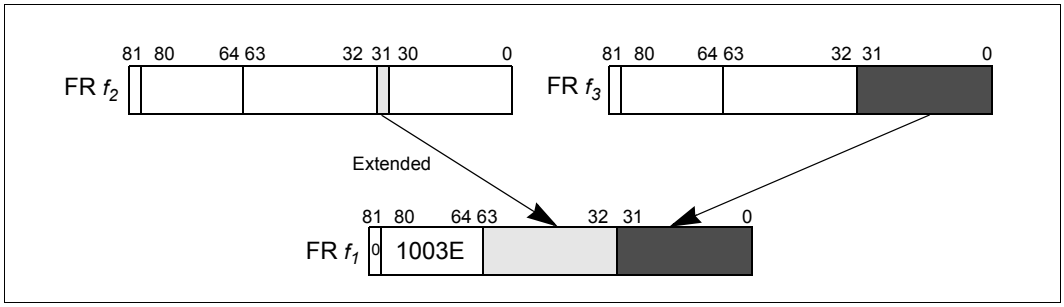


Figure 2-22. Floating-point Sign Extend Right



Operation:

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

    if (fp_is_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 M24

Description: The processor is instructed to expedite flushing of any pending stores held in write or coalescing buffers. Since this operation is a hint, the processor may or may not take any action and actually flush any outstanding stores. The processor gives no indication when flushing of any prior stores is completed. An `fwb` instruction does not ensure ordering of stores, since later stores may be flushed before prior stores.

To ensure prior coalesced stores are made visible before later stores, software must issue a release operation between stores (see Table 4-15 on page 2:83 for a list of release operations).

This instruction can be used to help ensure stores held in write or coalescing buffers are not delayed for long periods or to expedite high priority stores out of the processors.

Operation:

```
if (PR[qp]) {
   mem_flush_pending_stores();
}
```

Interruptions: None

fxor — Floating-point Exclusive Or

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

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

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

Operation:

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

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

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

getf — Get Floating-point Value or Exponent or Significand

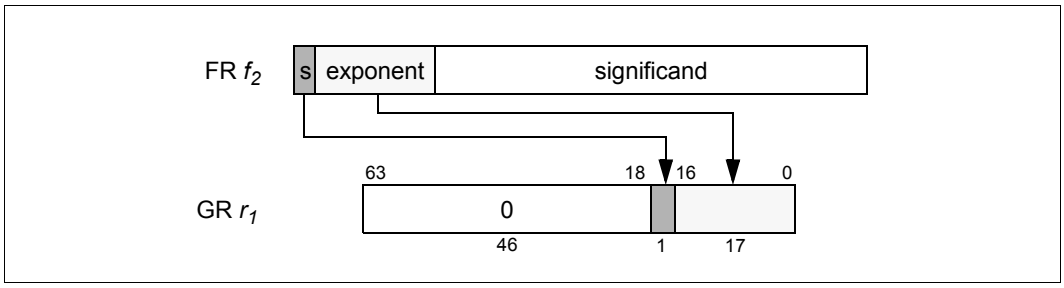
Format:

(*qp*) getf.s $r_1 = f_2$	single_form	M19
(*qp*) getf.d $r_1 = f_2$	double_form	M19
(*qp*) getf.exp $r_1 = f_2$	exponent_form	M19
(*qp*) getf.sig $r_1 = f_2$	significand_form	M19

Description: In the single and double forms, the value in FR f_2 is converted into a single precision (single_form) or double precision (double_form) memory representation and placed in GR r_1, as shown in Figure 5-7 and Figure 5-8 on page 1:95, respectively. In the single_form, the most-significant 32 bits of GR r_1 are set to 0.

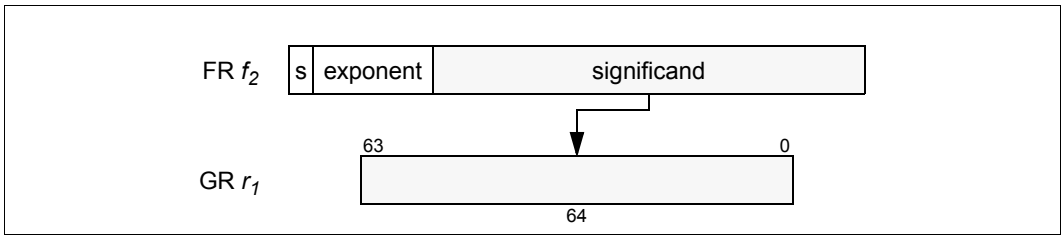
In the exponent_form, the exponent field of FR f_2 is copied to bits 16:0 of GR r_1 and the sign bit of the value in FR f_2 is copied to bit 17 of GR r_1. The most-significant 46-bits of GR r_1 are set to zero.

Figure 2-23. Function of getf.exp



In the significand_form, the significand field of the value in FR f_2 is copied to GR r_1

Figure 2-24. Function of getf.sig



For all forms, if FR f_2 contains a NaTVal, then the NaT bit corresponding to GR r_1 is set to 1.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);
    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (single_form) {
        GR[r1]{31:0} = fp_fr_to_mem_format(FR[f2], 4, 0);
        GR[r1]{63:32} = 0;
    } else if (double_form) {
        GR[r1] = fp_fr_to_mem_format(FR[f2], 8, 0);
    } else if (exponent_form) {
        GR[r1]{63:18} = 0;
        GR[r1]{16:0} = FR[f2].exponent;
        GR[r1]{17} = FR[f2].sign;
    } else // significand_form
        GR[r1] = FR[f2].significand;
    if (fp_is_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}	pseudo-op	
(*qp*) hint.i imm_{21}	i_unit_form	I18
(*qp*) hint.b imm_{21}	b_unit_form	B9
(*qp*) hint.m imm_{21}	m_unit_form	M48
(*qp*) hint.f imm_{21}	f_unit_form	F16
(*qp*) hint.x imm_{62}	x_unit_form	X5

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

The immediate, imm_{21} or imm_{62}, specifies the hint. For the 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 pseudo-op can be used if the unit type to execute on is unimportant.

Table 2-31. Hint Immediates

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

Operation:

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

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

Interruptions: None

invala — Invalidate ALAT

Format:

(*qp*) invala	complete_form	M24
(*qp*) invala.e r_1	gr_form, entry_form	M26
(*qp*) invala.e f_1	fr_form, entry_form	M27

Description: The selected entry or entries in the ALAT are invalidated.

In the complete_form, all ALAT entries are invalidated. In the entry_form, the ALAT is queried using the general register specifier r_1 (gr_form), or the floating-point register specifier f_1 (fr_form), and if any ALAT entry matches, it is invalidated.

Operation:

```
if (PR[qp]) {
    if (complete_form)
        alat_inval();
    else { // entry_form
        if (gr_form)
            alat_inval_single_entry(GENERAL, r1);
        else // fr_form
            alat_inval_single_entry(FLOAT, f1);
    }
}
```

Interruptions: None

itc — Insert Translation Cache

Format:

(*qp*) itc.i r_2	instruction_form	M41
(*qp*) itc.d r_2	data_form	M41

Description: An entry is inserted into the instruction or data translation cache. GR r_2 specifies the physical address portion of the translation. ITIR specifies the protection key, page size and additional information. The virtual address is specified by the IFA register and the region register is selected by IFA{63:61}. The processor determines which entry to replace based on an implementation-specific replacement algorithm.

The visibility of the `itc` instruction to externally generated purges (`ptc.g`, `ptc.ga`) must occur before subsequent memory operations. From a software perspective, this is similar to acquire semantics. Serialization is still required to observe the side-effects of a translation being present.

`itc` must be the last instruction in an instruction group; otherwise, its behavior (including its ordering semantics) is undefined.

The TLB is first purged of any overlapping entries as specified by Table 4-1 on page 2:52.

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

To ensure forward progress, software must ensure that PSR.ic remains 0 until `rfi`-ing to the instruction that requires the translation.

Operation:

```
if (PR[qp]) {
    if (!followed_by_stop())
        undefined_behavior();
    if (PSR.ic)
        illegal_operation_fault();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r2].nat)
        register_nat_consumption_fault(0);

    tmp_size = CR[ITIR].ps;
    tmp_va = CR[IFA]{60:0};
    tmp_rid = RR[CR[IFA]{63:61}].rid;
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    if (is_reserved_field(TLB_TYPE, GR[r2], CR[ITIR]))
        reserved_register_field_fault();
    if (!impl_check_mov_ifa() &&
           unimplemented_virtual_address(CR[IFA], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    if (instruction_form) {
        tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        slot = tlb_replacement_algorithm(ITC_TYPE);
        tlb_insert_inst(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TC);
    } else {                                          // data_form
        tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        slot = tlb_replacement_algorithm(DTC_TYPE);
        tlb_insert_data(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TC);
    }
}
```

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.

itr — Insert Translation Register

Format:

(*qp*) itr.i itr[r_3] = r_2	instruction_form	M42
(*qp*) itr.d dtr[r_3] = r_2	data_form	M42

Description: A translation is inserted into the instruction or data translation register specified by the contents of GR r_3. GR r_2 specifies the physical address portion of the translation. ITIR specifies the protection key, page size and additional information. The virtual address is specified by the IFA register and the region register is selected by IFA{63:61}.

As described in Table 4-1, "Purge Behavior of TLB Inserts and Purges" on page 2:52, the TLB is first purged of any entries that overlap with the newly inserted translation. The translation previously contained in the TR slot specified by GR r_3 is not necessarily purged from the processor's TLBs and may remain as a TC entry. To ensure that the previous TR translation is purged, software must use explicit `ptr` instructions before inserting the new TR entry.

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

Operation:

```
if (PR[qp]) {
    if (PSR.ic)
        illegal_operation_fault();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);

    slot = GR[r3]{7:0};
    tmp_size = CR[ITIR].ps;
    tmp_va = CR[IFA]{60:0};
    tmp_rid = RR[CR[IFA]{63:61}].rid;
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    tmp_tr_type = instruction_form ? ITR_TYPE : DTR_TYPE;

    if (is_reserved_reg(tmp_tr_type, slot))
        reserved_register_field_fault();
    if (is_reserved_field(TLB_TYPE, GR[r2], CR[ITIR]))
        reserved_register_field_fault();
    if (!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:

(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3]	no_base_update_form	M2
(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3], r_2	reg_base_update_form	M2
(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3], imm_9	imm_base_update_form	M3
(*qp*) ld16.*ldhint* r_1, ar.csd = [r_3]	sixteen_byte_form, no_base_update_form	M2
(*qp*) ld16.acq.*ldhint* r_1, ar.csd = [r_3]	sixteen_byte_form, acquire_form, no_base_update_form	M2
(*qp*) ld8.fill.*ldhint* r_1 = [r_3]	fill_form, no_base_update_form	M2
(*qp*) ld8.fill.*ldhint* r_1 = [r_3], r_2	fill_form, reg_base_update_form	M2
(*qp*) ld8.fill.*ldhint* r_1 = [r_3], imm_9	fill_form, imm_base_update_form	M3

Description:

A value consisting of *sz* bytes is read from memory starting at the address specified by the value in GR r_3. The value is then zero extended and placed in GR r_1. The values of the *sz* completer are given in Table 2-32. The NaT bit corresponding to GR r_1 is cleared, except as described below for speculative loads. The *ldtype* completer specifies special load operations, which are described in Table 2-33.

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 (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 `ld16` instruction.

Table 2-32. *sz* Completers

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

Table 2-33. Load Types

ldtype Completer	Interpretation	Special Load Operation
none	Normal load	
s	Speculative load	Certain exceptions may be deferred rather than generating a fault. Deferral causes the target register's NaT bit to be set. The NaT bit is later used to detect deferral.
a	Advanced load	An entry is added to the ALAT. This allows later instructions to check for colliding stores. If the referenced data page has a non-speculative attribute, the target register and NaT bit is cleared, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.

Table 2-33. Load Types (Continued)

ldtype Completer	Interpretation	Special Load Operation
sa	Speculative Advanced load	An entry is added to the ALAT, and certain exceptions may be deferred. Deferral causes the target register's NaT bit to be set, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.
c.nc	Check load – no clear	The ALAT is searched for a matching entry. If found, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a load is performed, and an entry is added to the ALAT (unless the referenced data page has a non-speculative attribute, in which case no ALAT entry is allocated).
c.clr	Check load – clear	The ALAT is searched for a matching entry. If found, the entry is removed, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a clear check load behaves like a normal load.
c.clr.acq	Ordered check load – clear	This type behaves the same as the unordered clear form, except that the ALAT lookup (and resulting load, if no ALAT entry is found) is performed with acquire semantics.
acq	Ordered load	An ordered load is performed with acquire semantics.
bias	Biased load	A hint is provided to the implementation to acquire exclusive ownership of the accessed cache line.

For 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

ldhint Completer	Interpretation
none	Temporal locality, level 1

Table 2-34. Load Hints (Continued)

ldhint Completer	Interpretation
nt1	No temporal locality, level 1
nta	No temporal locality, all levels

In the no_base_update form, the value in GR r_3 is not modified and no prefetch hint is implied.

For the base update forms, specifying the same register address in r_1 and r_3 will cause an Illegal Operation fault.

Hardware support for `ld16` instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `ld16` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.

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

Operation:

```
if (PR[qp]) {
    size = fill_form ? 8 : (sixteen_byte_form ? 16 : sz);

    speculative = (ldtype == 's' || ldtype == 'sa');
    advanced = (ldtype == 'a' || ldtype == 'sa');
    check_clear = (ldtype == 'c.clr' || ldtype == 'c.clr.acq');
    check_no_clear = (ldtype == 'c.nc');
    check = check_clear || check_no_clear;
    acquire = (acquire_form || ldtype == 'acq' || ldtype == 'c.clr.acq');
    otype = acquire ? ACQUIRE : UNORDERED;
    bias = (ldtype == 'bias') ? BIAS : 0 ;
    translate_address = 1;
    read_memory = 1;

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

    if (sixteen_byte_form && !instruction_implemented(LD16))
        illegal_operation_fault();
    if ((reg_base_update_form || imm_base_update_form) && (r1 == r3))
        illegal_operation_fault();
    check_target_register(r1);
    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);

    if (reg_base_update_form) {
        tmp_r2 = GR[r2];
        tmp_r2nat = GR[r2].nat;
    }

    if (!speculative && GR[r3].nat)              // fault on NaT address
        register_nat_consumption_fault(itype);
    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

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

```
                val = mem_read(paddr, size, UM.be, mattr, otype,
                               bias | ldhint);
            }
        }
    }
    if (check_clear || advanced)              // remove any old ALAT entry
        alat_inval_single_entry(GENERAL, r1);
    if (defer) {
        if (speculative) {
            GR[r1] = natd_gr_read(paddr, size, UM.be, mattr, otype,
                                  bias | ldhint);
            GR[r1].nat = 1;
        } else {
            GR[r1] = 0;                     // ld.a to sequential memory
            GR[r1].nat = 0;
        }
    } else {                                  // execute load normally
        if (fill_form) {                      // fill NaT on ld8.fill
            bit_pos = GR[r3]{8:3};
            GR[r1] = val;
            GR[r1].nat = AR[UNAT]{bit_pos};
        } else {                              // clear NaT on other types
            if (size == 16) {
                GR[r1] = val;
                AR[CSD] = val_ar;
            }
            else {
                GR[r1] = zero_ext(val, size * 8);
            }
            GR[r1].nat = 0;
        }
        if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                              // add entry to ALAT
            alat_write(ldtype, GENERAL, r1, paddr, size);
    }
}

if (imm_base_update_form) {                   // update base register
    GR[r3] = GR[r3] + sign_ext(imm9, 9);
    GR[r3].nat = GR[r3].nat;
} else if (reg_base_update_form) {
    GR[r3] = GR[r3] + tmp_r2;
    GR[r3].nat = GR[r3].nat || tmp_r2nat;
}

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

Interruptions:

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

ldf — Floating-point Load

Format:

(*qp*) ldf*fsz*.*fldtype*.*ldhint* $f_1 = [r_3]$	no_base_update_form	M9
(*qp*) ldf*fsz*.*fldtype*.*ldhint* $f_1 = [r_3], r_2$	reg_base_update_form	M7
(*qp*) ldf*fsz*.*fldtype*.*ldhint* $f_1 = [r_3], imm_9$	imm_base_update_form	M8
(*qp*) ldf8.*fldtype*.*ldhint* $f_1 = [r_3]$	integer_form, no_base_update_form	M9
(*qp*) ldf8.*fldtype*.*ldhint* $f_1 = [r_3], r_2$	integer_form, reg_base_update_form	M7
(*qp*) ldf8.*fldtype*.*ldhint* $f_1 = [r_3], imm_9$	integer_form, imm_base_update_form	M8
(*qp*) ldf.fill.*ldhint* $f_1 = [r_3]$	fill_form, no_base_update_form	M9
(*qp*) ldf.fill.*ldhint* $f_1 = [r_3], r_2$	fill_form, reg_base_update_form	M7
(*qp*) ldf.fill.*ldhint* $f_1 = [r_3], imm_9$	fill_form, imm_base_update_form	M8

Description:

A value consisting of *fsz* bytes is read from memory starting at the address specified by the value in GR r_3. The value is then converted into the floating-point register format and placed in FR f_1. See Section 5.1, "Data Types and Formats" on page 1:85 for details on conversion to floating-point register format. The values of the *fsz* completer are given in Table 2-35. The *fldtype* completer specifies special load operations, which are described in Table 2-36.

For the integer_form, an 8-byte value is loaded and placed in the significand field of FR f_1 without conversion. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For the fill_form, a 16-byte value is loaded, and the appropriate fields are placed in FR f_1 without conversion. This instruction is used for reloading a spilled register. See Section 4.4.4, "Control Speculation" on page 1:60 for details.

In the base update forms, the value in GR r_3 is added to either a signed immediate value (imm_9) or a value from GR r_2, and the result is placed back in GR r_3. This base register update is done after the load, and does not affect the load address. In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set and no fault is raised.

Table 2-35. *fsz* Completers

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

Table 2-36. FP Load Types

fldtype Completer	Interpretation	Special Load Operation
none	Normal load	
s	Speculative load	Certain exceptions may be deferred rather than generating a fault. Deferral causes NaTVal to be placed in the target register. The NaTVal value is later used to detect deferral.
a	Advanced load	An entry is added to the ALAT. This allows later instructions to check for colliding stores. If the referenced data page has a non-speculative attribute, no ALAT entry is added to the ALAT and the target register is set as follows: for the integer_form, the exponent is set to 0x1003E and the sign and significand are set to zero; for all other forms, the sign, exponent and significand are set to zero. The absence of an ALAT entry is later used to detect deferral or collision.

Table 2-36. FP Load Types (Continued)

fldtype Completer	Interpretation	Special Load Operation
sa	Speculative Advanced load	An entry is added to the ALAT, and certain exceptions may be deferred. Deferral causes NaTVal to be placed in the target register, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.
c.nc	Check load – no clear	The ALAT is searched for a matching entry. If found, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a load is performed, and an entry is added to the ALAT (unless the referenced data page has a non-speculative attribute, in which case no ALAT entry is allocated).
c.clr	Check load – clear	The ALAT is searched for a matching entry. If found, the entry is removed, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a clear check load behaves like a normal load.

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

For the non-speculative load types, if NaT bit associated with GR r_3 is 1, a Register NaT Consumption fault is taken. For speculative and speculative advanced loads, no fault is raised, and the exception is deferred. For the base-update calculation, if the NaT bit associated with GR r_2 is 1, the NaT bit associated with GR r_3 is set to 1 and no fault is raised.

The value of the *ldhint* modifier specifies the locality of the memory access. The mnemonic values of *ldhint* are given in Table 2-34 on page 3:152. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *ldhint*. Prefetch and locality hints do not affect program functionality and may be ignored by the implementation. See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69 for details.

In the no_base_update form, the value in GR r_3 is not modified and no prefetch hint is implied.

The PSR.mfl and PSR.mfh bits are updated to reflect the modification of FR f_1.

Hardware support for `ldfe` (10-byte) instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `ldfe` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted. The fault is delivered only on the normal, advanced, and check load flavors. Control-speculative flavors of `ldfe` always defer the Unsupported Data Reference fault.

Operation:

```
if (PR[qp]) {
    size = (fill_form ? 16 : (integer_form ? 8 : fsz));
    speculative = (fldtype == 's' || fldtype == 'sa');
    advanced = (fldtype == 'a' || fldtype == 'sa');
    check_clear = (fldtype == 'c.clr' );
    check_no_clear = (fldtype == 'c.nc');
    check = check_clear || check_no_clear;
    translate_address = 1;
    read_memory = 1;

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

    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, itype);

    if (!speculative && GR[r3].nat)              // fault on NaT address
        register_nat_consumption_fault(itype);

    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(FLOAT, f1)) {
        translate_address = alat_translate_address_on_hit(fldtype, FLOAT, f1);
        read_memory = alat_read_memory_on_hit(fldtype, FLOAT, f1);
    }

    if (!translate_address) {
        if (check_clear || advanced)          // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                  &defer);
            spontaneous_deferral(paddr, size, UM.be, mattr, UNORDERED,
                                  ldhint, &defer);
            if (!defer && read_memory)
                val = mem_read(paddr, size, UM.be, mattr, UNORDERED, ldhint);
        }
        if (check_clear || advanced)              // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
        if (speculative && defer) {
            FR[f1] = NATVAL;
        } else if (advanced && !speculative && defer) {
            FR[f1] = (integer_form ? FP_INT_ZERO : FP_ZERO);
        } else {                                  // execute load normally
            FR[f1] = fp_mem_to_fr_format(val, size, integer_form);

            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                  // add entry to ALAT
                alat_write(fldtype, FLOAT, f1, paddr, size);
        }
```

```
    }

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

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

    fp_update_psr(f1);
}
```

Interruptions:

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

ldfp — Floating-point Load Pair

Format:

(*qp*) ldfps.*fldtype*.*ldhint* f_1, f_2 = [r_3]	single_form, no_base_update_form	M11
(*qp*) ldfps.*fldtype*.*ldhint* f_1, f_2 = [r_3], 8	single_form, base_update_form	M12
(*qp*) ldfpd.*fldtype*.*ldhint* f_1, f_2 = [r_3]	double_form, no_base_update_form	M11
(*qp*) ldfpd.*fldtype*.*ldhint* f_1, f_2 = [r_3], 16	double_form, base_update_form	M12
(*qp*) ldfp8.*fldtype*.*ldhint* f_1, f_2 = [r_3]	integer_form, no_base_update_form	M11
(*qp*) ldfp8.*fldtype*.*ldhint* f_1, f_2 = [r_3], 16	integer_form, base_update_form	M12

Description:

Eight (single_form) or sixteen (double_form/integer_form) bytes are read from memory starting at the address specified by the value in GR r_3. The value read is treated as a contiguous pair of floating-point numbers for the single_form/double_form and as integer/Parallel FP data for the integer_form. Each number is converted into the floating-point register format. The value at the lowest address is placed in FR f_1, and the value at the highest address is placed in FR f_2. See Section 5.1, "Data Types and Formats" on page 1:85 for details on conversion to floating-point register format. The *fldtype* completer specifies special load operations, which are described in Table 2-36 on page 3:157.

For more details on speculative, advanced and check loads see Section 4.4.4, "Control Speculation" on page 1:60 and Section 4.4.5, "Data Speculation" on page 1:63.

For the non-speculative load types, if NaT bit associated with GR r_3 is 1, a Register NaT Consumption fault is taken. For speculative and speculative advanced loads, no fault is raised, and the exception is deferred.

In the base_update_form, the value in GR r_3 is added to an implied immediate value (equal to double the data size) and the result is placed back in GR r_3. This base register update is done after the load, and does not affect the load address.

The value of the *ldhint* modifier specifies the locality of the memory access. The mnemonic values of *ldhint* are given in Table 2-34 on page 3:152. A prefetch hint is implied in the base update form. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *ldhint*. Prefetch and locality hints do not affect program functionality and may be ignored by the implementation. See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69 for details.

In the no_base_update form, the value in GR r_3 is not modified and no prefetch hint is implied.

The PSR.mfl and PSR.mfh bits are updated to reflect the modification of FR f_1 and FR f_2.

There is a restriction on the choice of target registers. Register specifiers f_1 and f_2 must specify one odd-numbered physical FR and one even-numbered physical FR. Specifying two odd or two even registers will cause an Illegal Operation fault to be raised. The restriction is on physical register numbers after register rotation. This means that if f_1 and f_2 both specify static registers or both specify rotating registers, then f_1 and f_2 must be odd/even or even/odd. If f_1 and f_2 specify one static and one rotating register, the restriction depends on CFM.rrb.fr. If CFM.rrb.fr is even, the restriction is the same; f_1 and f_2 must be odd/even or even/odd. If CFM.rrb.fr is odd, then f_1 and f_2 must be even/even or odd/odd. Specifying one static and one rotating register should only be done when CFM.rrb.fr will have a predictable value (such as 0).

Operation:

```
if (PR[qp]) {
    size = single_form ? 8 : 16;

    speculative = (fldtype == 's' || fldtype == 'sa');
    advanced = (fldtype == 'a' || fldtype == 'sa');
    check_clear = (fldtype == 'c.clr');
    check_no_clear = (fldtype == 'c.nc');
    check = check_clear || check_no_clear;
    translate_address = 1;
    read_memory = 1;

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

    if (fp_reg_bank_conflict(f1, f2))
        illegal_operation_fault();

    if (base_update_form)
        check_target_register(r3);

    fp_check_target_register(f1);
    fp_check_target_register(f2);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, itype);

    if (!speculative && GR[r3].nat)              // fault on NaT address
        register_nat_consumption_fault(itype);

    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(FLOAT, f1)) {
        translate_address = alat_translate_address_on_hit(fldtype, FLOAT, f1);
        read_memory = alat_read_memory_on_hit(fldtype, FLOAT, f1);
    }

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

```
            FR[f2] = (integer_form ? FP_INT_ZERO : FP_ZERO);
        } else {                                    // execute load normally
            FR[f1] = fp_mem_to_fr_format(f1_val, size/2, integer_form);
            FR[f2] = fp_mem_to_fr_format(f2_val, size/2, integer_form);

            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                    // add entry to ALAT
                alat_write(fldtype, FLOAT, f1, paddr, size);
        }
    }

    if (base_update_form) {                         // update base register
        GR[r3] = GR[r3] + size;
        GR[r3].nat = GR[r3].nat;
        if (!GR[r3].nat)
            mem_implicit_prefetch(GR[r3], ldhint, itype);
    }

    fp_update_psr(f1);
    fp_update_psr(f2);
}
```

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

lfetch — Line Prefetch

Format:

(*qp*) lfetch.*lftype*.*lfhint* [r_3]	no_base_update_form	M18
(*qp*) lfetch.*lftype*.*lfhint* [r_3], r_2	reg_base_update_form	M20
(*qp*) lfetch.*lftype*.*lfhint* [r_3], imm_9	imm_base_update_form	M22
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3]	no_base_update_form, exclusive_form	M18
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], r_2	reg_base_update_form, exclusive_form	M20
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], imm_9	imm_base_update_form, exclusive_form	M22

Description:

The line containing the address specified by the value in GR r_3 is moved to the highest level of the data memory hierarchy. The value of the *lfhint* modifier specifies the locality of the memory access; see Section 4.4, "Memory Access Instructions" on page 1:57 for details. The mnemonic values of *lfhint* are given in Table 2-38.

The behavior of the memory read is also determined by the memory attribute associated with the accessed page. See Chapter 4, "Addressing and Protection" in Volume 2. Line size is implementation dependent but must be a power of two greater than or equal to 32 bytes. In the exclusive form, the cache line is allowed to be marked in an exclusive state. This qualifier is used when the program expects soon to modify a location in that line. If the memory attribute for the page containing the line is not cacheable, then no reference is made.

The completer, *lftype*, specifies whether or not the instruction raises faults normally associated with a regular load. Table 2-37 defines these two options.

Table 2-37. *lftype* Mnemonic Values

lftype Mnemonic	Interpretation
none	No faults are raised
fault	Raise faults

In the base update forms, after being used to address memory, the value in GR r_3 is incremented by either the sign-extended value in imm_9 (in the imm_base_update_form) or the value in GR r_2 (in the reg_base_update_form). In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set – no fault is raised.

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

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

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

`lfetch` instructions, like hardware prefetches, are not orderable operations, i.e., they have no order with respect to prior or subsequent memory operations.

Table 2-38. *lfhint* Mnemonic Values

lfhint Mnemonic	Interpretation
none	Temporal locality, level 1
nt1	No temporal locality, level 1
nt2	No temporal locality, level 2
nta	No temporal locality, all levels

A faulting `lfetch` to an unimplemented address results in an Unimplemented Data Address fault. A non-faulting `lfetch` to an unimplemented address does not take the fault and will not issue a prefetch request, but, if specified, will perform a register post-increment.

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

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

Operation:

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

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

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

    excl_hint = (exclusive_form) ? EXCLUSIVE : 0;

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

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

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

Interruptions:

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

loadrs — Load Register Stack

Format: loadrs M25

Description: This instruction ensures that a specified number of bytes (registers values and/or NaT collections) below the current BSP have been loaded from the backing store into the stacked general registers. The loaded registers are placed into the dirty partition of the register stack. All other stacked general registers are marked as invalid, without being saved to the backing store.

The number of bytes to be loaded is specified in a sub-field of the RSC application register (RSC.loadrs). Backing store addresses are always 8-byte aligned, and therefore the low order 3 bits of the `loadrs` field (RSC.loadrs{2:0}) are ignored. This instruction can be used to invalidate all stacked registers outside the current frame, by setting RSC.loadrs to zero.

This instruction will fault with an Illegal Operation fault under any of the following conditions:

- the RSE is not in enforced lazy mode (RSC.mode is non-zero).
- CFM.sof and RSC.loadrs are both non-zero.
- an attempt is made to load up more registers than are available in the physical stacked register file.

This instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0; otherwise, the results are undefined. This instruction cannot be predicated.

Operation:

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

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

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

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	ordering_form	M24
(*qp*) mf.a	acceptance_form	M24

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

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

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

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

Operation:

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

Interruptions: None

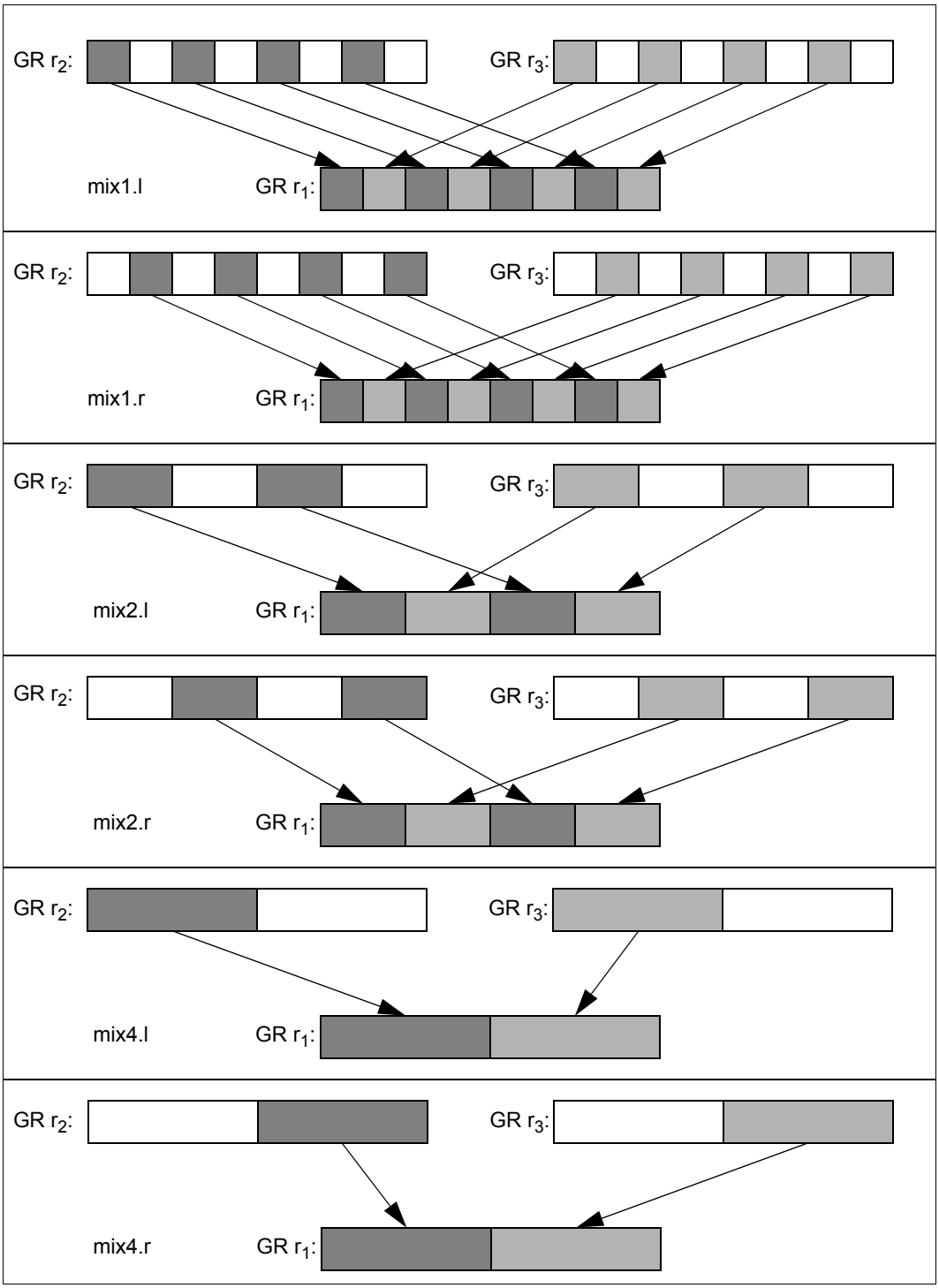
mix — Mix

Format:

(*qp*) mix1.l $r_1 = r_2, r_3$	one_byte_form, left_form	I2
(*qp*) mix2.l $r_1 = r_2, r_3$	two_byte_form, left_form	I2
(*qp*) mix4.l $r_1 = r_2, r_3$	four_byte_form, left_form	I2
(*qp*) mix1.r $r_1 = r_2, r_3$	one_byte_form, right_form	I2
(*qp*) mix2.r $r_1 = r_2, r_3$	two_byte_form, right_form	I2
(*qp*) mix4.r $r_1 = r_2, r_3$	four_byte_form, right_form	I2

Description: The data elements of GR r_2 and r_3 are mixed as shown in Figure 2-25, and the result placed in GR r_1. The data elements in the source registers are grouped in pairs, and one element from each pair is selected for the result. In the left_form, the result is formed from the leftmost elements from each of the pairs. In the right_form, the result is formed from the rightmost elements. Elements are selected alternately from the two source registers.

Figure 2-25. Mix Examples



Operation:

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

    if (one_byte_form) {                              // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (left_form)
            GR[r1] = concatenate8(x[7], y[7], x[5], y[5],
                                  x[3], y[3], x[1], y[1]);
        else // right_form
            GR[r1] = concatenate8(x[6], y[6], x[4], y[4],
                                  x[2], y[2], x[0], y[0]);

    } else if (two_byte_form) {                       // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (left_form)
            GR[r1] = concatenate4(x[3], y[3], x[1], y[1]);
        else // right_form
            GR[r1] = concatenate4(x[2], y[2], x[0], y[0]);

    } else {                                          // four-byte elements
        x[0] = GR[r2]{31:0};     y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};    y[1] = GR[r3]{63:32};

        if (left_form)
            GR[r1] = concatenate2(x[1], y[1]);
        else // right_form
            GR[r1] = concatenate2(x[0], y[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

mov — Move Application Register

Format:

(*qp*) mov $r_1 = ar_3$	pseudo-op	
(*qp*) mov $ar_3 = r_2$	pseudo-op	
(*qp*) mov $ar_3 = imm_8$	pseudo-op	
(*qp*) mov.i $r_1 = ar_3$	i_form, from_form	I28
(*qp*) mov.i $ar_3 = r_2$	i_form, register_form, to_form	I26
(*qp*) mov.i $ar_3 = imm_8$	i_form, immediate_form, to_form	I27
(*qp*) mov.m $r_1 = ar_3$	m_form, from_form	M31
(*qp*) mov.m $ar_3 = r_2$	m_form, register_form, to_form	M29
(*qp*) mov.m $ar_3 = imm_8$	m_form, immediate_form, to_form	M30

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

In the from_form, the application register specified by ar_3 is copied into GR r_1 and the corresponding NaT bit is cleared.

In the to_form, the value in GR r_2 (in the register_form), or the sign-extended value in imm_8 (in the immediate_form), is placed in AR ar_3. In the register_form if the NaT bit corresponding to GR r_2 is set, then a Register NaT Consumption fault is raised.

Only a subset of the application registers can be accessed by each execution unit (M or I). Table 3-3 on page 1:28 indicates which application registers may be accessed from which execution unit type. An access to an application register from the wrong unit type causes an Illegal Operation fault.

This instruction has multiple forms with the pseudo operation eliminating the need for specifying the execution unit. Accesses of the ARs are always implicitly serialized. While implicitly serialized, read-after-write and write-after-write dependency violations must be avoided (e.g., setting CCV, followed by `cmpxchg` in the same instruction group, or simultaneous writes to the UNAT register by `ld.fill` and mov to UNAT).

Operation:

```
if (PR[qp]) {
    tmp_type = (i_form ? AR_I_TYPE : AR_M_TYPE);
    if (is_reserved_reg(tmp_type, ar3))
        illegal_operation_fault();

    if (from_form) {
        check_target_register(r1);
        if (((ar3 == BSPSTORE) || (ar3 == RNAT)) && (AR[RSC].mode != 0))
            illegal_operation_fault();

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

        if ((ar3 == ITC || ar3 == RUC) && PSR.si && PSR.vm == 1)
            virtualization_fault();

        GR[r1] = (is_ignored_reg(ar3)) ? 0 : AR[ar3];
        GR[r1].nat = 0;
    } else {                                        // to_form
        tmp_val = (register_form) ? GR[r2] : sign_ext(imm8, 8);

        if (is_read_only_reg(AR_TYPE, ar3) ||
            (((ar3 == BSPSTORE) || (ar3 == RNAT)) && (AR[RSC].mode != 0)))
            illegal_operation_fault();

        if (register_form && GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(AR_TYPE, ar3, tmp_val))
            reserved_register_field_fault();

        if ((is_kernel_reg(ar3) || ar3 == ITC || ar3 == RUC) && (PSR.cpl != 0))
            privileged_register_fault();

        if ((ar3 == ITC || ar3 == RUC) && PSR.vm == 1)
            virtualization_fault();

        if (!is_ignored_reg(ar3)) {
            tmp_val = ignored_field_mask(AR_TYPE, ar3, tmp_val);
            // check for illegal promotion
            if (ar3 == RSC && tmp_val{3:2} u< PSR.cpl)
                tmp_val{3:2} = PSR.cpl;
            AR[ar3] = tmp_val;

            if (ar3 == BSPSTORE) {
                AR[BSP] = rse_update_internal_stack_pointers(tmp_val);
                AR[RNAT] = undefined();
            }
        }
    }
}
```

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

mov — Move Branch Register

Format:

(*qp*) mov $r_1 = b_2$	from_form	I22
(*qp*) mov $b_1 = r_2$	pseudo-op	
(*qp*) mov.*mwh.ih* $b_1 = r_2, tag_{13}$	to_form	I21
(*qp*) mov.ret.*mwh.ih* $b_1 = r_2, tag_{13}$	return_form, to_form	I21

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

In the from_form, the branch register specified by b_2 is copied into GR r_1. The NaT bit corresponding to GR r_1 is cleared.

In the to_form, the value in GR r_2 is copied into BR b_1. If the NaT bit corresponding to GR r_2 is 1, then a Register NaT Consumption fault is taken.

A set of hints can also be provided when moving to a branch register. These hints are very similar to those provided on the `brp` instruction, and provide prediction information about a future branch which may use the value being moved into BR b_1. The return_form is used to provide the hint that this value will be used in a return-type branch.

The values for the *mwh* whether hint completer are given in Table 2-39. 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

mwh Completer	Move to BR Whether Hint
none	Ignore all hints
sptk	Static Taken
dptk	Dynamic

A pseudo-op is provided for copying a general register into a branch register when there is no hint information to be specified. This is encoded with a value of 0 for tag_{13} and values corresponding to *none* for the hint completers.

Operation:

```
if (PR[qp]) {
    if (from_form) {
        check_target_register(r1);
        GR[r1] = BR[b2];
        GR[r1].nat = 0;
    } else { // to_form
        tmp_tag = IP + sign_ext((timm9 << 4), 13);
        if (GR[r2].nat)
            register_nat_consumption_fault(0);
        BR[b1] = GR[r2];
        branch_predict(mwh, ih, return_form, GR[r2], tmp_tag);
    }
}
```

Interruptions: Illegal Operation fault Register NaT Consumption fault

mov — Move Control Register

Format:

(*qp*) mov $r_1 = cr_3$	from_form	M33
(*qp*) mov $cr_3 = r_2$	to_form	M32

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

For the from_form, the control register specified by cr_3 is read and the value copied into GR r_1.

For the to_form, GR r_2 is read and the value copied into CR cr_3.

Control registers can only be accessed at the most privileged level, and when PSR.vm is 0. Reading or writing an interruption control register (CR16-CR27), when the PSR.ic bit is one, will result in an Illegal Operation fault.

Operation:

```
if (PR[qp]) {
    if (is_reserved_reg(CR_TYPE, cr3)
        || to_form && is_read_only_reg(CR_TYPE, cr3)
        || PSR.ic && is_interruption_cr(cr3))
    {
        illegal_operation_fault();
    }

    if (from_form)
        check_target_register(r1);
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (from_form) {
        if (PSR.vm == 1)
            virtualization_fault();
        if (cr3 == IVR)
            check_interrupt_request();

        if (cr3 == ITIR)
            GR[r1] = impl_itir_cwi_mask(CR[ITIR]);
        else
            GR[r1] = CR[cr3];

        GR[r1].nat = 0;
    } else {            // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(CR_TYPE, cr3, GR[r2]))
            reserved_register_field_fault();
        if ((cr3 == IFA) && impl_check_mov_ifa() &&
            unimplemented_virtual_address(GR[r2], PSR.vm))
            unimplemented_data_address_fault(0);
        if (PSR.vm == 1)
            virtualization_fault();
        if (cr3 == EOI)
            end_of_interrupt();

        tmp_val = ignored_field_mask(CR_TYPE, cr3, GR[r2]);
        CR[cr3] = tmp_val;
        if (cr3 == IIPA)
```

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

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.

Operation: See "fmerge — Floating-point Merge" on page 3:80.

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.

Operation: See "add — Add" on page 3:14.

mov — Move Immediate

Format: (*qp*) mov $r_1 = imm_{22}$ pseudo-op of: (*qp*) addl $r_1 = imm_{22}$, r0

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

Operation: See "add — Add" on page 3:14.

mov — Move Indirect Register

Format:

(*qp*) mov r_1 = *ireg*[r_3]	from_form	M43
(*qp*) mov *ireg*[r_3] = r_2	to_form	M42

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

For move from indirect register, GR r_3 is read and the value used as an index into the register file specified by *ireg* (see Table 2-40 below). The indexed register is read and its value is copied into GR r_1.

For move to indirect register, GR r_3 is read and the value used as an index into the register file specified by *ireg*. GR r_2 is read and its value copied into the indexed register.

Table 2-40. Indirect Register File Mnemonics

ireg	Register File
cpuid	Processor Identification Register
dbr	Data Breakpoint Register
ibr	Instruction Breakpoint Register
pkr	Protection Key Register
pmc	Performance Monitor Configuration Register
pmd	Performance Monitor Data Register
rr	Region Register

For all register files other than the region registers, bits {7:0} of GR r_3 are used as the index. For region registers, bits {63:61} are used. The remainder of the bits are ignored.

Instruction and data breakpoint, performance monitor configuration, protection key, and region registers can only be accessed at the most privileged level. Performance monitor data registers can only be written at the most privileged level.

The CPU identification registers can only be read. There is no to_form of this instruction.

For move to protection key register, the processor ensures uniqueness of protection keys by checking new valid protection keys against all protection key registers. If any matching keys are found, duplicate protection keys are invalidated.

Apart from the PMC and PMD register files, access of a non-existent register results in a Reserved Register/Field fault. All accesses to the implementation-dependent portion of PMC and PMD register files result in implementation dependent behavior but do not fault.

Modifying a region register or a protection key register which is being used to translate:

- the executing instruction stream when PSR.it == 1, or
- the data space for an eager RSE reference when PSR.rt == 1

is an undefined operation.

Operation:

```
if (PR[qp]) {
    if (ireg == RR_TYPE)
        tmp_index = GR[r3]{63:61};
    else // all other register types
        tmp_index = GR[r3]{7:0};
```

```
if (from_form) {
    check_target_register(r1);

    if (PSR.cpl != 0 && !(ireg == PMD_TYPE || ireg == CPUID_TYPE))
        privileged_operation_fault(0);

    if (GR[r3].nat)
        register_nat_consumption_fault(0);

    if (is_reserved_reg(ireg, tmp_index))
        reserved_register_field_fault();

    if (PSR.vm == 1 && ireg != PMD_TYPE)
        virtualization_fault();

    if (ireg == PMD_TYPE) {
        if ((PSR.cpl != 0) && ((PSR.sp == 1) ||
             (tmp_index > 3 &&
             tmp_index <= IMPL_MAXGENERIC_PMCPMD &&
             PMC[tmp_index].pm == 1)))
            GR[r1] = 0;
        else
            GR[r1] = pmd_read(tmp_index);
    } else
        switch (ireg) {
            case CPUID_TYPE: GR[r1] = CPUID[tmp_index]; break;
            case DBR_TYPE:   GR[r1] = DBR[tmp_index]; break;
            case IBR_TYPE:   GR[r1] = IBR[tmp_index]; break;
            case PKR_TYPE:   GR[r1] = PKR[tmp_index]; break;
            case PMC_TYPE:   GR[r1] = pmc_read(tmp_index); break;
            case RR_TYPE:    GR[r1] = RR[tmp_index]; break;
        }
    GR[r1].nat = 0;
} else {          // to_form
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (GR[r2].nat || GR[r3].nat)
        register_nat_consumption_fault(0);

    if (is_reserved_reg(ireg, tmp_index)
        || ireg == CPUID_TYPE
        || is_reserved_field(ireg, tmp_index, GR[r2]))
        reserved_register_field_fault();
    if (PSR.vm == 1)
        virtualization_fault();
    if (ireg == PKR_TYPE && GR[r2]{0} == 1) { // writing valid prot key
        if ((tmp_slot = tlb_search_pkr(GR[r2]{31:8})) != NOT_FOUND)
            PKR[tmp_slot].v = 0; // clear valid bit of matching key reg
    }
    tmp_val = ignored_field_mask(ireg, tmp_index, GR[r2]);
    switch (ireg) {
        case DBR_TYPE:   DBR[tmp_index] = tmp_val; break;
        case IBR_TYPE:   IBR[tmp_index] = tmp_val; break;
        case PKR_TYPE:   PKR[tmp_index] = tmp_val; break;
        case PMC_TYPE:   pmc_write(tmp_index, tmp_val); break;
```

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

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

Serialization: For move to data breakpoint registers, software must issue a data serialize operation before issuing a memory reference dependent on the modified register.

For move to instruction breakpoint registers, software must issue an instruction serialize operation before fetching an instruction dependent on the modified register.

For move to protection key, region, performance monitor configuration, and performance monitor data registers, software must issue an instruction or data serialize operation to ensure the changes are observed before issuing any dependent instruction.

To obtain improved accuracy, software can issue an instruction or data serialize operation before reading the performance monitors.

mov — Move Instruction Pointer

Format: (*qp*) mov r_1 = ip I25

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

Operation:

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

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

Interruptions: Illegal Operation fault

mov — Move Predicates

Format:

(*qp*) mov r_1 = pr	from_form	I25
(*qp*) mov pr = r_2, $mask_{17}$	to_form	I23
(*qp*) mov pr.rot = imm_{44}	to_rotate_form	I24

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

For moving the predicates to a GR, PR i is copied to bit position i within GR r_1.

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

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

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

Operation:

```
if (PR[qp]) {
    if (from_form) {
        check_target_register(r1);
        GR[r1] = 1;                              // PR[0] is always 1
        for (i = 1; i <= 63; i++) {
            GR[r1]{i} = PR[pr_phys_to_virt(i)];
        }
        GR[r1].nat = 0;
    } else if (to_form) {
        if (GR[r2].nat)
            register_nat_consumption_fault(0);
        tmp_src = sign_ext(mask17, 17);
        for (i = 1; i <= 63; i++) {
            if (tmp_src{i})
                PR[pr_phys_to_virt(i)] = GR[r2]{i};
        }
    } else {   // to_rotate_form
        tmp_src = sign_ext(imm44, 44);
        for (i = 16; i <= 63; i++) {
            PR[pr_phys_to_virt(i)] = tmp_src{i};
        }
    }
}
```

Interruptions: Illegal Operation fault Register NaT Consumption fault

mov — Move Processor Status Register

Format:

(*qp*) mov r_1 = psr	from_form	M36
(*qp*) mov psr.l = r_2	to_form	M35

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

For move from processor status register, PSR bits {36:35} and {31:0} are read, and copied into GR r_1. All other bits of the PSR read as zero.

For move to processor status register, GR r_2 is read, bits {31:0} copied into PSR{31:0} and bits {63:32} are ignored. Bits {31:0} of GR r_2 corresponding to reserved fields of the PSR must be 0 or a Reserved Register/Field fault will result. An implementation may also raise Reserved Register/Field fault if bits {63:32} in GR r_2 corresponding to reserved fields of the PSR are non-zero.

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

The contents of the interruption resources (that are overwritten when the PSR.ic bit is 1) are undefined if an interruption occurs between the enabling of the PSR.ic bit and a subsequent instruction serialize operation.

Operation:

```
if (PR[qp]) {
    if (from_form)
        check_target_register(r1);
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (from_form) {
        if (PSR.vm == 1)
            virtualization_fault();
        tmp_val = zero_ext(PSR{31:0}, 32);     // read lower 32 bits
        tmp_val |= PSR{36:35} << 35;           // read mc and it bits
        GR[r1] = tmp_val;                      // other bits read as zero
        GR[r1].nat = 0;
    } else {          // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(PSR_TYPE, PSR_MOVPART, GR[r2]))
            reserved_register_field_fault();

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

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

Interruptions:

Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Virtualization fault

Serialization: Software must issue an instruction or data serialize operation before issuing instructions dependent upon the altered PSR bits. Unlike with the `rsm` instruction, the PSR.i bit is not treated specially when cleared.

mov — Move User Mask

Format:

(*qp*) mov r_1 = psr.um	from_form	M36
(*qp*) mov psr.um = r_2	to_form	M35

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

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

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

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

Operation:

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

        GR[r1] = zero_ext(PSR{5:0}, 6);
        GR[r1].nat = 0;
    } else {                                          // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(PSR_TYPE, PSR_UM, GR[r2]))
            reserved_register_field_fault();

        PSR{1:0} = GR[r2]{1:0};

        if (PSR.sp == 0)            // unsecured perf monitor
            PSR{2} = GR[r2]{2};

        PSR{5:3} = GR[r2]{5:3};
    }
}
```

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

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

movl — Move Long Immediate

Format: (*qp*) movl $r_1 = imm_{64}$ X2

Description: The immediate value imm_{64} is copied to GR r_1. The L slot of the bundle contains 41 bits of imm_{64}.

Operation:

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

    GR[r1] = imm64;
    GR[r1].nat = 0;
}
```

Interruptions: Illegal Operation fault

mpy4 — Unsigned Integer Multiply

Format: (*qp*) mpy4 $r_1 = r_2, r_3$ I2

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

Operation:

```
if (PR[qp]) {
    if (!instruction_implemented(mpy4))
        illegal_operation_fault();
    check_target_register(r1);

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

Interruptions: Illegal Operation fault

mpyshl4 — Unsigned Integer Shift Left and Multiply

Format: (*qp*) mpyshl4 $r_1 = r_2, r_3$ I2

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

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

```
mpy4       r1 = ra, rb;;  //partial product low 32 bits * low 32 bits
mpyshl4    r2 = ra, rb;;  //partial product high 32 bits * low 32 bits
mpyshl4    r3 = rb, ra    //partial product low 32 bits * high 32 bits
add        r1 = r1, r2;;  //partial sum
add        rc = r1, r3    //final sum
```

Operation:

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

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

Interruptions: Illegal Operation fault

mux — Mux

Format:

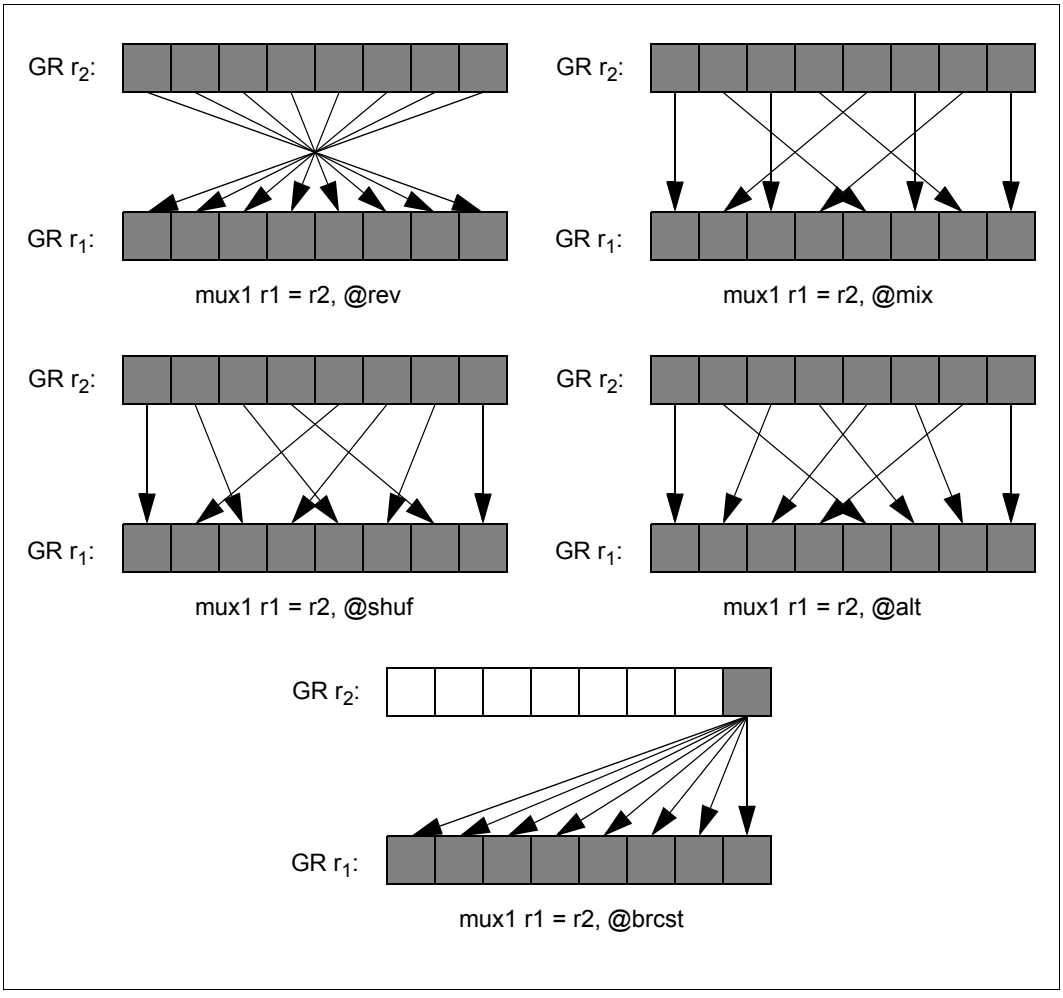
(*qp*) mux1 $r_1 = r_2$, $mbtype_4$	one_byte_form	I3
(*qp*) mux2 $r_1 = r_2$, $mhtype_8$	two_byte_form	I4

Description: A permutation is performed on the packed elements in a single source register, GR r_2, and the result is placed in GR r_1. For 8-bit elements, only some of all possible permutations can be specified. The five possible permutations are given in Table 2-41 and shown in Figure 2-26.

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

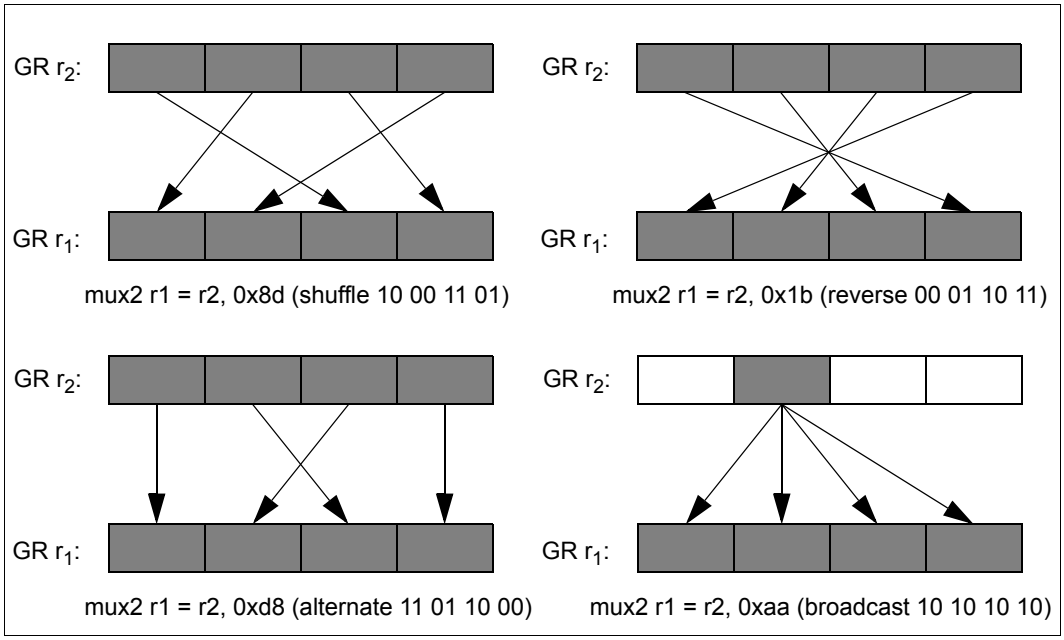
$mbtype_4$	Function
@rev	Reverse the order of the bytes
@mix	Perform a Mix operation on the two halves of GR r_2
@shuf	Perform a Shuffle operation on the two halves of GR r_2
@alt	Perform an Alternate operation on the two halves of GR r_2
@brcst	Perform a Broadcast operation on the least significand byte of GR r_2

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



For 16-bit elements, all possible permutations, with and without repetitions can be specified. They are expressed with an 8-bit *mhtype*$_8$ field, which encodes the indices of the four 16-bit data elements. The indexed 16-bit elements of GR r_2 are copied to corresponding 16-bit positions in the target register GR r_1. The indices are encoded in little-endian order. (The 8 bits of *mhtype*$_8$[7:0] are grouped in pairs of bits and named *mhtype*$_8$[3], *mhtype*$_8$[2], *mhtype*$_8$[1], *mhtype*$_8$[0] in the Operation section).

Figure 2-27. Mux2 Examples (16-bit elements)



Operation:

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

    if (one_byte_form) {
        x[0] = GR[r2]{7:0};
        x[1] = GR[r2]{15:8};
        x[2] = GR[r2]{23:16};
        x[3] = GR[r2]{31:24};
        x[4] = GR[r2]{39:32};
        x[5] = GR[r2]{47:40};
        x[6] = GR[r2]{55:48};
        x[7] = GR[r2]{63:56};

        switch (mbtype) {
            case '@rev':
                GR[r1] = concatenate8(x[0], x[1], x[2], x[3],
                                      x[4], x[5], x[6], x[7]);
                break;

            case '@mix':
                GR[r1] = concatenate8(x[7], x[3], x[5], x[1],
                                      x[6], x[2], x[4], x[0]);
                break;

            case '@shuf':
                GR[r1] = concatenate8(x[7], x[3], x[6], x[2],
                                      x[5], x[1], x[4], x[0]);
                break;

            case '@alt':
                GR[r1] = concatenate8(x[7], x[5], x[3], x[1],
                                      x[6], x[4], x[2], x[0]);
                break;

            case '@brcst':
                GR[r1] = concatenate8(x[0], x[0], x[0], x[0],
                                      x[0], x[0], x[0], x[0]);
                break;
        }
    } else {                                          // two_byte_form
        x[0] = GR[r2]{15:0};
        x[1] = GR[r2]{31:16};
        x[2] = GR[r2]{47:32};
        x[3] = GR[r2]{63:48};

        res[0] = x[mhtype8{1:0}];
        res[1] = x[mhtype8{3:2}];
        res[2] = x[mhtype8{5:4}];
        res[3] = x[mhtype8{7:6}];

        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat;
}
```

Interruptions: Illegal Operation fault

nop — No Operation

Format:

(*qp*) nop *imm*$_{21}$	pseudo-op	
(*qp*) nop.i *imm*$_{21}$	i_unit_form	I18
(*qp*) nop.b *imm*$_{21}$	b_unit_form	B9
(*qp*) nop.m *imm*$_{21}$	m_unit_form	M48
(*qp*) nop.f *imm*$_{21}$	f_unit_form	F16
(*qp*) nop.x *imm*$_{62}$	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:

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

Interruptions: None

or — Logical Or

Format:

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

Description: The two source operands are logically ORed and the result placed in GR r_1. In the register form the first operand is GR r_2; in the immediate form the first operand is taken from the imm_8 encoding field.

Operation:

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

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

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

Interruptions: Illegal Operation fault

pack — Pack

Format:

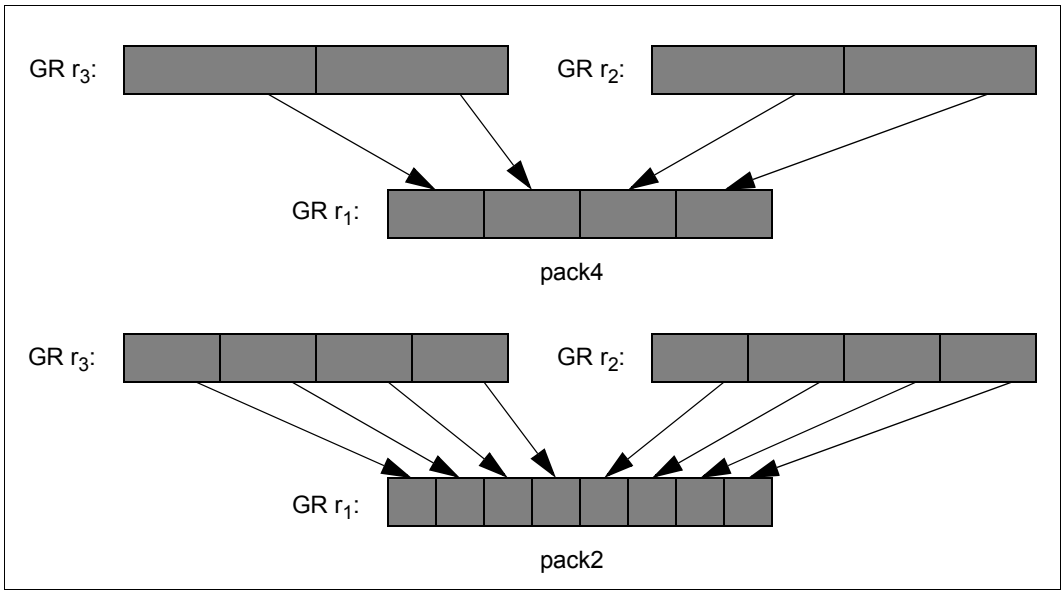
(*qp*) pack2.sss $r_1 = r_2, r_3$	two_byte_form, signed_saturation_form	I2
(*qp*) pack2.uss $r_1 = r_2, r_3$	two_byte_form, unsigned_saturation_form	I2
(*qp*) pack4.sss $r_1 = r_2, r_3$	four_byte_form, signed_saturation_form	I2

Description: 32-bit or 16-bit elements from GR r_2 and GR r_3 are converted into 16-bit or 8-bit elements respectively, and the results are placed GR r_1. The source elements are treated as signed values. If a source element cannot be represented in the result element, then saturation clipping is performed. The saturation can either be signed or unsigned. If an element is larger than the upper limit value, the result is the upper limit value. If it is smaller than the lower limit value, the result is the lower limit value. The saturation limits are given in Table 2-42.

Table 2-42. Pack Saturation Limits

Size	Source Element Width	Result Element Width	Saturation	Upper Limit	Lower Limit
2	16 bit	8 bit	signed	0x7f	0x80
2	16 bit	8 bit	unsigned	0xff	0x00
4	32 bit	16 bit	signed	0x7fff	0x8000

Figure 2-28. Pack Operation



Operation:

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

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

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

            if (temp[i] < min)
                temp[i] = min;
        }

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

    } else {                                        // four_byte_form
        max = sign_ext(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.

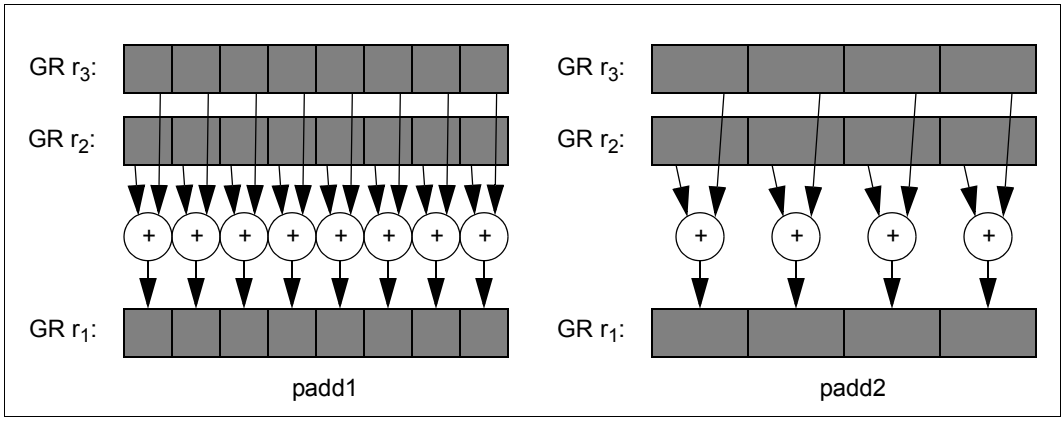
Table 2-43. Parallel Add Saturation Completers

Completer	Result r_1 treated as	Source r_2 treated as	Source r_3 treated as
sss	signed	signed	signed
uus	unsigned	unsigned	signed
uuu	unsigned	unsigned	unsigned

Table 2-44. Parallel Add Saturation Limits

Size	Element Width	Result r_1 Signed		Result r_1 Unsigned	
		Upper Limit	Lower Limit	Upper Limit	Lower Limit
1	8 bit	0x7f	0x80	0xff	0x00
2	16 bit	0x7fff	0x8000	0xffff	0x0000

Figure 2-29. Parallel Add Examples



Operation:

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

    if (one_byte_form) {                              // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (sss_saturation_form) {
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);

            for (i = 0; i < 8; i++) {
                temp[i] = sign_ext(x[i], 8) + sign_ext(y[i], 8);
            }
        } else if (uus_saturation_form) {
            max = 0xff;
            min = 0x00;

            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + sign_ext(y[i], 8);
            }
        } else if (uuu_saturation_form) {
            max = 0xff;
            min = 0x00;

            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
            }
        } else {                                      // modulo_form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
            }
        }

        if (sss_saturation_form || uus_saturation_form ||
            uuu_saturation_form) {
            for (i = 0; i < 8; i++) {
                if (temp[i] > max)
                    temp[i] = max;

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

    } else if (two_byte_form) {                       // 2-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
```

```
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (sss_saturation_form) {
            max = sign_ext(0x7fff, 16);
            min = sign_ext(0x8000, 16);

            for (i = 0; i < 4; i++) {
                temp[i] = sign_ext(x[i], 16) + sign_ext(y[i], 16);
            }
        } else if (uus_saturation_form) {
            max = 0xffff;
            min = 0x0000;

            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + sign_ext(y[i], 16);
            }
        } else if (uuu_saturation_form) {
            max = 0xffff;
            min = 0x0000;

            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
            }
        } else {                                        // modulo_form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
            }
        }

        if (sss_saturation_form || uus_saturation_form ||
            uuu_saturation_form) {
            for (i = 0; i < 4; i++) {
                if (temp[i] > max)
                    temp[i] = max;

                if (temp[i] < min)
                    temp[i] = min;
            }
        }
        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);

    } else {                                            // four-byte elements
        x[0] = GR[r2]{31:0};     y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};    y[1] = GR[r3]{63:32};

        for (i = 0; i < 2; i++) {                       // modulo_form
            temp[i] = zero_ext(x[i], 32) + zero_ext(y[i], 32);
        }

        GR[r1] = concatenate2(temp[1], temp[0]);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

pavg — Parallel Average

Format:

(*qp*) pavg1 $r_1 = r_2, r_3$	normal_form, one_byte_form	A9
(*qp*) pavg1.raz $r_1 = r_2, r_3$	raz_form, one_byte_form	A9
(*qp*) pavg2 $r_1 = r_2, r_3$	normal_form, two_byte_form	A9
(*qp*) pavg2.raz $r_1 = r_2, r_3$	raz_form, two_byte_form	A9

Description: The unsigned data elements of GR r_2 are added to the unsigned data elements of GR r_3. The results of the add are then each independently shifted to the right by one bit position. The high-order bits of each element are filled with the carry bits of the sums. To prevent cumulative round-off errors, an averaging is performed. The unsigned results are placed in GR r_1.

The averaging operation works as follows. In the normal_form, the low-order bit of each result is set to 1 if at least one of the two least significant bits of the corresponding sum is 1. In the raz_form, the average rounds away from zero by adding 1 to each of the sums.

Figure 2-30. Parallel Average Example

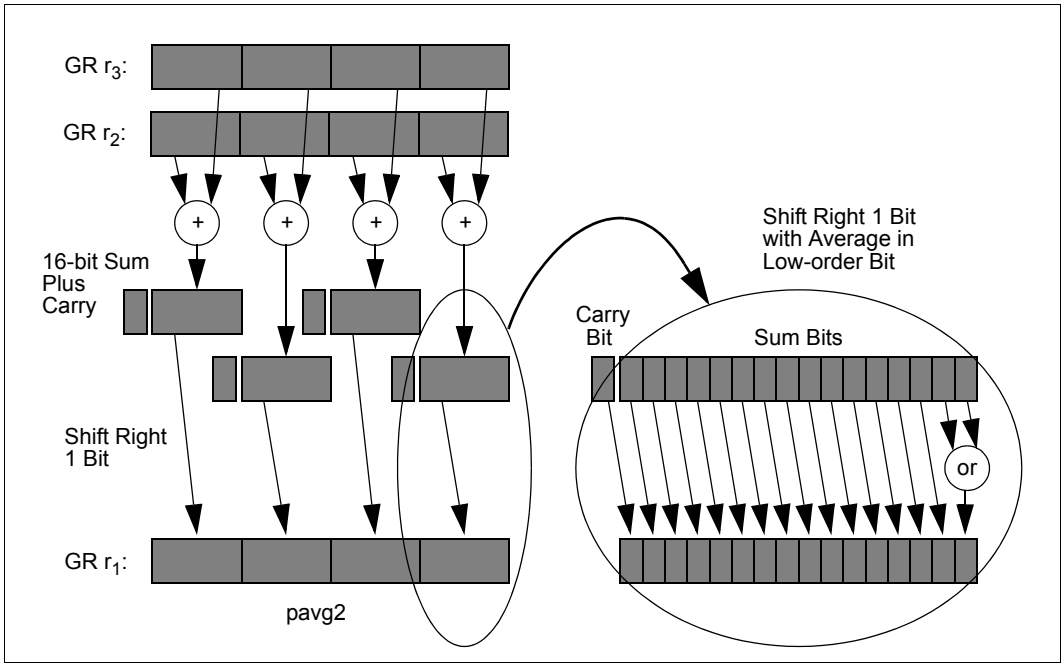
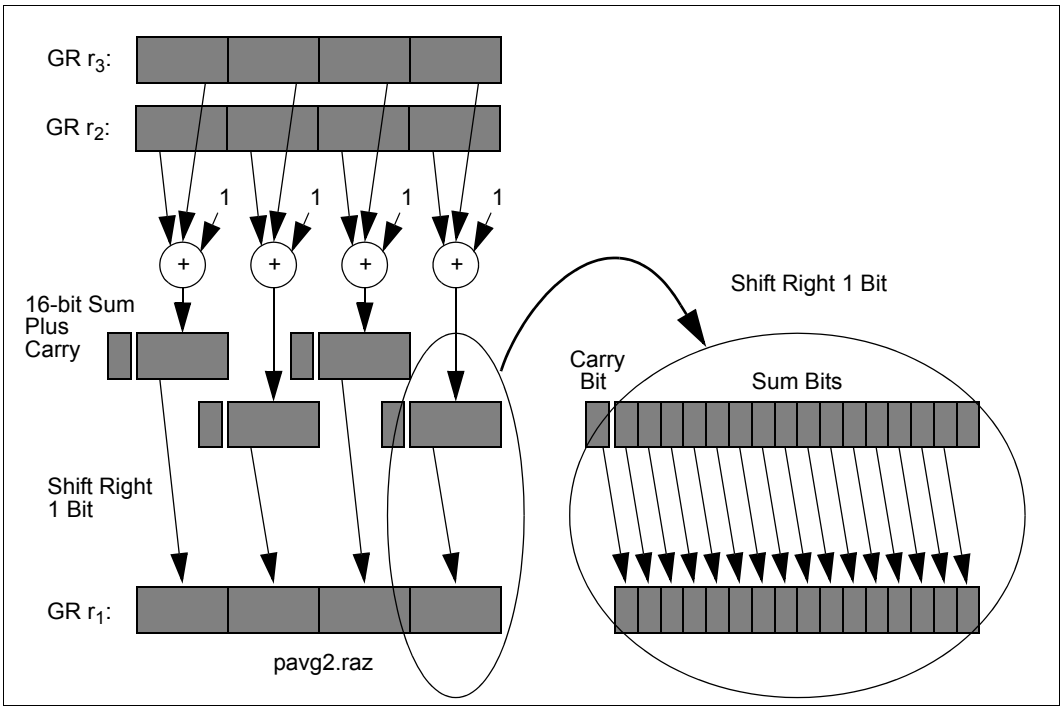


Figure 2-31. Parallel Average with Round Away from Zero Example



Operation:

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

    if (one_byte_form) {
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (raz_form) {
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8) + 1;
                res[i] = shift_right_unsigned(temp[i], 1);
            }
        } else {                                        // normal form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
                res[i] = shift_right_unsigned(temp[i], 1) | (temp[i]{0});
            }
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);

    } else {                                            // two_byte_form
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (raz_form) {
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16) + 1;
                res[i] = shift_right_unsigned(temp[i], 1);
            }
        } else {                                        // normal form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
                res[i] = shift_right_unsigned(temp[i], 1) | (temp[i]{0});
            }
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

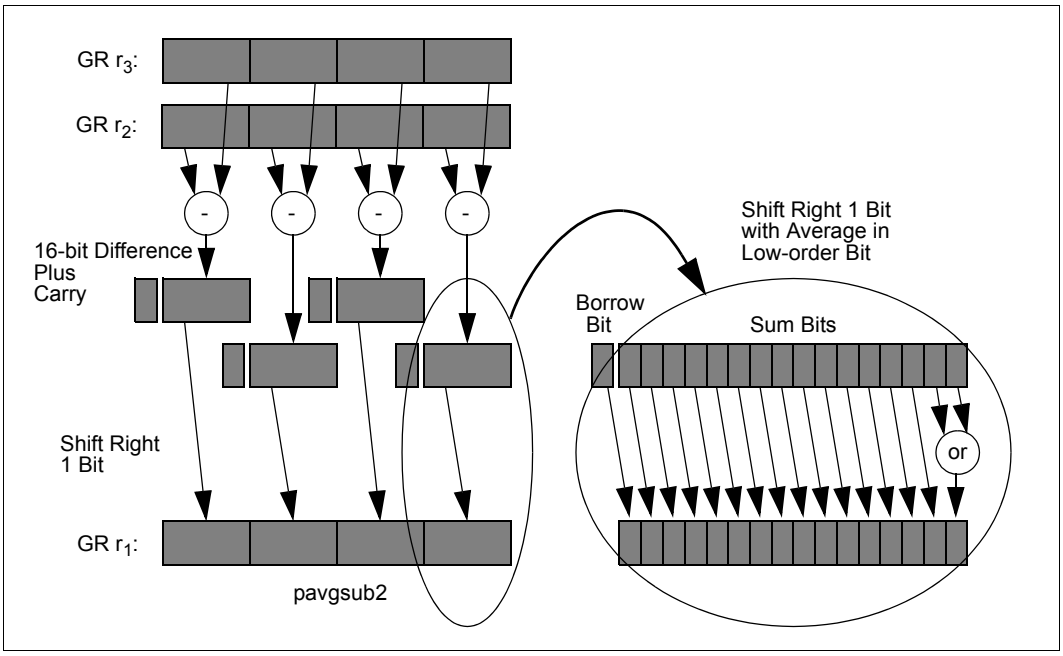
pavgsub — Parallel Average Subtract

Format:

(*qp*) pavgsub1 $r_1 = r_2, r_3$	one_byte_form	A9
(*qp*) pavgsub2 $r_1 = r_2, r_3$	two_byte_form	A9

Description: The unsigned data elements of GR r_3 are subtracted from the unsigned data elements of GR r_2. The results of the subtraction are then each independently shifted to the right by one bit position. The high-order bits of each element are filled with the borrow bits of the subtraction (the complements of the ALU carries). To prevent cumulative round-off errors, an averaging is performed. The low-order bit of each result is set to 1 if at least one of the two least significant bits of the corresponding difference is 1. The signed results are placed in GR r_1.

Figure 2-32. Parallel Average Subtract Example



Operation:

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

    if (one_byte_form) {
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        for (i = 0; i < 8; i++) {
            temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            res[i] = (temp[i]{8:0} u>> 1) | (temp[i]{0});
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);

    } else {                                          // two_byte_form
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        for (i = 0; i < 4; i++) {
            temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            res[i] = (temp[i]{16:0} u>> 1) | (temp[i]{0});
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

pcmp — Parallel Compare

Format:

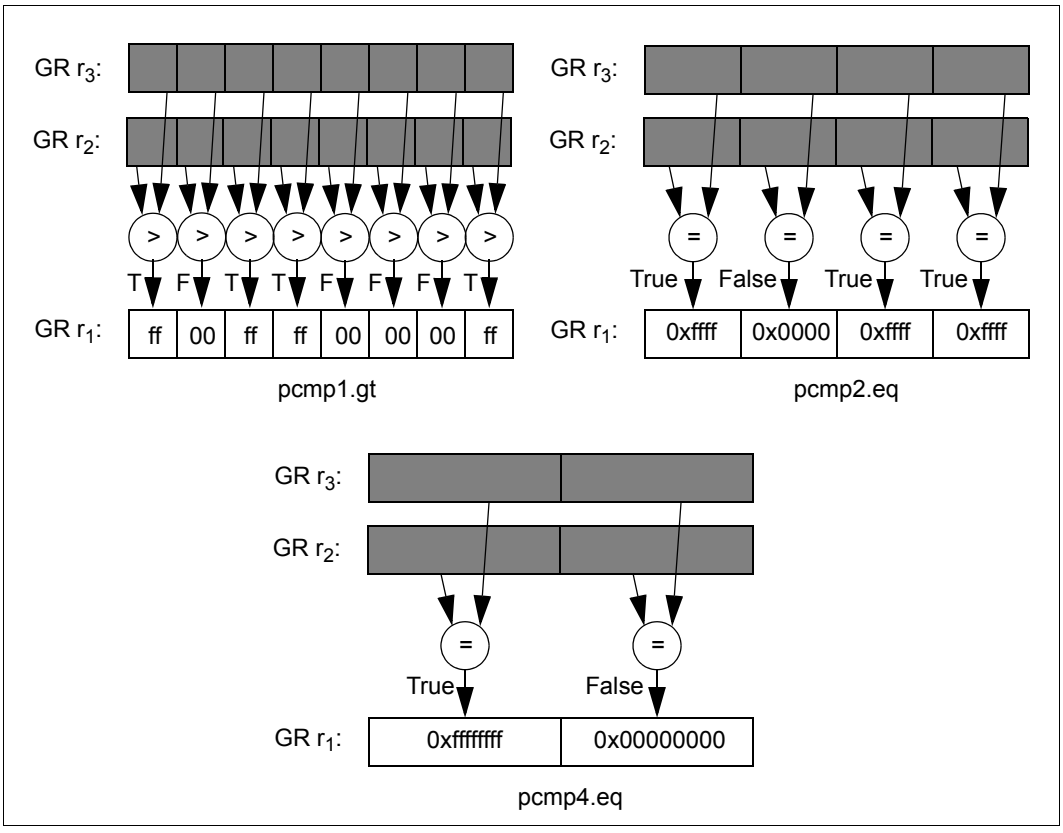
(*qp*) pcmp1.*prel* $r_1 = r_2, r_3$	one_byte_form	A9
(*qp*) pcmp2.*prel* $r_1 = r_2, r_3$	two_byte_form	A9
(*qp*) pcmp4.*prel* $r_1 = r_2, r_3$	four_byte_form	A9

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

prel	Compare Relation (r_2 *prel* r_3)
eq	$r_2 == r_3$
gt	$r_2 > r_3$ (signed)

Figure 2-33. Parallel Compare Examples



Operation:

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

    if (one_byte_form) {                              // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else // 'gt'
                tmp_rel = greater_signed(sign_ext(x[i], 8),
                                         sign_ext(y[i], 8));

            if (tmp_rel)
                res[i] = 0xff;
            else
                res[i] = 0x00;
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
    } else if (two_byte_form) {                       // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else // 'gt'
                tmp_rel = greater_signed(sign_ext(x[i], 16),
                                         sign_ext(y[i], 16));

            if (tmp_rel)
                res[i] = 0xffff;
            else
                res[i] = 0x0000;
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    } else {                                          // four-byte elements
        x[0] = GR[r2]{31:0};     y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};    y[1] = GR[r3]{63:32};
        for (i = 0; i < 2; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else // 'gt'
                tmp_rel = greater_signed(sign_ext(x[i], 32),
                                         sign_ext(y[i], 32));

            if (tmp_rel)
                res[i] = 0xffffffff;
```

```
                else
                    res[i] = 0x00000000;
            }
            GR[r1] = concatenate2(res[1], res[0]);
        }
        GR[r1].nat = GR[r2].nat || GR[r3].nat;
    }
```

Interruptions: Illegal Operation fault

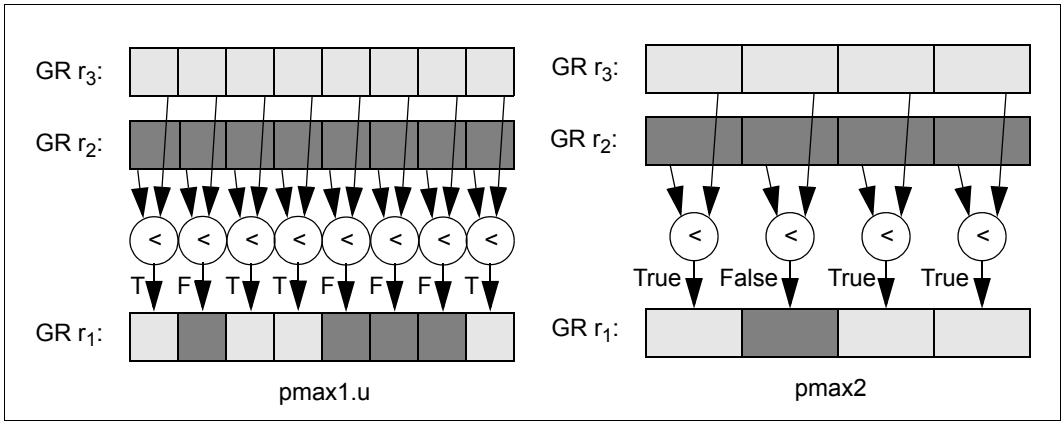
pmax — Parallel Maximum

Format:

(*qp*) pmax1.u $r_1 = r_2, r_3$	one_byte_form	I2
(*qp*) pmax2 $r_1 = r_2, r_3$	two_byte_form	I2

Description: The maximum of the two source operands is placed in the result register. In the one_byte_form, each unsigned 8-bit element of GR r_2 is compared with the corresponding unsigned 8-bit element of GR r_3 and the greater of the two is placed in the corresponding 8-bit element of GR r_1. In the two_byte_form, each signed 16-bit element of GR r_2 is compared with the corresponding signed 16-bit element of GR r_3 and the greater of the two is placed in the corresponding 16-bit element of GR r_1.

Figure 2-34. Parallel Maximum Examples



Operation:

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

    if (one_byte_form) {                              // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            res[i] = (zero_ext(x[i],8) < zero_ext(y[i],8)) ? y[i] : x[i];
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
    } else {                                          // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            res[i] = (sign_ext(x[i],16) < sign_ext(y[i],16)) ? y[i] : x[i];
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

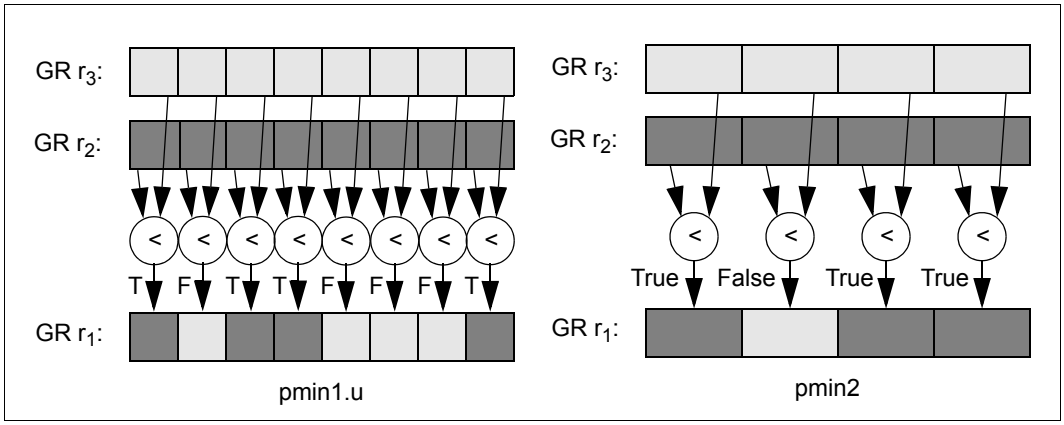
pmin — Parallel Minimum

Format:

(*qp*) pmin1.u $r_1 = r_2, r_3$	one_byte_form	I2
(*qp*) pmin2 $r_1 = r_2, r_3$	two_byte_form	I2

Description: The minimum of the two source operands is placed in the result register. In the one_byte_form, each unsigned 8-bit element of GR r_2 is compared with the corresponding unsigned 8-bit element of GR r_3 and the smaller of the two is placed in the corresponding 8-bit element of GR r_1. In the two_byte_form, each signed 16-bit element of GR r_2 is compared with the corresponding signed 16-bit element of GR r_3 and the smaller of the two is placed in the corresponding 16-bit element of GR r_1.

Figure 2-35. Parallel Minimum Examples



Operation:

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

    if (one_byte_form) {                              // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            res[i] = (zero_ext(x[i],8) < zero_ext(y[i],8)) ? x[i] : y[i];
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
    } else {                                          // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            res[i] = (sign_ext(x[i],16) < sign_ext(y[i],16)) ? x[i] : y[i];
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

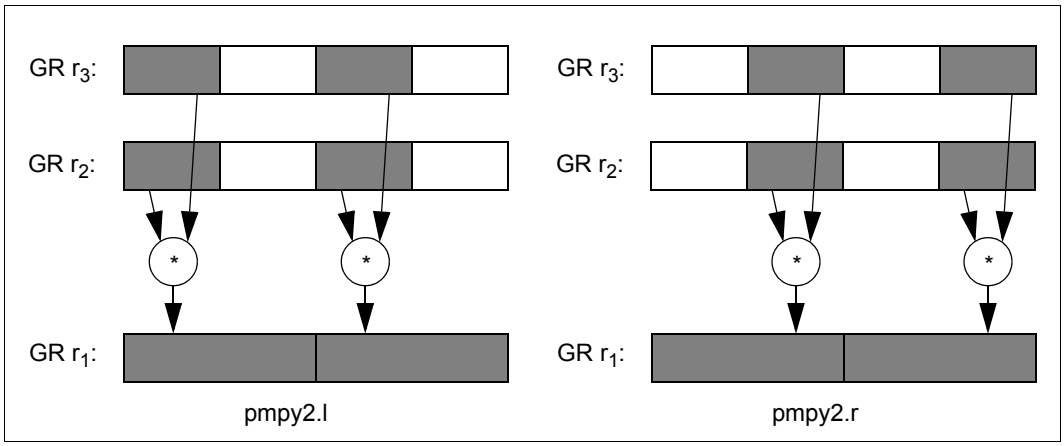
pmpy — Parallel Multiply

Format:

(*qp*) pmpy2.r $r_1 = r_2, r_3$	right_form	I2
(*qp*) pmpy2.l $r_1 = r_2, r_3$	left_form	I2

Description: Two signed 16-bit data elements of GR r_2 are multiplied by the corresponding two signed 16-bit data elements of GR r_3 as shown in Figure 2-36. The two 32-bit results are placed in GR r_1.

Figure 2-36. Parallel Multiply Operation



Operation:

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

    if (right_form) {
        GR[r1]{31:0} = sign_ext(GR[r2]{15:0}, 16) *
                       sign_ext(GR[r3]{15:0}, 16);
        GR[r1]{63:32} = sign_ext(GR[r2]{47:32}, 16) *
                       sign_ext(GR[r3]{47:32}, 16);
    } else {                                          // left_form
        GR[r1]{31:0} = sign_ext(GR[r2]{31:16}, 16) *
                       sign_ext(GR[r3]{31:16}, 16);
        GR[r1]{63:32} = sign_ext(GR[r2]{63:48}, 16) *
                       sign_ext(GR[r3]{63:48}, 16);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

pmpyshr — Parallel Multiply and Shift Right

Format:

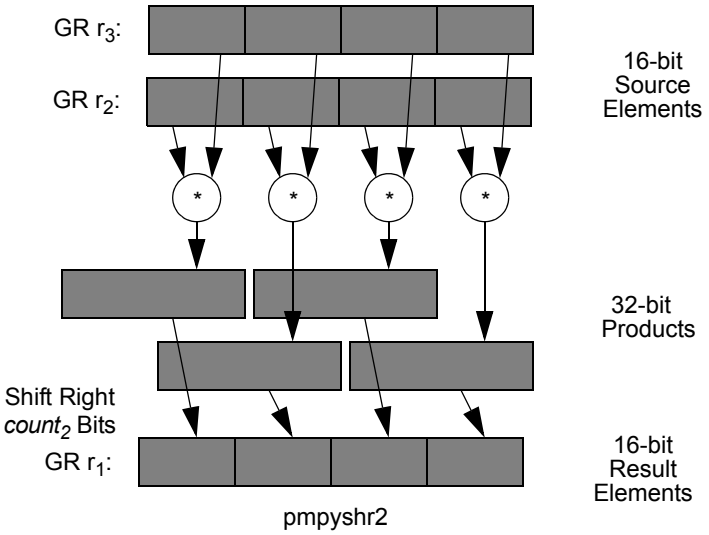
(*qp*) pmpyshr2 $r_1 = r_2, r_3, count_2$	signed_form	I1
(*qp*) pmpyshr2.u $r_1 = r_2, r_3, count_2$	unsigned_form	I1

Description: The four 16-bit data elements of GR r_2 are multiplied by the corresponding four 16-bit data elements of GR r_3 as shown in Figure 2-37. This multiplication can either be signed (pmpyshr2), or unsigned (pmpyshr2.u). Each product is then shifted to the right $count_2$ bits, and the least-significant 16-bits of each shifted product form 4 16-bit results, which are placed in GR r_1. A $count_2$ of 0 gives the 16 low bits of the results, a $count_2$ of 16 gives the 16 high bits of the results. The allowed values for $count_2$ are given in Table 2-46.

Table 2-46. Parallel Multiply and Shift Right Shift Options

$count_2$	Selected Bit Field from Each 32-bit Product
0	15:0
7	22:7
15	30:15
16	31:16

Figure 2-37. Parallel Multiply and Shift Right Operation



Operation:

```
if (PR[qp]) {
    check_target_register(r1);
    x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};
    for (i = 0; i < 4; i++) {
        if (unsigned_form)                          // unsigned multiplication
            temp[i] = zero_ext(x[i], 16) * zero_ext(y[i], 16);
        else                                        // signed multiplication
            temp[i] = sign_ext(x[i], 16) * sign_ext(y[i], 16);

        res[i] = temp[i]{(count2 + 15):count2};
    }

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

Interruptions: Illegal Operation fault

popcnt — Population Count

Format: (*qp*) popcnt $r_1 = r_3$ I9

Description: The number of bits in GR r_3 having the value 1 is counted, and the resulting sum is placed in GR r_1.

Operation:

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

    res = 0;
    // Count up all the one bits
    for (i = 0; i < 64; i++) {
        res += GR[r3]{i};
    }

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

Interruptions: Illegal Operation fault

probe — Probe Access

Format:

(*qp*) probe.r $r_1 = r_3, r_2$	regular_form, read_form, register_form	M38
(*qp*) probe.w $r_1 = r_3, r_2$	regular_form, write_form, register_form	M38
(*qp*) probe.r $r_1 = r_3, imm_2$	regular_form, read_form, immediate_form	M39
(*qp*) probe.w $r_1 = r_3, imm_2$	regular_form, write_form, immediate_form	M39
(*qp*) probe.r.fault r_3, imm_2	fault_form, read_form, immediate_form	M40
(*qp*) probe.w.fault r_3, imm_2	fault_form, write_form, immediate_form	M40
(*qp*) probe.rw.fault r_3, imm_2	fault_form, read_write_form, immediate_form	M40

Description:

This instruction determines whether read or write access, with a specified privilege level, to a given virtual address is permitted. In the regular_form, GR r_1 is set to 1 if the specified access is allowed and to 0 otherwise. In the fault_form, if the specified access is allowed this instruction does nothing; if the specified access is not allowed, a fault is taken.

When PSR.dt is 1, the DTLB and the VHPT are queried for present translations to determine if access to the virtual address specified by GR r_3 bits {60:0} and the region register indexed by GR r_3 bits {63:61}, is permitted at the privilege level given by either GR r_2 bits{1:0} or imm_2. If PSR.pk is 1, protection key checks are also performed. The read or write form specifies whether the instruction checks for read or write access, or both.

When PSR.dt is 0, a regular_form `probe` uses its address operand as a virtual address to query the DTLB only, because the VHPT walker is disabled. If the probed address is found in the DTLB, the regular_form `probe` returns the appropriate value, if not an Alternate Data TLB fault is raised if psr.ic is 1 or a Data Nested TLB fault is raised if psr.ic is 0 or in-flight.

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

A regular_form `probe` to an unimplemented virtual address returns 0. A fault_form `probe` to an unimplemented virtual address (when PSR.dt is 1) or unimplemented physical address (when PSR.dt is 0) takes an Unimplemented Data Address fault.

If this instruction faults, then it will set the non-access bit in the ISR and set the ISR read or write bits depending on the completer. The faults generated by the different forms of the `probe` instruction are shown in Table 2-47 below:

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

Probe Form Type	Faults
regular_form	Register NaT Consumption fault Virtualization fault[a] 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[a] Data Nested TLB fault Alternate Data TLB fault VHPT Data fault Data TLB fault Data Page Not Present fault Data NaT Page Consumption fault Data Key Miss fault Data Key Permission fault Data Access Rights fault Data Dirty Bit fault Data Access Bit fault Data Debug fault

a. This instruction may optionally raise Virtualization faults, see Section 11.7.4.2.8, "Probe Instruction Virtualization" on page 2:344 for details.

This instruction can only probe with equal or lower privilege levels. If the specified privilege level is higher (lower number), then the probe is performed with the current privilege level.

When PSR.vm is 1, this instruction may optionally raise Virtualization faults, see Section 11.7.4.2.8, "Probe Instruction Virtualization" on page 2:344 for details.

Please refer to the ***Intel® Itanium® Software Conventions and Runtime Architecture Guide*** for usage information of the `probe` instruction.

Operation:

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

    if (!fault_form)
        check_target_register(r1);

    if (GR[r3].nat || (register_form ? GR[r2].nat : 0))
        register_nat_consumption_fault(itype);

    tmp_pl = (register_form) ? GR[r2]{1:0} : imm2;
    if (tmp_pl < PSR.cpl)
        tmp_pl = PSR.cpl;

    if (fault_form) {
        tlb_translate(GR[r3], 1, itype, tmp_pl, &mattr, &defer);
    } else {             // regular_form
        if (impl_probe_intercept())
            check_probe_virtualization_fault(itype, tmp_pl);
        GR[r1] = tlb_grant_permission(GR[r3], itype, tmp_pl);
        GR[r1].nat = 0;
    }
}
```

Interruptions:

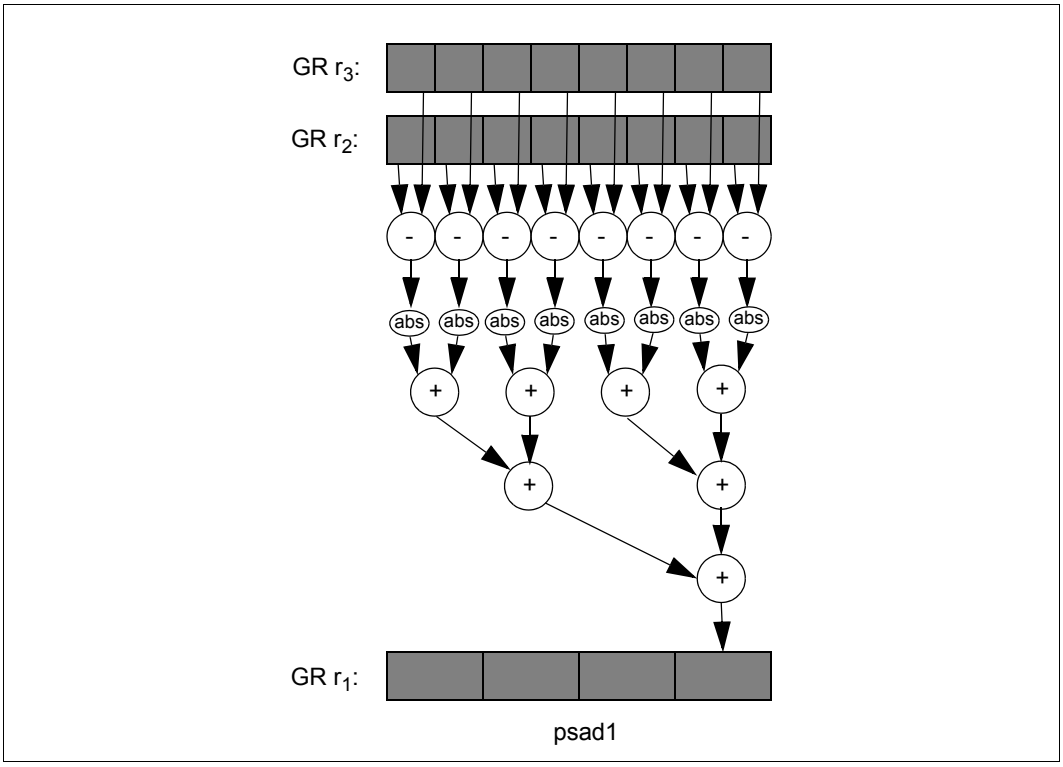
Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Virtualization fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access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

psad — Parallel Sum of Absolute Difference

Format: (*qp*) psad1 $r_1 = r_2, r_3$ I2

Description: The unsigned 8-bit elements of GR r_2 are subtracted from the unsigned 8-bit elements of GR r_3. The absolute value of each difference is accumulated across the elements and placed in GR r_1.

Figure 2-38. Parallel Sum of Absolute Difference Example



Operation:

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

    x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
    x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
    x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
    x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
    x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
    x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
    x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
    x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

    GR[r1] = 0;
    for (i = 0; i < 8; i++) {
        temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
        if (temp[i] < 0)
            temp[i] = -temp[i];
        GR[r1] += temp[i];
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

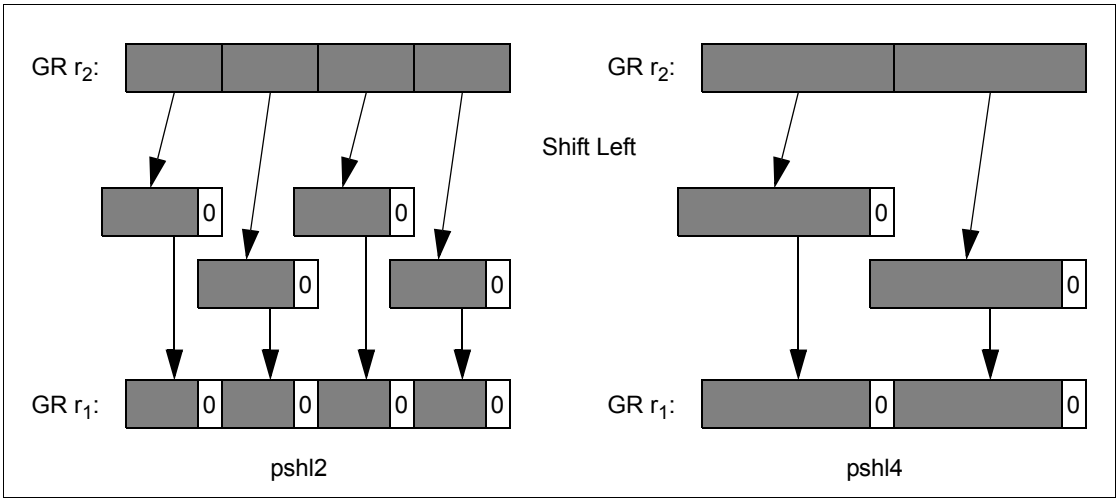
pshl — Parallel Shift Left

Format:

(*qp*) pshl2 $r_1 = r_2, r_3$	two_byte_form, variable_form	I7
(*qp*) pshl2 $r_1 = r_2, count_5$	two_byte_form, fixed_form	I8
(*qp*) pshl4 $r_1 = r_2, r_3$	four_byte_form, variable_form	I7
(*qp*) pshl4 $r_1 = r_2, count_5$	four_byte_form, fixed_form	I8

Description: The data elements of GR r_2 are each independently shifted to the left by the scalar shift count in GR r_3, or in the immediate field $count_5$. The low-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 15 (for 16-bit quantities) or 31 (for 32-bit quantities) yield all zero results. The results are placed in GR r_1.

Figure 2-39. Parallel Shift Left Examples



Operation:

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

    shift_count = (variable_form ? GR[r3] : count5);
    tmp_nat = (variable_form ? GR[r3].nat : 0);

    if (two_byte_form) {                              // two_byte_form
        if (shift_count u> 16)
            shift_count = 16;
        GR[r1]{15:0}  = GR[r2]{15:0}  << shift_count;
        GR[r1]{31:16} = GR[r2]{31:16} << shift_count;
        GR[r1]{47:32} = GR[r2]{47:32} << shift_count;
        GR[r1]{63:48} = GR[r2]{63:48} << shift_count;
    } else {                                          // four_byte_form
        if (shift_count u> 32)
            shift_count = 32;
        GR[r1]{31:0}  = GR[r2]{31:0}  << shift_count;
        GR[r1]{63:32} = GR[r2]{63:32} << shift_count;
    }

    GR[r1].nat = GR[r2].nat || tmp_nat;
}
```

Interruptions: Illegal Operation fault

pshladd — Parallel Shift Left and Add

Format: (*qp*) pshladd2 $r_1 = r_2, count_2, r_3$ A10

Description: The four signed 16-bit data elements of GR r_2 are each independently shifted to the left by $count_2$ bits (shifting zeros into the low-order bits), and added to the four signed 16-bit data elements of GR r_3. Both the left shift and the add operations are saturating: if the result of either the shift or the add is not representable as a signed 16-bit value, the final result is saturated. The four signed 16-bit results are placed in GR r_1. The first operand can be shifted by 1, 2 or 3 bits.

Operation:

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

    x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

    max = sign_ext(0x7fff, 16);
    min = sign_ext(0x8000, 16);

    for (i = 0; i < 4; i++) {
        temp[i] = sign_ext(x[i], 16) << count2;

        if (temp[i] > max)
            res[i] = max;
        else if (temp[i] < min)
            res[i] = min;
        else {
            res[i] = temp[i] + sign_ext(y[i], 16);
            if (res[i] > max)
                res[i] = max;
            if (res[i] < min)
                res[i] = min;
        }
    }

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

Interruptions: Illegal Operation fault

pshr — Parallel Shift Right

Format:			
(*qp*) pshr2 $r_1 = r_3, r_2$	signed_form, two_byte_form, variable_form	I5
(*qp*) pshr2 $r_1 = r_3, count_5$	signed_form, two_byte_form, fixed_form	I6
(*qp*) pshr2.u $r_1 = r_3, r_2$	unsigned_form, two_byte_form, variable_form	I5
(*qp*) pshr2.u $r_1 = r_3, count_5$	unsigned_form, two_byte_form, fixed_form	I6
(*qp*) pshr4 $r_1 = r_3, r_2$	signed_form, four_byte_form, variable_form	I5
(*qp*) pshr4 $r_1 = r_3, count_5$	signed_form, four_byte_form, fixed_form	I6
(*qp*) pshr4.u $r_1 = r_3, r_2$	unsigned_form, four_byte_form, variable_form	I5
(*qp*) pshr4.u $r_1 = r_3, count_5$	unsigned_form, four_byte_form, fixed_form	I6

Description: The data elements of GR r_3 are each independently shifted to the right by the scalar shift count in GR r_2, or in the immediate field $count_5$. The high-order bits of each element are filled with either the initial value of the sign bits of the data elements in GR r_3 (arithmetic shift) or zeros (logical shift). The shift count is interpreted as unsigned. Shift counts greater than 15 (for 16-bit quantities) or 31 (for 32-bit quantities) yield all zero or all one results depending on the initial values of the sign bits of the data elements in GR r_3 and whether a signed or unsigned shift is done. The results are placed in GR r_1.

Operation:

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

    shift_count = (variable_form ? GR[r2] : count5);
    tmp_nat = (variable_form ? GR[r2].nat : 0);

    if (two_byte_form) {                              // two_byte_form
        if (shift_count u> 16)
            shift_count = 16;
        if (unsigned_form) {                          // unsigned shift
            GR[r1]{15:0}  = shift_right_unsigned(zero_ext(GR[r3]{15:0},  16),
                                                 shift_count);
            GR[r1]{31:16} = shift_right_unsigned(zero_ext(GR[r3]{31:16}, 16),
                                                 shift_count);
            GR[r1]{47:32} = shift_right_unsigned(zero_ext(GR[r3]{47:32}, 16),
                                                 shift_count);
            GR[r1]{63:48} = shift_right_unsigned(zero_ext(GR[r3]{63:48}, 16),
                                                 shift_count);
        } else {                                      // signed shift
            GR[r1]{15:0}  = shift_right_signed(sign_ext(GR[r3]{15:0},  16),
                                               shift_count);
            GR[r1]{31:16} = shift_right_signed(sign_ext(GR[r3]{31:16}, 16),
                                               shift_count);
            GR[r1]{47:32} = shift_right_signed(sign_ext(GR[r3]{47:32}, 16),
                                               shift_count);
            GR[r1]{63:48} = shift_right_signed(sign_ext(GR[r3]{63:48}, 16),
                                               shift_count);
        }
    } else {                                          // four_byte_form
        if (shift_count > 32)
            shift_count = 32;
        if (unsigned_form) {                          // unsigned shift
            GR[r1]{31:0}  = shift_right_unsigned(zero_ext(GR[r3]{31:0},  32),
                                                 shift_count);
            GR[r1]{63:32} = shift_right_unsigned(zero_ext(GR[r3]{63:32}, 32),
                                                 shift_count);
        } else {                                      // signed shift
            GR[r1]{31:0}  = shift_right_signed(sign_ext(GR[r3]{31:0},  32),
                                               shift_count);
            GR[r1]{63:32} = shift_right_signed(sign_ext(GR[r3]{63:32}, 32),
                                               shift_count);
        }
    }

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

Interruptions: Illegal Operation fault

pshradd — Parallel Shift Right and Add

Format: (*qp*) pshradd2 $r_1 = r_2, count_2, r_3$ A10

Description: The four signed 16-bit data elements of GR r_2 are each independently shifted to the right by $count_2$ bits, and added to the four signed 16-bit data elements of GR r_3. The right shift operation fills the high-order bits of each element with the initial value of the sign bits of the data elements in GR r_2. The add operation is performed with signed saturation. The four signed 16-bit results of the add are placed in GR r_1. The first operand can be shifted by 1, 2 or 3 bits.

Operation:

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

    x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

    max = sign_ext(0x7fff, 16);
    min = sign_ext(0x8000, 16);

    for (i = 0; i < 4; i++) {
        temp[i] = shift_right_signed(sign_ext(x[i], 16), count2);

        res[i] = temp[i] + sign_ext(y[i], 16);
        if (res[i] > max)
            res[i] = max;
        if (res[i] < min)
            res[i] = min;
    }

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

Interruptions: Illegal Operation fault

psub — Parallel Subtract

Format:

(*qp*) psub1 $r_1 = r_2, r_3$	one_byte_form, modulo_form	A9
(*qp*) psub1.sss $r_1 = r_2, r_3$	one_byte_form, sss_saturation_form	A9
(*qp*) psub1.uus $r_1 = r_2, r_3$	one_byte_form, uus_saturation_form	A9
(*qp*) psub1.uuu $r_1 = r_2, r_3$	one_byte_form, uuu_saturation_form	A9
(*qp*) psub2 $r_1 = r_2, r_3$	two_byte_form, modulo_form	A9
(*qp*) psub2.sss $r_1 = r_2, r_3$	two_byte_form, sss_saturation_form	A9
(*qp*) psub2.uus $r_1 = r_2, r_3$	two_byte_form, uus_saturation_form	A9
(*qp*) psub2.uuu $r_1 = r_2, r_3$	two_byte_form, uuu_saturation_form	A9
(*qp*) psub4 $r_1 = r_2, r_3$	four_byte_form, modulo_form	A9

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

If the difference between two elements cannot be represented in the result element and a saturation completer is specified, then saturation clipping is performed. The saturation can either be signed or unsigned, as given in Table 2-48. If the difference of two elements is larger than the upper limit value, the result is the upper limit value. If it is smaller than the lower limit value, the result is the lower limit value. The saturation limits are given in Table 2-49.

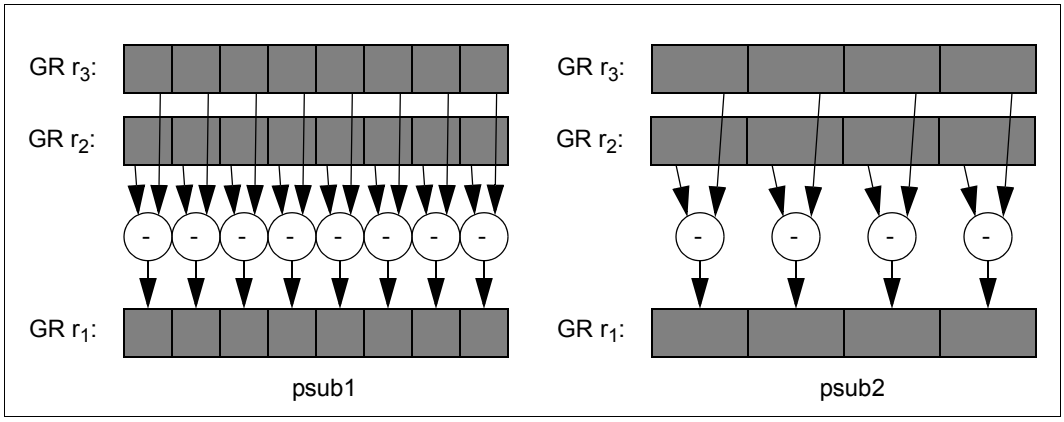
Table 2-48. Parallel Subtract Saturation Completers

Completer	Result r_1 treated as	Source r_2 treated as	Source r_3 treated as
sss	signed	signed	signed
uus	unsigned	unsigned	signed
uuu	unsigned	unsigned	unsigned

Table 2-49. Parallel Subtract Saturation Limits

Size	Element Width	Result r_1 Signed		Result r_1 Unsigned	
		Upper Limit	Lower Limit	Upper Limit	Lower Limit
1	8 bit	0x7f	0x80	0xff	0x00
2	16 bit	0x7fff	0x8000	0xffff	0x0000

Figure 2-40. Parallel Subtract Examples



Operation:

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

    if (one_byte_form) {                              // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (sss_saturation_form) {                    // sss_saturation_form
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);
            for (i = 0; i < 8; i++) {
                temp[i] = sign_ext(x[i], 8) - sign_ext(y[i], 8);
            }
        } else if (uus_saturation_form) {             // uus_saturation_form
            max = 0xff;
            min = 0x00;
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - sign_ext(y[i], 8);
            }
        } else if (uuu_saturation_form) {             // uuu_saturation_form
            max = 0xff;
            min = 0x00;
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            }
        } else {                                      // modulo_form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            }
        }

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

        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                              temp[3], temp[2], temp[1], temp[0]);
    } else if (two_byte_form) {                       // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (sss_saturation_form) {                    // sss_saturation_form
```

```
            max = sign_ext(0x7fff, 16);
            min = sign_ext(0x8000, 16);
            for (i = 0; i < 4; i++) {
                temp[i] = sign_ext(x[i], 16) - sign_ext(y[i], 16);
            }
        } else if (uus_saturation_form) {            // uus_saturation_form
            max = 0xffff;
            min = 0x0000;
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - sign_ext(y[i], 16);
            }
        } else if (uuu_saturation_form) {            // uuu_saturation_form
            max = 0xffff;
            min = 0x0000;
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            }
        } else {                                     // modulo_form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            }
        }

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

        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);
    } else {                                         // four-byte elements
        x[0] = GR[r2]{31:0};     y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};    y[1] = GR[r3]{63:32};

        for (i = 0; i < 2; i++) {                    // modulo_form
            temp[i] = zero_ext(x[i], 32) - zero_ext(y[i], 32);
        }

        GR[r1] = concatenate2(temp[1], temp[0]);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

ptc.e — Purge Translation Cache Entry

Format: (*qp*) ptc.e r_3 M47

Description: One or more translation entries are purged from the local processor's instruction and data translation cache. Translation Registers and the VHPT are not modified.

The number of translation cache entries purged is implementation specific. Some implementations may purge all levels of the translation cache hierarchy with one iteration of PTC.e, while other implementations may require several iterations to flush all levels, sets and associativities of both instruction and data translation caches. GR r_3 specifies an implementation-specific parameter associated with each iteration.

The following loop is defined to flush the entire translation cache for all processor models. Software can acquire parameters through a processor dependent layer that is accessed through a procedural interface. The selected region registers must remain unchanged during the loop.

```
disable_interrupts();
addr = base;
for (i = 0; i < count1; i++) {
    for (j = 0; j < count2; j++) {
           ptc.e(addr);
           addr += stride2;
       }
    addr += stride1;
}
enable_interrupts();
```

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

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[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	global_form	M45
(*qp*) ptc.ga r_3, r_2	global_alat_form	M45

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

The purge virtual address is specified by GR r_3 bits{60:0} and the purge region identifier is selected by GR r_3 bits {63:61}. GR r_2 specifies the address range of the purge as 1<<GR[r_2]{7:2} bytes in size. 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 what ever additional action is necessary to make their ALAT consistent. Matching entries in the local ALAT are optionally invalidated; software must perform a local ALAT invalidation via the `invala` instruction on the processor issuing the `ptc.ga` to ensure the local ALAT is coherent.

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

Unless specifically supported by the processors and platform, only one global purge transaction may be issued at a time by all processors, the operation is undefined otherwise. Software is responsible for enforcing this restriction. Implementations may optionally support multiple concurrent global purge transactions. The firmware returns if implementations support this optional behavior. It also returns the maximum number of simultaneous outstanding purges allowed.

Propagation of `ptc.g` between multiple local TLB coherence domains is platform dependent, and must be handled by software. It is expected that the local TLB coherence domain covers at least the processors on the same local bus.

Operation:

```
if (PR[qp]) {
    if (!followed_by_stop())
        undefined_behavior();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);
    tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);

    if (global_alat_form) tmp_ptc_type = GLOBAL_ALAT_FORM;
    else tmp_ptc_type = GLOBAL_FORM;

    tlb_broadcast_purge(tmp_rid, tmp_va, tmp_size, tmp_ptc_type);
}
```

Interruptions:

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

Serialization:

The broadcast purge TC is not synchronized with the instruction stream on a remote processor. Software cannot depend on any such synchronization with the instruction stream. Hardware on the remote machine cannot reload an instruction from memory or cache after acknowledging a broadcast purge TC without first retranslating the I-side access in the TLB. Hardware may continue to use a valid private copy of the instruction stream data (possibly in an I-buffer) obtained prior to acknowledging a broadcast purge TC to a page containing the i-stream data. Hardware must retranslate access to an instruction page upon an interruption or any explicit or implicit instruction serialization event (e.g., `srlz.i`, `rfi`).

Software must issue the appropriate data and/or instruction serialization operation to ensure the purge is completed before a local data access, non-access reference, or local instruction fetch access dependent upon the purge.

ptc.l — Purge Local Translation Cache

Format: (*qp*) ptc.l r_3, r_2 M45

Description: The instruction and data translation cache of the local processor is searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. All these entries are removed.

The purge virtual address is specified by GR r_3 bits{60:0} and the purge region identifier is selected by GR r_3 bits {63:61}. GR r_2 specifies the address range of the purge as 1<<GR[r_2]{7:2} bytes in size. See Section 4.1.1.7, "Page Sizes" on page 2:57 for details on supported page sizes for TLB purges.

The processor ensures that all entries matching the purging parameters are removed. However, based on the processor model, the translation cache may be also purged of more translations than specified by the purge parameters up to and including removal of all entries within the translation cache.

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

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

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);
    tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
}
```

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

Serialization: Software must issue the appropriate data and/or instruction serialization operation to ensure the purge is completed before a data access, non-access reference, or instruction fetch access dependent upon the purge.

ptr — Purge Translation Register

Format:

(*qp*) ptr.d r_3, r_2	data_form	M45
(*qp*) ptr.i r_3, r_2	instruction_form	M45

Description:

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

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

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

The purge virtual address is specified by GR r_3 bits{60:0} and the purge region identifier is selected by GR r_3 bits {63:61}. GR r_2 specifies the address range of the purge as 1<<GR[r_2]{7:2} bytes in size. See Section 4.1.1.7, “Page Sizes” on page 2:57 for details on supported page sizes for TLB purges.

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

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

As described in Section 4.1.1.2, “Translation Cache (TC)” on page 2:49, the processor may use the translation caches to cache virtual address mappings held by translation registers. The `ptr.i` and `ptr.d` instructions purge the processor’s translation registers as well as cached translation register copies that may be contained in the respective translation caches.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    if (data_form) {
        tlb_must_purge_dtr_entries(tmp_rid, tmp_va, tmp_size);
        tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
    } else {                                          // instruction_form
        tlb_must_purge_itr_entries(tmp_rid, tmp_va, tmp_size);
        tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    }
}
```

Interruptions: Privileged Operation fault; Unimplemented Data Address fault; Register NaT Consumption fault; Virtualization fault

Serialization: For the data form, software must issue a data serialization operation to ensure the purge is completed before issuing an instruction dependent upon the purge. For the instruction form, software must issue an instruction serialization operation to ensure the purge is completed before fetching an instruction dependent on that purge.

rfi — Return From Interruption

Format: rfi B8

Description: The machine context prior to an interruption is restored. PSR is restored from IPSR, IPSR is unmodified, and IP is restored from IIP. Execution continues at the bundle address loaded into the IP, and the instruction slot loaded into PSR.ri.

This instruction must be immediately followed by a stop; otherwise, operation is undefined. This instruction switches to the register bank specified by IPSR.bn. Instructions in the same instruction group that access GR16 to GR31 reference the previous register bank. Subsequent instruction groups reference the new register bank.

This instruction performs instruction serialization, which ensures:

- prior modifications to processor register resources that affect fetching of subsequent instruction groups are observed.
- prior modifications to processor register resources that affect subsequent execution or data memory accesses are observed.
- prior memory synchronization (`sync.i`) operations have taken effect on the local processor instruction cache.
- subsequent instruction group fetches (including the target instruction group) are re-initiated after `rfi` completes.

The `rfi` instruction must be in an instruction group after the instruction group containing the operation that is to be serialized.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0. This instruction can not be predicated.

Execution of this instruction is undefined if PSR.ic or PSR.i are 1. Software must ensure that an interruption cannot occur that could modify IIP, IPSR, or IFS between when they are written and the subsequent `rfi`.

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

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

If this instruction sets PSR.ri to 2 and the target is an MLX bundle, then an Illegal Operation fault will be taken on the target bundle.

If IPSR.is is 1, control is resumed in the IA-32 instruction set at the virtual linear address specified by IIP{31:0}. PSR.di does not inhibit instruction set transitions for this instruction. If PSR.dfh is 1 after `rfi` completes execution, a Disabled FP Register fault is raised on the target IA-32 instruction.

If IPSR.is is 1 and an Unimplemented Instruction Address trap is taken, IIP will contain the original 64-bit target IP. (The value will not have been zero extended from 32 bits.)

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

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

Software must issue a `mf` instruction before this instruction if memory ordering is required between IA-32 processor-consistent and Itanium unordered memory references. The processor does not ensure Itanium-instruction-set-generated writes into the instruction stream are seen by subsequent IA-32 instructions.

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

IA-32 instruction set execution leaves the contents of the ALAT undefined. Software can not rely on ALAT state across an instruction set transition. On entry to IA-32 code, existing entries in the ALAT are ignored.

Operation:

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

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

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

taken_rfi = 1;

PSR = CR[IPSR];
if (CR[IPSR].is == 1) {         //resume IA-32 instruction set
    if (CR[IPSR].ic == 0 || CR[IPSR].dt == 0 ||
        CR[IPSR].mc == 1 || CR[IPSR].it == 0)
        undefined_behavior();
    tmp_IP = CR[IIP];
    if (!impl_uia_fault_supported() &&
        ((CR[IPSR].it && unimplemented_virtual_address(tmp_IP, IPSR.vm))
        || (!CR[IPSR].it && unimplemented_physical_address(tmp_IP))))
        unimplemented_address = 1;
                                //compute effective instruction pointer
    EIP{31:0} = CR[IIP]{31:0} - AR[CSD].Base;
                                //force zero-sized restored frame
    rse_restore_frame(0, 0, CFM.sof);
    CFM.sof = 0;
    CFM.sol = 0;
    CFM.sor = 0;
    CFM.rrb.gr = 0;
    CFM.rrb.fr = 0;
    CFM.rrb.pr = 0;
    rse_invalidate_non_current_regs();
    //The register stack engine is disabled during IA-32
```

```
    //instruction set execution.
} else {                        //return to Itanium instruction set
    tmp_IP = CR[IIP] & ~0xf;
    slot = CR[IPSR].ri;
    if ((CR[IPSR].it && unimplemented_virtual_address(tmp_IP, IPSR.vm))
        || (!CR[IPSR].it && unimplemented_physical_address(tmp_IP)))
        unimplemented_address = 1;
    if (CR[IFS].v) {
        tmp_growth = -CFM.sof;
        alat_frame_update(-CR[IFS].ifm.sof, 0);
        rse_restore_frame(CR[IFS].ifm.sof, tmp_growth, CFM.sof);
        CFM = CR[IFS].ifm;
    }
    rse_enable_current_frame_load();
}
IP = tmp_IP;
instruction_serialize();
if (unimplemented_address)
    unimplemented_instruction_address_trap(0, tmp_IP);
```

Interruptions: Privileged Operation fault 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 *imm*$_{24}$ M44

Description: The complement of the *imm*$_{24}$ operand is ANDed with the system mask (PSR{23:0}) and the result is placed in the system mask. 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;)     // dfl
    if (imm24{19})    PSR{19} = 0;)     // dfh
    if (imm24{20})    PSR{20} = 0;)     // sp
```

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

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: (*qp*) rum *imm*$_{24}$ M44

Description: The complement of the *imm*$_{24}$ operand is ANDed with the user mask (PSR{5:0}) and the result is placed in the user mask. See Section 3.3.2, "Processor Status Register (PSR)" on page 2:23.

PSR.up is only cleared if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Operation:

```
if (PR[qp]) {
    if (is_reserved_field(PSR_TYPE, PSR_UM, imm24))
        reserved_register_field_fault();

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

Interruptions: Reserved Register/Field fault

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

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

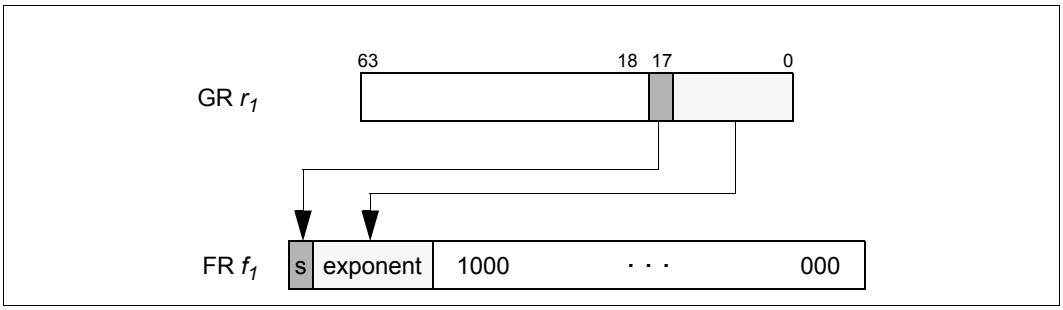
Format:

(*qp*) setf.s $f_1 = r_2$	single_form	M18
(*qp*) setf.d $f_1 = r_2$	double_form	M18
(*qp*) setf.exp $f_1 = r_2$	exponent_form	M18
(*qp*) setf.sig $f_1 = r_2$	significand_form	M18

Description: In the single and double forms, GR r_2 is treated as a single precision (in the single_form) or double precision (in the double_form) memory representation, converted into floating-point register format, and placed in FR f_1, as shown in Figure 5-4 and Figure 5-5 on page 1:93, respectively.

In the exponent_form, bits 16:0 of GR r_2 are copied to the exponent field of FR f_1 and bit 17 of GR r_2 is copied to the sign bit of FR f_1. The significand field of FR f_1 is set to one (0x800...000).

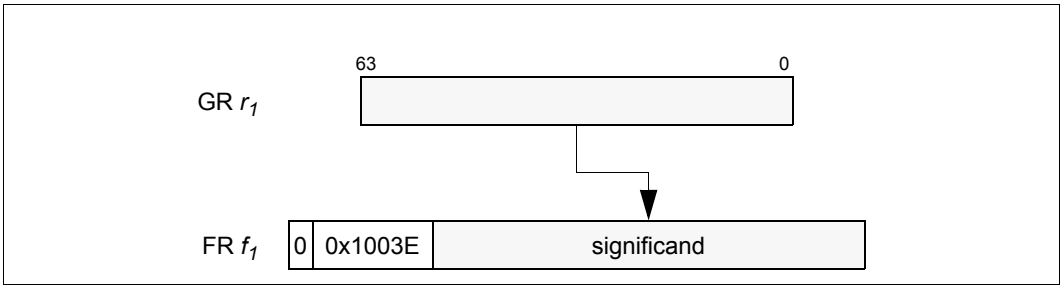
Figure 2-41. Function of setf.exp



In the significand_form, the value in GR r_2 is copied to the significand field of FR f_1.

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

Figure 2-42. Function of setf.sig



For all forms, if the NaT bit corresponding to r_2 is equal to 1, FR f_1 is set to NaTVal instead of the computed result.

Operation:

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

    if (!GR[r2].nat) {
        if (single_form)
            FR[f1] = fp_mem_to_fr_format(GR[r2], 4, 0);
        else if (double_form)
            FR[f1] = fp_mem_to_fr_format(GR[r2], 8, 0);
        else if (significand_form) {
            FR[f1].significand = GR[r2];
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = 0;
        } else {                                        // exponent_form
            FR[f1].significand = 0x8000000000000000;
            FR[f1].exp = GR[r2]{16:0};
            FR[f1].sign = GR[r2]{17};
        }
    } else
        FR[f1] = NATVAL;

    fp_update_psr(f1);
}
```

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

shl — Shift Left

Format: (*qp*) shl $r_1 = r_2, r_3$ I7
(*qp*) shl $r_1 = r_2, count_6$ pseudo-op of: (*qp*) dep.z $r_1 = r_2, count_6, 64\text{-}count_6$

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

See "dep — Deposit" on page 3:51 for the immediate form.

Operation:

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

    count = GR[r3];
    GR[r1] = (count > 63) ? 0: GR[r2] << count;

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

shladd — Shift Left and Add

Format: (*qp*) shladd $r_1 = r_2, count_2, r_3$ A2

Description: The first source operand is shifted to the left by $count_2$ bits and then added to the second source operand and the result placed in GR r_1. The first operand can be shifted by 1, 2, 3, or 4 bits.

Operation:

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

    GR[r1] = (GR[r2] << count2) + GR[r3];
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

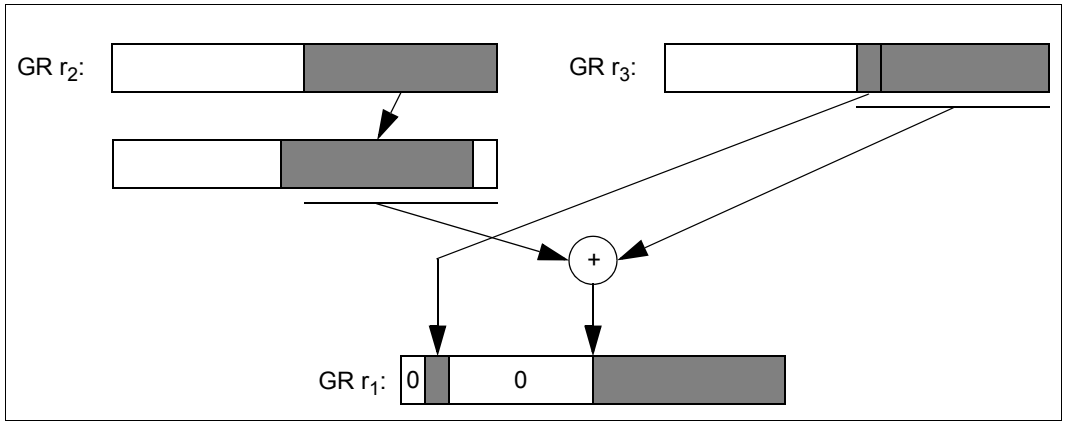
Interruptions: Illegal Operation fault

shladdp4 — Shift Left and Add Pointer

Format: (*qp*) shladdp4 $r_1 = r_2, count_2, r_3$ A2

Description: The first source operand is shifted to the left by $count_2$ bits and then is added to the second source operand. The upper 32 bits of the result are forced to zero, and then bits {31:30} of GR r_3 are copied to bits {62:61} of the result. This result is placed in GR r_1. The first operand can be shifted by 1, 2, 3, or 4 bits.

Figure 2-43. Shift Left and Add Pointer



Operation:

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

    tmp_res = (GR[r2] << count2) + GR[r3];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[r3]{31:30};
    GR[r1] = tmp_res;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

shr — Shift Right

Format:

(*qp*) shr $r_1 = r_3, r_2$	signed_form	I5
(*qp*) shr.u $r_1 = r_3, r_2$	unsigned_form	I5
(*qp*) shr $r_1 = r_3, count_6$	pseudo-op of: (*qp*) extr $r_1 = r_3, count_6, 64\text{-}count_6$	
(*qp*) shr.u $r_1 = r_3, count_6$	pseudo-op of: (*qp*) extr.u $r_1 = r_3, count_6, 64\text{-}count_6$	

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 $count_6$. The shift count is interpreted as an unsigned number. If the value in GR r_2 is greater than 63, then the result is all zeroes (for the unsigned_form, or if bit 63 of GR r_3 was 0) or all ones (for the signed_form if bit 63 of GR r_3 was 1).

If the .u completer is specified, the shift is unsigned (logical), otherwise it is signed (arithmetic).

See "extr — Extract" on page 3:54 for the immediate forms.

Operation:

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

    if (signed_form) {
        count = (GR[r2] > 63) ? 63 : GR[r2];
        GR[r1] = shift_right_signed(GR[r3], count);
    } else {
        count = GR[r2];
        GR[r1] = (count > 63) ? 0 : shift_right_unsigned(GR[r3], count);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

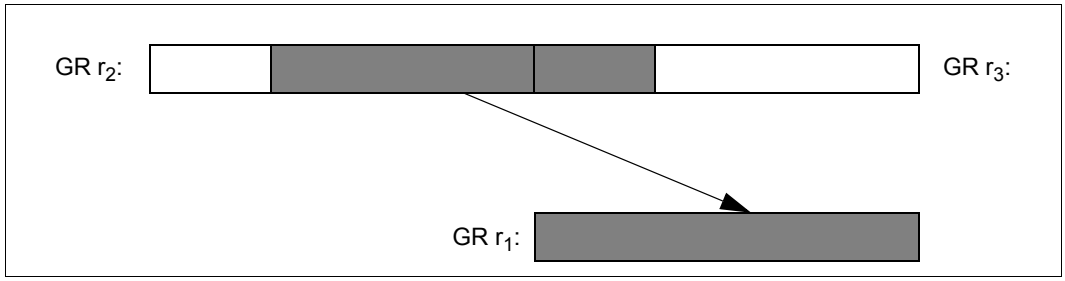
shrp — Shift Right Pair

Format: (*qp*) shrp $r_1 = r_2, r_3, count_6$ I10

Description: The two source operands, GR r_2 and GR r_3, are concatenated to form a 128-bit value and shifted to the right $count_6$ bits. The least-significant 64 bits of the result are placed in GR r_1.

The immediate value $count_6$ can be any number in the range 0 to 63.

Figure 2-44. Shift Right Pair



Operation:

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

    temp1 = shift_right_unsigned(GR[r3], count6);
    temp2 = GR[r2] << (64 - count6);
    GR[r1] = zero_ext(temp1, 64 - count6) | temp2;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

srlz — Serialize

Format:

(*qp*) srlz.i	instruction_form	M24
(*qp*) srlz.d	data_form	M24

Description: Instruction serialization (`srlz.i`) ensures:

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

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

Data serialization (`srlz.d`) ensures:

- prior modifications to processor register resources that affect subsequent execution or data memory accesses are observed.

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

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

The following processor resources require a serialize to ensure side-effects are observed; CRs, PSR, DBRs, IBRs, PMDs, PMCs, RRs, PKRs, TRs and TCs (refer to 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} M44

Description: The imm_{24} operand is ORed with the system mask (PSR{23:0}) and the result is placed in the system mask. See Section 3.3.2, "Processor Status Register (PSR)" on page 2:23.

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

The contents of the interruption resources (that are overwritten when the PSR.ic bit is 1), are undefined if an interruption occurs between the enabling of the PSR.ic bit and a subsequent instruction serialize operation.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (is_reserved_field(PSR_TYPE, PSR_SM, imm24))
        reserved_register_field_fault();

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

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

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*) st*sz*.*sttype*.*sthint* [r_3] = r_2	normal_form, no_base_update_form	M6
(*qp*) st*sz*.*sttype*.*sthint* [r_3] = r_2, imm_9	normal_form, imm_base_update_form	M5
(*qp*) st16.*sttype*.*sthint* [r_3] = r_2, ar.csd	sixteen_byte_form, no_base_update_form	M6
(*qp*) st8.spill.*sthint* [r_3] = r_2	spill_form, no_base_update_form	M6
(*qp*) st8.spill.*sthint* [r_3] = r_2, imm_9	spill_form, imm_base_update_form	M5

Description:

A value consisting of the least significant *sz* bytes of the value in GR r_2 is written to memory starting at the address specified by the value in GR r_3. The values of the *sz* completer are given in Table 2-32 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 r_3 is 1, or in sixteen_byte_form or normal_form, if the NaT bit corresponding to GR r_2 is 1, a Register NaT Consumption fault is taken.

In the sixteen_byte_form, two 8-byte values are stored as a single, 16-byte atomic memory write. The value in GR r_2 is written to memory starting at the address specified by the value in GR r_3. The value in the Compare and Store Data application register (AR[CSD]) is written to memory starting at the address specified by the value in GR r_3 plus 8.

In the spill_form, an 8-byte value is stored, and the NaT bit corresponding to GR r_2 is copied to a bit in the UNAT application register. This instruction is used for spilling a register/NaT pair. See Section 4.4.4, "Control Speculation" on page 1:60 for details.

In the imm_base_update form, the value in GR r_3 is added to a signed immediate value (imm_9) and the result is placed back in GR r_3. This base register update is done after the store, and does not affect the store address, nor the value stored (for the case where r_2 and r_3 specify the same register). Base register update is not supported for the `st16` instruction.

Table 2-50. Store Types

sttype Completer	Interpretation	Special Store Operation
none	Normal store	
rel	Ordered store	An ordered store is performed with release semantics.

For more details on ordered stores see Section 4.4.7, "Memory Access Ordering" on page 1:73.

The ALAT is queried using the physical memory address and the access size, and all overlapping entries are invalidated.

The value of the *sthint* completer specifies the locality of the memory access. The values of the *sthint* completer are given in Table 2-51. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 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

sthint Completer	Interpretation
none	Temporal locality, level 1
nta	Non-temporal locality, all levels

Operation:

```
if (PR[qp]) {
    size = spill_form ? 8 : (sixteen_byte_form ? 16 : sz);
    itype = WRITE;
    if (size == 16) itype |= UNCACHE_OPT;
    otype = (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

Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Unaligned Data Reference fault
Unsupported Data Reference fault

stf — Floating-point Store

Format:

(*qp*) stf*fsz*.*sthint* [r_3] = f_2	normal_form, no_base_update_form	M13
(*qp*) stf*fsz*.*sthint* [r_3] = f_2, imm_9	normal_form, imm_base_update_form	M10
(*qp*) stf8.*sthint* [r_3] = f_2	integer_form, no_base_update_form	M13
(*qp*) stf8.*sthint* [r_3] = f_2, imm_9	integer_form, imm_base_update_form	M10
(*qp*) stf.spill.*sthint* [r_3] = f_2	spill_form, no_base_update_form	M13
(*qp*) stf.spill.*sthint* [r_3] = f_2, imm_9	spill_form, imm_base_update_form	M10

Description:

A value, consisting of *fsz* bytes, is generated from the value in FR f_2 and written to memory starting at the address specified by the value in GR r_3. In the normal_form, the value in FR f_2 is converted to the memory format and then stored. In the integer_form, the significand of FR f_2 is stored. The values of the *fsz* completer are given in Table 2-35 on page 3:157. In the normal_form or the integer_form, if the NaT bit corresponding to GR r_3 is 1 or if FR f_2 contains NaTVal, a Register NaT Consumption fault is taken. See Section 5.1, "Data Types and Formats" on page 1:85 for details on conversion from floating-point register format.

In the spill_form, a 16-byte value from FR f_2 is stored without conversion. This instruction is used for spilling a register. See Section 4.4.4, "Control Speculation" on page 1:60 for details.

In the imm_base_update form, the value in GR r_3 is added to a signed immediate value (imm_9) and the result is placed back in GR r_3. This base register update is done after the store, and does not affect the store address.

The ALAT is queried using the physical memory address and the access size, and all overlapping entries are invalidated.

The value of the *sthint* completer specifies the locality of the memory access. The values of the *sthint* completer are given in Table 2-51 on page 3:252. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *sthint*. See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69.

Hardware support for `stfe` (10-byte) instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `stfe` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.

Operation:

```
if (PR[qp]) {
    if (imm_base_update_form)
        check_target_register(r3);
    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, WRITE);

    if (GR[r3].nat || (!spill_form && (FR[f2] == NATVAL)))
        register_nat_consumption_fault(WRITE);

    size = spill_form ? 16 : (integer_form ? 8 : fsz);
    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:

(*qp*) sub $r_1 = r_2, r_3$	register_form	A1
(*qp*) sub $r_1 = r_2, r_3, 1$	minus1_form, register_form	A1
(*qp*) sub $r_1 = imm_8, r_3$	imm8_form	A3

Description: The second source operand (and an optional constant 1) are subtracted from the first operand and the result placed in GR r_1. In the register form the first operand is GR r_2; in the immediate form the first operand is taken from the sign-extended imm_8 encoding field.

The minus1_form is available only in the register_form (although the equivalent effect can be achieved by adjusting the immediate).

Operation:

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

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

    if (minus1_form)
        GR[r1] = tmp_src - GR[r3] - 1;
    else
        GR[r1] = tmp_src - GR[r3];

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

Interruptions: Illegal Operation fault

sum — Set User Mask

Format: (*qp*) sum imm_{24} M44

Description: The imm_{24} operand is ORed with the user mask (PSR{5:0}) and the result is placed in the user mask. See Section 3.3.2, "Processor Status Register (PSR)" on page 2:23.

PSR.up can only be set if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Operation:

```
if (PR[qp]) {
    if (is_reserved_field(PSR_TYPE, PSR_UM, imm24))
        reserved_register_field_fault();

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

Interruptions: Reserved Register/Field fault

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

sxt — Sign Extend

Format: (*qp*) sxt*xsz* $r_1 = r_3$ I29

Description: The value in GR r_3 is sign extended from the bit position specified by *xsz* and the result is placed in GR r_1. The mnemonic values for *xsz* are given in Table 2-52.

Table 2-52. *xsz* Mnemonic Values

xsz Mnemonic	Bit Position
1	7
2	15
4	31

Operation:

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

    GR[r1] = sign_ext(GR[r3],xsz * 8);
    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

sync — Memory Synchronization

Format: (*qp*) sync.i M24

Description: `sync.i` ensures that when previously initiated Flush Cache (`fc`, `fc.i`) operations issued by the local processor become visible to local data memory references, prior Flush Cache operations are also observed by the local processor instruction fetch stream. `sync.i` also ensures that at the time previously initiated Flush Cache (`fc`, `fc.i`) operations are observed on a remote processor by data memory references they are also observed by instruction memory references on the remote processor. `sync.i` is ordered with respect to all cache flush operations as observed by another processor. A `sync.i` and a previous `fc` must be in separate instruction groups. If semantically required, the programmer must explicitly insert ordered data references (acquire, release or fence type) to appropriately constrain `sync.i` (and hence `fc` and `fc.i`) visibility to the data stream on other processors.

`sync.i` is used to maintain an ordering relationship between instruction and data caches on local and remote processors. An instruction serialize operation must be used to ensure synchronization initiated by `sync.i` on the local processor has been observed by a given point in program execution.

An example of self-modifying code (local processor):

```
    st  [L1] = data   //store into local instruction stream
    fc.i L1           //flush stale datum from instruction/data cache
    ;;                //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 $r_1 = r_3$ M46

Description: The protection key for a given virtual address is obtained and placed in GR r_1.

When PSR.dt is 1, the DTLB and the VHPT are searched for the virtual address specified by GR r_3 and the region register indexed by GR r_3 bits {63:61}. If a matching present translation is found, the protection key of the translation is placed in bits 31:8 of GR r_1. If a matching present translation is not found or if an unimplemented virtual address is specified by GR r_3, the value 1 is returned.

When PSR.dt is 0, only the DTLB is searched, because the VHPT walker is disabled. If no matching present translation is found in the DTLB, the value 1 is returned.

A translation with the NaTPage attribute is not treated differently and returns its key field.

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

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS|TAK;
    check_target_register(r1);

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

    if (GR[r3].nat)
        register_nat_consumption_fault(itype);

    if (PSR.vm == 1)
        virtualization_fault();

    GR[r1] = tlb_access_key(GR[r3], itype);
    GR[r1].nat = 0;
}
```

Interruptions: Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Virtualization fault

tbit — Test Bit

Format: (*qp*) tbit.*trel*.*ctype* p_1, p_2 = r_3, pos_6 I16

Description: The bit specified by the pos_6 immediate is selected from GR r_3. The selected bit forms a single bit result either complemented or not depending on the *trel* completer. This result is written to the two predicate register destinations p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-15 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

trel	Test Relation	Pseudo-op of	
nz	selected bit == 1	z	$p_1 \leftrightarrow p_2$
z	selected bit == 0		

Table 2-54. Test Bit Relations for Parallel tbits

trel	Test Relation
nz	selected bit == 1
z	selected bit == 0

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is set, or if the compare type is unc.

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (trel == 'nz')                                   // 'nz' - test for 1
        tmp_rel = GR[r3]{pos6};
    else                                                // 'z' - test for 0
        tmp_rel = !GR[r3]{pos6};

    switch (ctype) {
        case 'and':                                     // and-type compare
            if (GR[r3].nat || !tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                                      // or-type compare
            if (!GR[r3].nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                                // or.andcm-type compare
            if (!GR[r3].nat && tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case 'unc':                                     // unc-type compare
        default:                                        // normal compare
            if (GR[r3].nat) {
                PR[p1] = 0;
                PR[p2] = 0;
            } else {
                PR[p1] = tmp_rel;
                PR[p2] = !tmp_rel;
            }
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Interruptions: Illegal Operation fault

tf — Test Feature

Format: (*qp*) tf.*trel.ctype* p_1, p_2 = imm_5 I30

Description: The imm_5 value (in the range of 32-63) selects the feature bit defined in Table 2-57 to be tested from the features vector in CPUID[4]. See Section 3.1.11, "Processor Identification Registers" on page 1:34 for details on CPUID registers. The selected bit forms a single-bit result either complemented or not depending on the *trel* completer. This result is written to the two predicate register destinations p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-15 on page 3:39.

The *trel* completer values .nz and .z indicate non-zero and zero sense of the test. For normal and unc types, only the .z value is directly implemented in hardware; the .nz value is actually a pseudo-op. For it, the assembler simply switches the predicate target specifiers and uses the implemented relation. For the parallel types, both relations are implemented in hardware.

Table 2-55. Test Feature Relations for Normal and unc tf

trel	Test Relation	Pseudo-op of	
nz	selected feature available	z	$p_1 \leftrightarrow p_2$
z	selected feature unavailable		

Table 2-56. Test Feature Relations for Parallel tf

trel	Test Relation
nz	selected feature available
z	selected feature unavailable

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is set or the compare type is unc.

Table 2-57. Test Feature Features Assignment

imm_5	Feature Symbol	Feature
32	@clz	`clz` feature
33	@mpy	`mpy4`, `mpyshl4` feature
34 - 63	*none*	Not currently defined

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    tmp_rel = (psr.vm && pal_vp_env_enabled() && VAC.a_tf) ?
              vcpuid[4]{imm5} : cpuid[4]{imm5};

    if (trel == 'z')                              // 'z' - test for 0, not 1
        tmp_rel = !tmp_rel;

    switch (ctype) {
        case 'and':                               // and-type compare
            if (!tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                                // or-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                          // or.andcm-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case 'unc':                               // unc-type compare
        default:                                  // normal compare
            PR[p1] = tmp_rel;
            PR[p2] = !tmp_rel;
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Interruptions: Illegal Operation fault

thash — Translation Hashed Entry Address

Format: (*qp*) thash $r_1 = r_3$ M46

Description: A Virtual Hashed Page Table (VHPT) entry address is generated based on the specified virtual address and the result is placed in GR r_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(r1);

    if (PSR.vm == 1)
        virtualization_fault();

    if (GR[r3].nat || unimplemented_virtual_address(GR[r3], PSR.vm)) {
        GR[r1] = undefined();
        GR[r1].nat = 1;
    } else {
        tmp_vr = GR[r3]{63:61};
        tmp_va = GR[r3]{60:0};
        GR[r1] = tlb_vhpt_hash(tmp_vr, tmp_va, RR[tmp_vr].rid,
                               RR[tmp_vr].ps);
        GR[r1].nat = 0;
    }
}
```

Interruptions: Illegal Operation fault Virtualization fault

tnat — Test NaT

Format: (*qp*) tnat.*trel.ctype* $p_1, p_2 = r_3$ I17

Description: The NaT bit from GR r_3 forms a single bit result, either complemented or not depending on the *trel* completer. This result is written to the two predicate register destinations, p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-15 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

trel	Test Relation	Pseudo-op of	
nz	selected bit == 1	z	$p_1 \leftrightarrow p_2$
z	selected bit == 0		

Table 2-59. Test NaT Relations for Parallel tnats

trel	Test Relation
nz	selected bit == 1
z	selected bit == 0

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is set, or if the compare type is unc.

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (trel == 'nz')                                   // 'nz' - test for 1
        tmp_rel = GR[r3].nat;
    else                                                // 'z' - test for 0
        tmp_rel = !GR[r3].nat;

    switch (ctype) {
        case 'and':                                     // and-type compare
            if (!tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                                      // or-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                                // or.andcm-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case 'unc':                                     // unc-type compare
        default:                                        // normal compare
            PR[p1] = tmp_rel;
            PR[p2] = !tmp_rel;
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Interruptions: Illegal Operation fault

tpa — Translate to Physical Address

Format: (*qp*) tpa $r_1 = r_3$ M46

Description: The physical address for the virtual address specified by GR r_3 is obtained and placed in GR r_1.

When PSR.dt is 1, the DTLB and the VHPT are searched for the virtual address specified by GR r_3 and the region register indexed by GR r_3 bits {63:61}. If a matching present translation is found the physical address of the translation is placed in GR r_1. If a matching present translation is not found the appropriate TLB fault is taken.

When PSR.dt is 0, only the DTLB is searched, because the VHPT walker is disabled. If no matching present translation is found in the DTLB, an Alternate Data TLB fault is raised if psr.ic is one or a Data Nested TLB fault is raised if psr.ic is zero.

If this instruction faults, then it will set the non-access bit in the ISR. The ISR read and write bits are not set.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS|TPA;
    check_target_register(r1);

    if (PSR.cpl != 0)
        privileged_operation_fault(itype);

    if (GR[r3].nat)
        register_nat_consumption_fault(itype);

    GR[r1] = tlb_translate_nonaccess(GR[r3], itype);
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Privileged 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

ttag — Translation Hashed Entry Tag

Format: (*qp*) ttag $r_1 = r_3$ M46

Description: A tag used for matching during searches of the long format Virtual Hashed Page Table (VHPT) is generated and placed in GR r_1. The virtual address is specified by GR r_3 and the region register selected by GR r_3 bits {63:61}.

If `ttag` is given a NaT input argument or an unimplemented virtual address as an input, the resulting target register value is undefined, and its NaT bit is set to one.

The tag generation function generates an implementation-specific long format VHPT tag. The tag generation function must use all implemented region bits and only virtual address bits {60:0}. PTA.vf is ignored by this instruction.

A translation in the long format VHPT must be uniquely identified by its hash index generated by the `thash` instruction and the tag produced from this instruction.

This instruction must be implemented on all processor models, even processor models that do not implement a VHPT walker.

This instruction can only be executed when PSR.vm is 0.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (PSR.vm == 1)
        virtualization_fault();

    if (GR[r3].nat || unimplemented_virtual_address(GR[r3], PSR.vm)) {
        GR[r1] = undefined();
        GR[r1].nat = 1;
    } else {
        tmp_vr = GR[r3]{63:61};
        tmp_va = GR[r3]{60:0};
        GR[r1] = tlb_vhpt_tag(tmp_va, RR[tmp_vr].rid, RR[tmp_vr].ps);
        GR[r1].nat = 0;
    }
}
```

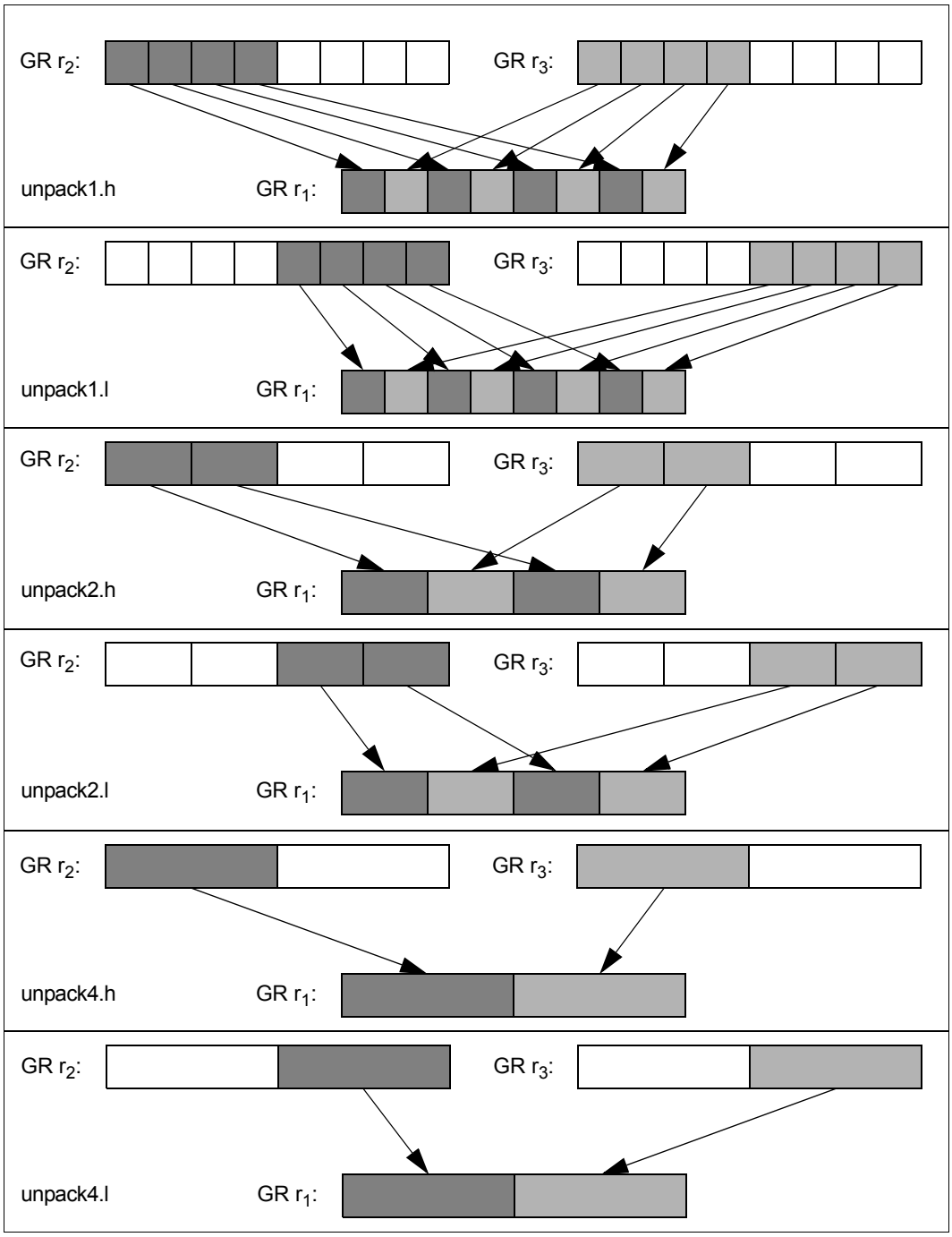
Interruptions: Illegal Operation fault Virtualization fault

unpack — Unpack

Format:	(*qp*) unpack1.h $r_1 = r_2, r_3$	one_byte_form, high_form	I2
	(*qp*) unpack2.h $r_1 = r_2, r_3$	two_byte_form, high_form	I2
	(*qp*) unpack4.h $r_1 = r_2, r_3$	four_byte_form, high_form	I2
	(*qp*) unpack1.l $r_1 = r_2, r_3$	one_byte_form, low_form	I2
	(*qp*) unpack2.l $r_1 = r_2, r_3$	two_byte_form, low_form	I2
	(*qp*) unpack4.l $r_1 = r_2, r_3$	four_byte_form, low_form	I2

Description: The data elements of GR r_2 and r_3 are unpacked, and the result placed in GR r_1. In the high_form, the most significant elements of each source register are selected, while in the low_form the least significant elements of each source register are selected. Elements are selected alternately from the source registers.

Figure 2-45. Unpack Operation



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                              // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (high_form)
            GR[r1] = concatenate8(   x[7], y[7], x[6], y[6],
                                     x[5], y[5], x[4], y[4]);
        else // low_form
            GR[r1] = concatenate8(   x[3], y[3], x[2], y[2],
                                     x[1], y[1], x[0], y[0]);
    } else if (two_byte_form) {                       // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (high_form)
            GR[r1] = concatenate4(x[3], y[3], x[2], y[2]);
        else // low_form
            GR[r1] = concatenate4(x[1], y[1], x[0], y[0]);
    } else {                                          // four-byte elements
        x[0] = GR[r2]{31:0};     y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};    y[1] = GR[r3]{63:32};

        if (high_form)
            GR[r1] = concatenate2(x[1], y[1]);
        else // low_form
            GR[r1] = concatenate2(x[0], y[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

vmsw — Virtual Machine Switch

Format:

vmsw.0	zero_form	B8
vmsw.1	one_form	B8

Description: This instruction sets the PSR.vm bit to the specified value. This instruction can be used to implement transitions to/from virtual machine mode without the overhead of an interruption.

If instruction address translation is enabled and the page containing the `vmsw` instruction has access rights equal to 7, then the new value is written to the PSR.vm bit. In the zero_form, PSR.vm is set to 0, and in the one_form, PSR.vm is set to 1.

Instructions after the `vmsw` instruction in the same instruction group may be executed with the old or new value of PSR.vm. Instructions in subsequent instruction groups will be executed with PSR.vm equal to the new value.

If the above conditions are not met, this instruction takes a Virtualization fault.

This instruction can only be executed at the most privileged level. This instruction cannot be predicated.

Implementation of PSR.vm is optional. If it is not implemented, this instruction takes Illegal Operation fault. If it is implemented but either virtual machine features or the `vmsw` instruction are disabled, this instruction takes Virtualization fault when executed at the most privileged level.

Operation:

```
if (!implemented_vm())
    illegal_operation fault();

if (PSR.cpl != 0)
    privileged_operation_fault(0);

if (!(PSR.it == 1 && itlb_ar() == 7) || vm_disabled() || vmsw_disabled())
    virtualization_fault();

if (zero_form) {
    PSR.vm = 0;
}
else {
    PSR.vm = 1;
}
```

Interruptions: Illegal Operation fault
Privileged Operation fault
Virtualization fault

xchg — Exchange

Format: (*qp*) xchg*sz*.*ldhint* r_1 = [r_3], r_2 M16

Description: A value consisting of *sz* bytes is read from memory starting at the address specified by the value in GR r_3. The least significant *sz* bytes of the value in GR r_2 are written to memory starting at the address specified by the value in GR r_3. The value read from memory is then zero extended and placed in GR r_1 and the NaT bit corresponding to GR r_1 is cleared. The values of the *sz* completer are given in Table 2-60.

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

sz Completer	Bytes Accessed
1	1 byte
2	2 bytes
4	4 bytes
8	8 bytes

The exchange is performed with acquire semantics, i.e., the memory read/write is made visible prior to all subsequent data memory accesses. See 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[qp]) {
    check_target_register(r1);

    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(SEMAPHORE);

    paddr = tlb_translate(GR[r3], sz, SEMAPHORE, PSR.cpl, &mattr,
                          &tmp_unused);

    if (!ma_supports_semaphores(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

    val = mem_xchg(GR[r2], paddr, sz, UM.be, mattr, ACQUIRE, ldhint);

    alat_inval_multiple_entries(paddr, sz);

    GR[r1] = zero_ext(val, sz * 8);
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

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: f1 as an operand is not an integer 1; it is just the register file format's 1.0 value.

In all forms, if any of FR f_3 , FR f_4 , or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
    } else {
        if (low_form || high_form)
            tmp_res_128 =
                fp_I64_x_I64_to_I128(FR[f3].significand, FR[f4].significand);
        else // high_unsigned_form
            tmp_res_128 =
                fp_U64_x_U64_to_U128(FR[f3].significand, FR[f4].significand);

        tmp_res_128 =
            fp_U128_add(tmp_res_128, fp_U64_to_U128(FR[f2].significand));

        if (high_form || high_unsigned_form)
            FR[f1].significand = tmp_res_128.hi;
        else // low_form
            FR[f1].significand = tmp_res_128.lo;

        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

Interruptions: Disabled Floating-point Register fault

xmpy — Fixed-Point Multiply

Format:

(*qp*) xmpy.l $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.l $f_1 = f_3, f_4$, f0
(*qp*) xmpy.lu $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.l $f_1 = f_3, f_4$, f0
(*qp*) xmpy.h $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.h $f_1 = f_3, f_4$, f0
(*qp*) xmpy.hu $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.hu $f_1 = f_3, f_4$, f0

Description: Two source operands (FR f_3 and FR f_4) are treated as either signed or unsigned integers and multiplied. The upper or lower 64 bits of the resultant product are selected and placed in FR f_1.

In the high_unsigned_form, the significand fields of FR f_3 and FR f_4 are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The most significant 64-bits of the resultant product are placed in the significand field of FR f_1.

In the high_form, the significand fields of FR f_3 and FR f_4 are treated as signed integers and multiplied to produce a full 128-bit signed result. The most significant 64-bits of the resultant product are placed in the significand field of FR f_1.

In the other forms, the significand fields of FR f_3 and FR f_4 are treated as signed integers and multiplied to produce a full 128-bit signed result. The least significant 64-bits of the resultant product are placed in the significand field of FR f_1.

In all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0). Note: f1 as an operand is not an integer 1; it is just the register file format's 1.0 value.

Operation: See "xma — Fixed-Point Multiply Add" on page 3:276.

xor — Exclusive Or

Format:

(*qp*) xor $r_1 = r_2, r_3$	register_form	A1
(*qp*) xor $r_1 = imm_8, r_3$	imm8_form	A3

Description: The two source operands are logically XORed and the result placed in GR r_1. In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the imm_8 encoding field.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src ^ GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

zxt — Zero Extend

Format: (*qp*) zxt*xsz* $r_1 = r_3$ I29

Description: The value in GR r_3 is zero extended above the bit position specified by *xsz* and the result is placed in GR r_1. The mnemonic values for *xsz* are given in Table 2-52 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

§

Pseudo-Code Functions 3

This chapter contains a table of all pseudo-code functions used on the Itanium instruction pages.

Table 3-1. Pseudo-code Functions

Function	Operation
*xxx*_fault(parameters ...)	There are several fault functions. Each fault function accepts parameters specific to the fault, e.g., exception code values, virtual addresses, etc. If the fault is deferred for speculative load exceptions the fault function will return with a deferral indication. Otherwise, fault routines do not return and terminate the instruction sequence.
*xxx*_trap(parameters ...)	There are several trap functions. Each trap function accepts parameters specific to the trap, e.g., trap code values, virtual addresses, etc. Trap routines do not return.
acceptance_fence()	Ensures prior data memory references to uncached ordered-sequential memory pages are “accepted” before subsequent data memory references are performed by the processor.
alat_cmp(rtype, raddr)	Returns a one if the implementation finds an ALAT entry which matches the register type specified by `rtype` and the register address specified by `raddr`, else returns zero. This function is implementation specific. Note that an implementation may optionally choose to return zero (indicating no match) even if a matching entry exists in the ALAT. This provides implementation flexibility in designing fast ALAT lookup circuits.
alat_frame_update(delta_bof, delta_sof)	Notifies the ALAT of a change in the bottom of frame and/or size of frame. This allows management of the ALAT’s tag bits or other management functions it might need.
alat_inval()	Invalidate all entries in the ALAT.
alat_inval_multiple_entries(paddr, size)	The ALAT is queried using the physical memory address specified by `paddr` and the access size specified by `size`. All matching ALAT entries are invalidated. No value is returned.
alat_inval_single_entry(rtype, rega)	The ALAT is queried using the register type specified by `rtype` and the register address specified by `rega`. At most one matching ALAT entry is invalidated. No value is returned.
alat_read_memory_on_hit(ldtype, rtype, raddr)	Returns a one if the implementation requires that the requested check load should perform a memory access (requires prior address translation); returns a zero otherwise.
alat_translate_address_on_hit(ldtype, rtype, raddr)	Returns a one if the implementation requires that the requested check load should translate the source address and take associated faults; returns a zero otherwise.
alat_write(ldtype, rtype, raddr, paddr, size)	Allocates a new ALAT entry or updates an existing entry using the load type specified by `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 `raddr`. 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`.
align_to_size_boundary(vaddr, size)	Returns `vaddr` aligned to the boundary specified by `size`.
branch_predict(wh, ih, ret, target, tag)	Implementation-dependent routine which updates the processor’s branch prediction structures.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
check_branch_implemented(check_type)	Implementation-dependent routine which returns TRUE or FALSE, depending on whether a failing check instruction causes a branch (TRUE), or a Speculative Operation fault (FALSE). The result may be different for different types of check instructions: CHKS_GENERAL, CHKS_FLOAT, CHKA_GENERAL, CHKA_FLOAT. In addition, the result may depend on other implementation-dependent parameters.
check_probe_virtualization_fault(type, cpl)	If implemented, this function may raise virtualization faults for specific probe instructions. Please refer to the instruction page for probe instruction for details.
check_target_register(r1)	If the `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.
check_target_register_sof(r1, newsof)	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.
concatenate2(x1, x2)	Concatenates the lower 32 bits of the 2 arguments, and returns the 64-bit result.
concatenate4(x1, x2, x3, x4)	Concatenates the lower 16 bits of the 4 arguments, and returns the 64-bit result.
concatenate8(x1, x2, x3, x4, x5, x6, x7, x8)	Concatenates the lower 8 bits of the 8 arguments, and returns the 64-bit result.
data_serialize()	Ensures all prior register updates with side-effects are observed before subsequent execution and data memory references are performed.
deliver_unmasked_pending_interrupt()	This implementation-specific function checks whether any unmasked external interrupts are pending, and if so, transfers control to the external interrupt vector.
execute_hint(hint)	Executes the hint specified by `hint`.
fadd(fp_dp, fr2)	Adds a floating-point register value to the infinitely precise product and return the infinitely precise sum, ready for rounding.
fcmp_exception_fault_check(f2, f3, frel, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fcmp` instruction.
fcvt_fx_exception_fault_check(fr2, signed_form, trunc_form, sf *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fcvt.fx`, `fcvt.fxu`, `fcvt.fx.trunc` and `fcvt.fxu.trunc` instructions. It propagates NaNs.
fma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fma` instruction. It propagates NaNs and special IEEE results.
fminmax_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `famax`, `famin`, `fmax`, and `fmin` instructions.
fms_fnma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fms` and `fnma` instructions. It propagates NaNs and special IEEE results.
fmul(fr3, fr4)	Performs an infinitely precise multiply of two floating-point register values.
followed_by_stop()	Returns TRUE if the current instruction is followed by a stop; otherwise, returns FALSE.
fp_check_target_register(f1)	If the specified floating-point register identifier is 0 or 1, this function causes an illegal operation fault.
fp_decode_fault(tmp_fp_env)	Returns floating-point exception fault code values for ISR.code.
fp_decode_traps(tmp_fp_env)	Returns floating-point trap code values for ISR.code.
fp_equal(fr1, fr2)	IEEE standard equality relationship test.
fp_fr_to_mem_format(freg, size)	Converts a floating-point value in register format to floating-point memory format. It assumes that the floating-point value in the register has been previously rounded to the correct precision which corresponds with the `size` parameter.
fp_ieee_recip(num, den)	Returns the true quotient for special sets of operands, or an approximation to the reciprocal of the divisor to be used in the software divide algorithm.
fp_ieee_recip_sqrt(root)	Returns the true square root result for special operands, or an approximation to the reciprocal square root to be used in the software square root algorithm.
fp_is_nan(freg)	Returns true when floating register contains a NaN.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
fp_is_nan_or_inf(freg)	Returns true if the floating-point exception_fault_check functions returned a IEEE fault disabled default result or a propagated NaN.
fp_is_natval(freg)	Returns true when floating register contains a NaTVal
fp_is_normal(freg)	Returns true when floating register contains a normal number.
fp_is_pos_inf(freg)	Returns true when floating register contains a positive infinity.
fp_is_qnan(freg)	Returns true when floating register contains a quiet NaN.
fp_is_snan(freg)	Returns true when floating register contains a signalling NaN.
fp_is_unorm(freg)	Returns true when floating register contains an unnormalized number.
fp_is_unsupported(freg)	Returns true when floating register contains an unsupported format.
fp_less_than(fr1, fr2)	IEEE standard less-than relationship test.
fp_lesser_or_equal(fr1, fr2)	IEEE standard less-than or equal-to relationship test
fp_mem_to_fr_format(mem, size)	Converts a floating-point value in memory format to floating-point register format.
fp_normalize(fr1)	Normalizes an unnormalized fp value. This function flushes to zero any unnormal values which can not be represented in the register file
fp_raise_fault(tmp_fp_env)	Checks the local instruction state for any faulting conditions which require an interruption to be raised.
fp_raise_traps(tmp_fp_env)	Checks the local instruction state for any trapping conditions which require an interruption to be raised.
fp_reg_bank_conflict(f1, f2)	Returns true if the two specified FRs are in the same bank.
fp_reg_disabled(f1, f2, f3, f4)	Check for possible disabled floating-point register faults.
fp_reg_read(freg)	Reads the FR and gives canonical double-extended denormals (and pseudo-denormals) their true mathematical exponent. Other classes of operands are unaltered.
fp_unordered(fr1, fr2)	IEEE standard unordered relationship
fp_update_fpsr(sf, tmp_fp_env)	Copies a floating-point instruction's local state into the global FPSR.
fp_update_psr(dest_freg)	Conditionally sets PSR.mfl or PSR.mfh based on dest_freg.
fpcmp_exception_fault_check(f2, f3, frel, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpcmp` instruction.
fpcvt_exception_fault_check(f2, signed_form, trunc_form, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpcvt.fx`, `fpcvt.fxu`, `fpcvt.fx.trunc`, and `fpcvt.fxu.trunc` instructions. It propagates NaNs.
fpma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpma` instruction. It propagates NaNs and special IEEE results.
fpminmax_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpmin`, `fpmax`, `fpamin` and `fpamax` instructions.
fpms_fpnma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpms` and `fpnma` instructions. It propagates NaNs and special IEEE results.
fprcpa_exception_fault_check(f2, f3, sf, *tmp_fp_env, *limits_check)	Checks for all floating-point faulting conditions for the `fprcpa` instruction. It propagates NaNs and special IEEE results. It also indicates operand limit violations.
fprsqrta_exception_fault_check(f3, sf, *tmp_fp_env, *limits_check)	Checks for all floating-point faulting conditions for the `fprsqrta` instruction. It propagates NaNs and special IEEE results. It also indicates operand limit violations.
frcpa_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `frcpa` instruction. It propagates NaNs and special IEEE results.
frsqrta_exception_fault_check(f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `frsqrta` instruction. It propagates NaNs and special IEEE results
ignored_field_mask(regclass, reg, value)	Boolean function that returns value with bits cleared to 0 corresponding to ignored bits for the specified register and register type.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
impl_check_mov_itir()	Implementation-specific function that returns TRUE if ITIR is checked for reserved fields and encodings on a mov to ITIR instruction.
impl_check_mov_psr_l(gr)	Implementation-specific function to check bits {63:32} of `gr` corresponding to reserved fields of the PSR for Reserved Register/Field fault.
impl_check_tlb_itir()	Implementation-specific function that returns TRUE if all fields of ITIR are checked for reserved encodings on a TLB insert instruction regardless of whether the translation is present.
impl_gitc_enable()	Implementation-specific function that indicates whether guest MOV-from-AR.ITC optimization is enabled.
impl_ia32_ar_reserved_ignored(ar3)	Implementation-specific function which indicates how the reserved and ignored fields in the specified IA-32 application register, `ar3`, behave. If it returns FALSE, the reserved and/or ignored bits in the specified application register can be written, and when read they return the value most-recently written. If it returns TRUE, attempts to write a non-zero value to a reserved field in the specified application register cause a Reserved Register/Field fault, and reads return 0; writing to an ignored field in the specified application register is ignored, and reads return the constant value defined for that field.
impl_iib()	Implementation-specific function which indicates whether Interruption Instruction Bundle registers (IIB0-1) are implemented.
impl_itir_cwi_mask()	Implementation-specific function that either returns the value passed to it or the value passed to it masked with zeros in bit positions {63:32} and/or {1:0}.
impl_ito()	Implementation-specific function which indicates whether Interval Timer Offset (ITO) register is implemented.
impl_probe_intercept()	Implementation-specific function indicates whether probe interceptions are supported.
impl_ruc()	Implementation-specific function which indicates whether Resource Utilization Counter (RUC) application register is implemented.
impl_uia_fault_supported()	Implementation-specific function that either returns TRUE if the processor reports unimplemented instruction addresses with an Unimplemented Instruction Address fault, and returns FALSE if the processor reports them with an Unimplemented Instruction Address trap.
implemented_vm()	Returns TRUE if the processor implements the PSR.vm bit (regardless of whether virtual machine features are enabled or disabled).
instruction_implemented(inst)	Implementation-dependent routine which returns TRUE or FALSE, depending on whether `inst` is implemented.
instruction_serialize()	Ensures all prior register updates with side-effects are observed before subsequent instruction and data memory references are performed. Also ensures prior SYNC.i operations have been observed by the instruction cache.
instruction_synchronize()	Synchronizes the instruction and data stream for Flush Cache operations. This function ensures that when prior Flush Cache operations are observed by the local data cache they are observed by the local instruction cache, and when prior Flush Cache operations are observed by another processor's data cache they are observed within the same processor's instruction cache.
is_finite(freg)	Returns true when floating register contains a finite number.
is_ignored_reg(regnum)	Boolean function that returns true if `regnum` is an ignored application register, otherwise false.
is_inf(freg)	Returns true when floating register contains an infinite number.
is_interruption_cr(regnum)	Boolean function that returns true if `regnum` is one of the Interruption Control registers (see Section 3.3.5, "Interruption Control Registers" on page 2:36), otherwise false.
is_kernel_reg(ar_addr)	Returns a one if `ar_addr` is the address of a kernel register application register

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
is_read_only_reg(rtype, raddr)	Returns a one if the register addressed by 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)

Function	Operation
mem_xchg_add(add_val, paddr, size, byte_order, mattr, otype, hint)	Returns `size` bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. The least significant `size` bytes of the sum of the value read from memory and `add_val` is then written to `size` bytes in memory starting at the physical address specified by `paddr`. The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
mem_xchg_cond(cmp_val, data, paddr, size, byte_order, mattr, otype, hint)	Returns `size` bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. If the value read from memory is equal to `cmp_val`, then the least significant `size` bytes of data are written to `size bytes` in memory starting at the physical address specified by `paddr`. If the write is performed, the read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
mem_xchg16_cond(cmp_val, gr_data, ar_data, paddr, byte_order, mattr, otype, hint)	Returns 8 bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. If the value read from memory is equal to `cmp_val`, then the 8 bytes of `gr_data` are written to 8 bytes in memory starting at the physical address specified by (`paddr` & ~0x8), and the 8 bytes of `ar_data` are written to 8 bytes in memory starting at the physical address specified by ((`paddr` & ~0x8) + 8). If the write is performed, the read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by mattr and the byte ordering in memory is specified by `byte_order`. The byte ordering only affects the ordering of bytes within each of the 8-byte values stored. `otype` specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
ordering_fence()	Ensures prior data memory references are made visible before future data memory references are made visible by the processor.
partially_implemented_ip()	Implementation-dependent routine which returns TRUE if the implementation, on an Unimplemented Instruction Address trap, writes IIP with the sign-extended virtual address or zero-extended physical address for what would have been the next value of IP. Returns FALSE if the implementation, on this trap, simply writes IIP with the full address which would have been the next value of IP.
pending_virtual_interrupt()	Check for unmasked pending virtual interrupt.
pr_phys_to_virt(phys_id)	Returns the virtual register id of the predicate from the physical register id, `phys_id` of the predicate.
rotate_regs()	Decrements the Register Rename Base registers, effectively rotating the register files. CFM.rrb.gr is decremented only if CFM.sor is non-zero.
rse_enable_current_frame_load()	If the RSE load pointer (RSE.BSPLoad) is greater than AR[BSP], the `RSE.CFLE` bit is set to indicate that mandatory RSE loads are allowed to restore registers in the current frame (in no other case does the RSE spill or fill registers in the current frame). This function does not perform mandatory RSE loads. This procedure does not cause any interruptions.
rse_ensure_regs_loaded(number_of_bytes)	All registers and NaT collections between `AR[BSP]` and `(AR[BSP]-number_of_bytes)` which are not already in stacked registers are loaded into the register stack with mandatory RSE loads. If the number of registers to be loaded is greater than `RSE.N_STACK_PHYS` an Illegal Operation fault is raised. All registers starting with backing store address (AR[BSP] - 8) and decrementing down to and including backing store address (AR[BSP] - number_of_bytes) are made part of the dirty partition. With exception of the current frame, all other stacked registers are made part of the invalid partition. Note that `number_of_bytes` may be zero. The resulting sequence of RSE loads may be interrupted. Mandatory RSE loads may cause an interruption; see Table 6-6, "RSE Interruption Summary" on page 6-145.
rse_invalidate_non_current_regs()	All registers outside the current frame are invalidated.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
rse_load(type)	Restores a register or NaT collection from the backing store (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.
rse_new_frame(current_frame_size, new_frame_size)	A new frame is defined without changing any register renaming. The new frame size is completely defined by the new_frame_size parameter (successive calls are not cumulative). If new_frame_size is larger than current_frame_size and the number of registers in the invalid and clean partitions is less than the size of frame growth then mandatory RSE stores are issued until enough registers are available. The resulting sequence of RSE stores may be interrupted. Mandatory RSE stores may cause interruptions; see Table 6-6, “RSE Interruption Summary” on page 6-145.
rse_preserve_frame(preserved_frame_size)	The number of registers specified by preserved_frame_size are marked to be preserved by the RSE. Register renaming causes the preserved_frame_size registers after GR[32] to be renamed to GR[32]. AR[BSP] is updated to contain the backing store address where the new GR[32] will be stored.
rse_restore_frame(preserved_sol, growth, current_frame_size)	The first two parameters define how the current frame is about to be updated by a branch return or rfi: preserved_sol defines how many registers need to be restored below RSE.BOF; growth defines by how many registers the top of the current frame will grow (growth will generally be negative). The number of registers specified by preserved_sol are marked to be restored. Register renaming causes the preserved_sol registers before GR[32] to be renamed to GR[32]. AR[BSP] is updated to contain the backing store address where the new GR[32] will be stored. If the number of dirty and clean registers is less than preserved_sol then mandatory RSE loads must be issued before the new current frame is considered valid. This function does not perform mandatory RSE loads. This function returns TRUE if the preserved frame grows beyond the invalid and clean regions into the dirty region. In this case the third argument, current_frame_size, is used to force the returned to frame to zero (see Section 6.5.5, “Bad PFS used by Branch Return” on page 2:143).
rse_store(type)	Saves a register or NaT collection to the backing store (store_address = AR[BSPSTORE]). If store_address{8:3} is equal to 0x3f then the NaT collection AR[RNAT] is stored. If store_address{8:3} is not equal to 0x3f then the register RSE.StoreReg is stored and the NaT bit from that register is deposited in AR[RNAT]{store_address{8:3}}. If the store is successful AR[BSPSTORE] is incremented by 8. If the store is successful and a register was stored RSE.StoreReg is incremented by 1 (possibly wrapping in the stacked registers). This store moves a register from the dirty partition to the clean partition. For mandatory RSE stores, type is MANDATORY. Mandatory RSE stores may cause interruptions. See Table 6-6, “RSE Interruption Summary” on page 6-145.
rse_update_internal_stack_pointers(new_store_pointer)	Given a new value for AR[BSPSTORE] (new_store_pointer) this function computes the new value for AR[BSP]. This value is equal to new_store_pointer plus the number of dirty registers plus the number of intervening NaT collections. This means that the size of the dirty partition is the same before and after a write to AR[BSPSTORE]. All clean registers are moved to the invalid partition.
sign_ext(value, pos)	Returns a 64 bit number with bits pos-1 through 0 taken from value and bit pos-1 of value replicated in bit positions pos through 63. If pos is greater than or equal to 64, value is returned.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
spontaneous_deferral(paddr, size, border, mattr, otype, hint, *defer)	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).
spontaneous_deferral_enabled()	Implementation-dependent routine which returns TRUE or FALSE, depending on whether spontaneous deferral of speculative loads is enabled or disabled in the processor.
tlb_access_key(vaddr, itype)	This function returns, in bits 31:8, the access key from the TLB for the entry corresponding to `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.
tlb_broadcast_purge(rid, vaddr, size, type)	Sends a broadcast purge DTC and ITC transaction to other processors in the multiprocessor coherency domain, where the region identifier (`rid`), virtual address (`vaddr`) and page size (`size`) specify the translation entry to purge. The operation waits until all processors that receive the purge have completed the purge operation. The purge type (`type`) specifies whether the ALAT on other processors should also be purged in conjunction with the TC.
tlb_enter_privileged_code()	This function determines the new privilege level for epc from the TLB entry for the page containing this instruction. If the page containing the epc instruction has execute-only page access rights and the privilege level assigned to the page is higher than (numerically less than) the current privilege level, then the current privilege level is set to the privilege level field in the translation for the page containing the epc instruction.
tlb_grant_permission(vaddr, type, pl)	Returns a boolean indicating if read, write access is granted for the specified virtual memory address (`vaddr`) and privilege level (`pl`). The access type (`type`) specifies either read or write. The following faults are checked:: • 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)

Function	Operation
tlb_may_purge_itc_entries(rid, vaddr, size)	May locally purge ITC entries that match the specified virtual address (vaddr), region identifier (rid) and page size (size). May also invalidate entries that partially overlap the parameters. The extent of purging is implementation dependent. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_must_purge_dtc_entries(rid, vaddr, size)	Purges all local, possibly overlapping, DTC entries matching the specified region identifier (rid), virtual address (vaddr) and page size (size). vaddr{63:61} (VRN) is ignored in the purge, i.e all entries that match vaddr{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache. If the specified purge values overlap with an existing DTR translation, an implementation may generate a machine check abort.
tlb_must_purge_dtr_entries(rid, vaddr, size)	Purges all local, possibly overlapping, DTR entries matching the specified region identifier (rid), virtual address (vaddr) and page size (size). vaddr{63:61} (VRN) is ignored in the purge, i.e all entries that match vaddr{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_must_purge_itc_entries(rid, vaddr, size)	Purges all local, possibly overlapping, ITC entry matching the specified region identifier (rid), virtual address (vaddr) and page size (size). vaddr{63:61} (VRN) is ignored in the purge, i.e all entries that match vaddr{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache. If the specified purge values overlap with an existing ITR translation, an implementation may generate a machine check abort.
tlb_must_purge_itr_entries(rid, vaddr, size)	Purges all local, possibly overlapping, ITR entry matching the specified region identifier (rid), virtual address (vaddr) and page size (size). vaddr{63:61} (VRN) is ignored in the purge, i.e all entries that match vaddr{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_purge_translation_cache(loop)	Removes 1 to N translations from the local processor's ITC and DTC. The number of entries removed is implementation specific. The parameter loop is used to generate an implementation-specific purge parameter.
tlb_replacement_algorithm(tlb)	Returns the next ITC or DTC slot number to replace. Replacement algorithms are implementation specific. tlb specifies to perform the algorithm on the ITC or DTC.
tlb_search_pkr(key)	Searches for a valid protection key register with a matching protection key. The search algorithm is implementation specific. Returns the PKR register slot number if found, otherwise returns Not Found.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
tlb_translate(vaddr, size, type, cpl, *attr, *defer)	Returns the translated data physical address for the specified virtual memory address (`vaddr`) when translation enabled; otherwise, returns `vaddr`. `size` specifies the size of the access, `type` specifies the type of access (e.g., read, write, advance, spec). `cpl` specifies the privilege level for access checking purposes. `*attr` returns the mapped physical memory attribute. If any fault conditions are detected and deferred, tlb_translate returns with `*defer` set. If a fault is generated but the fault is not deferred, tlb_translate does not return. tlb_translate checks the following faults: • 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
tlb_translate_nonaccess(vaddr, type)	Returns the translated data physical address for the specified virtual memory address (`vaddr`). `type` specifies the type of access (e.g., `FC`, `TPA`). If a fault is generated, tlb_translate_nonaccess does not return. The following faults are checked: • Unimplemented Data Address fault • Virtualization fault (`tpa` only) • Data Nested TLB fault • Alternate Data TLB fault • VHPT Data fault • Data TLB fault • Data Page Not Present fault • Data NaT Page Consumption fault • Data Access Rights fault (`fc` only)
tlb_vhpt_hash(vrn, vaddr61, rid, size)	Generates a VHPT entry address for the specified virtual region number (`vrn`) and 61-bit virtual offset (`vaddr61`), region identifier (`rid`) and page size (`size`). Tlb_vhpt_hash hashes `vaddr`, `rid` and `size` parameters to produce a hash index. The hash index is then masked based on PTA.size and concatenated with PTA.base to generate the VHPT entry address. The long format hash is implementation specific.
tlb_vhpt_tag(vaddr, rid, size)	Generates a VHPT tag identifier for the specified virtual address (`vaddr`), region identifier (`rid`) and page size (`size`). Tlb_vhpt_tag hashes the `vaddr`, rid and `size` parameters to produce translation identifier. The tag in conjunction with the hash index is used to uniquely identify translations in the VHPT. Tag generation is implementation specific. All processor models tag function must guarantee that bit 63 of the generated tag is zero (ti bit).
undefined()	Returns an undefined 64-bit value.
undefined_behavior()	Causes undefined processor behavior. Extent of undefined behavior is described in Section 3.5, "Undefined Behavior" on page 1:44.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
unimplemented_physical_address(paddr)	Return TRUE if the presented physical address is unimplemented on this processor model; FALSE otherwise. This function is model specific.
unimplemented_virtual_address(vaddr, vm)	Return TRUE if the presented virtual address is unimplemented on this processor model; FALSE otherwise. If vm is 1, one additional bit of virtual address is treated as unimplemented. This function is model specific.
vm_all_probes()	Returns TRUE if the processor is configured to virtualize all probe instructions when PSR.vm is 1. See Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344 for details.
vm_disabled()	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.
vm_select_probes()	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.
vmsw_disabled()	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.
zero_ext(value, pos)	Returns a 64 bit unsigned number with bits `pos`-1 through 0 taken from `value` and zeroes in bit positions `pos` through 63. If `pos` is greater than or equal to 64, `value` is returned.

§

Instruction Formats 4

Each Itanium instruction is categorized into one of six types; each instruction type may be executed on one or more execution unit types. Table 4-1 lists the instruction types and the execution unit type on which they are executed:

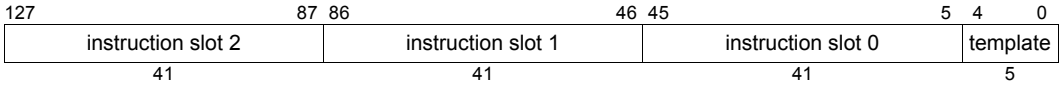
Table 4-1. Relationship between Instruction Type and Execution Unit Type

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit/B-unit[a]

a. L+X Major Opcodes 0 - 7 execute on an I-unit. L+X Major Opcodes 8 - F execute on a B-unit.

Three instructions are grouped together into 128-bit sized and aligned containers called **bundles**. Each bundle contains three 41-bit **instruction slots** and a 5-bit template field. The format of a bundle is depicted in Figure 4-1.

Figure 4-1. Bundle Format



The template field specifies two properties: stops within the current bundle, and the mapping of instruction slots to execution unit types. Not all combinations of these two properties are allowed - Table 4-2 indicates the defined combinations. The three rightmost columns correspond to the three instruction slots in a bundle; listed within each column is the execution unit type controlled by that instruction slot for each encoding of the template field. A double line to the right of an instruction slot indicates that a stop occurs at that point within the current bundle. See "Instruction Encoding Overview" on page 1:38 for the definition of a stop. Within a bundle, execution order proceeds from slot 0 to slot 2. Unused template values (appearing as empty rows in Table 4-2) are reserved and cause an Illegal Operation fault.

Extended instructions, used for long immediate integer and long branch instructions, occupy two instruction slots. Depending on the major opcode, extended instructions execute on a B-unit (long branch/call) or an I-unit (all other L+X instructions).

Table 4-2. Template Field Encoding and Instruction Slot Mapping

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	L-unit	X-unit[a]
05	M-unit	L-unit	X-unit[a]
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

a. The MLX template was formerly called MLI, and for compatibility, the X slot may encode break.i and nop.i in addition to any X-unit instruction.

4.1 Format Summary

All instructions in the instruction set are 41 bits in length. The leftmost 4 bits (40:37) of each instruction are the major opcode. Table 4-3 shows the major opcode assignments for each of the 5 instruction types — ALU (A), Integer (I), Memory (M), Floating-point (F), and Branch (B). Bundle template bits are used to distinguish among the 4 columns, so the same major op values can be reused in each column.

Unused major ops (appearing as blank entries in Table 4-3) behave in one of four ways:

- Ignored major ops (white entries in Table 4-3) execute as `nop` instructions.

- Reserved major ops (light gray in the gray scale version of Table 4-3, brown in the color version) cause an Illegal Operation fault.
- Reserved if PR[qp] is 1 major ops (dark gray in the gray scale version of Table 4-3, purple in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0.
- Reserved if PR[qp] is 1 B-unit major ops (medium gray in the gray scale version of Table 4-3, cyan in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0. These differ from the Reserved if PR[qp] is 1 major ops (purple) only in their RAW dependency behavior (see "RAW Dependency Table" on page 3:374).

Table 4-3. Major Opcode Assignments

Major Op (Bits 40:37)	Instruction Type				
	I/A	M/A	F	B	L+X
0	Misc 0	Sys/Mem Mgmt 0	FP Misc 0	Misc/Indirect Branch 0	Misc 0
1	1	Sys/Mem Mgmt 1	FP Misc 1	Indirect Call 1	1
2	2	2	2	Indirect Predict/Nop 2	2
3	3	3	3	3	3
4	Deposit 4	Int Ld +Reg/getf 4	FP Compare 4	IP-relative Branch 4	4
5	Shift/Test Bit 5	Int Ld/St +Imm 5	FP Class 5	IP-rel Call 5	5
6	6	FP Ld/St +Reg/setf 6	6	6	movl 6
7	MM Mpy/Shift 7	FP Ld/St +Imm 7	7	IP-relative Predict 7	7
8	ALU/MM ALU 8	ALU/MM ALU 8	fma 8	e 8	8
9	Add Imm_{22} 9	Add Imm_{22} 9	fma 9	e 9	9
A	A	A	fms A	e A	A
B	B	B	fms B	e B	B
C	Compare C	Compare C	fnma C	e C	Long Branch C
D	Compare D	Compare D	fnma D	e D	Long Call D
E	Compare E	Compare E	fselect/xma E	e E	E
F	F	F	F	e F	F

Table 4-4 on page 3:296 summarizes all the instruction formats. The instruction fields are color-coded for ease of identification, as described in Table 4-5 on page 3:298. A color version of this chapter is available for those heavily involved in working with the instruction encodings.

The instruction field names, used throughout this chapter, are described in Table 4-6 on page 3:298. The set of special notations (such as whether an instruction is privileged) are listed in Table 4-7 on page 3:299. These notations appear in the "Instruction" column of the opcode tables.

Most instruction containing immediates encode those immediates in more than one instruction field. For example, the 14-bit immediate in the Add Imm_{14} instruction (format A4) is formed from the imm_{7b}, imm_{6d}, 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

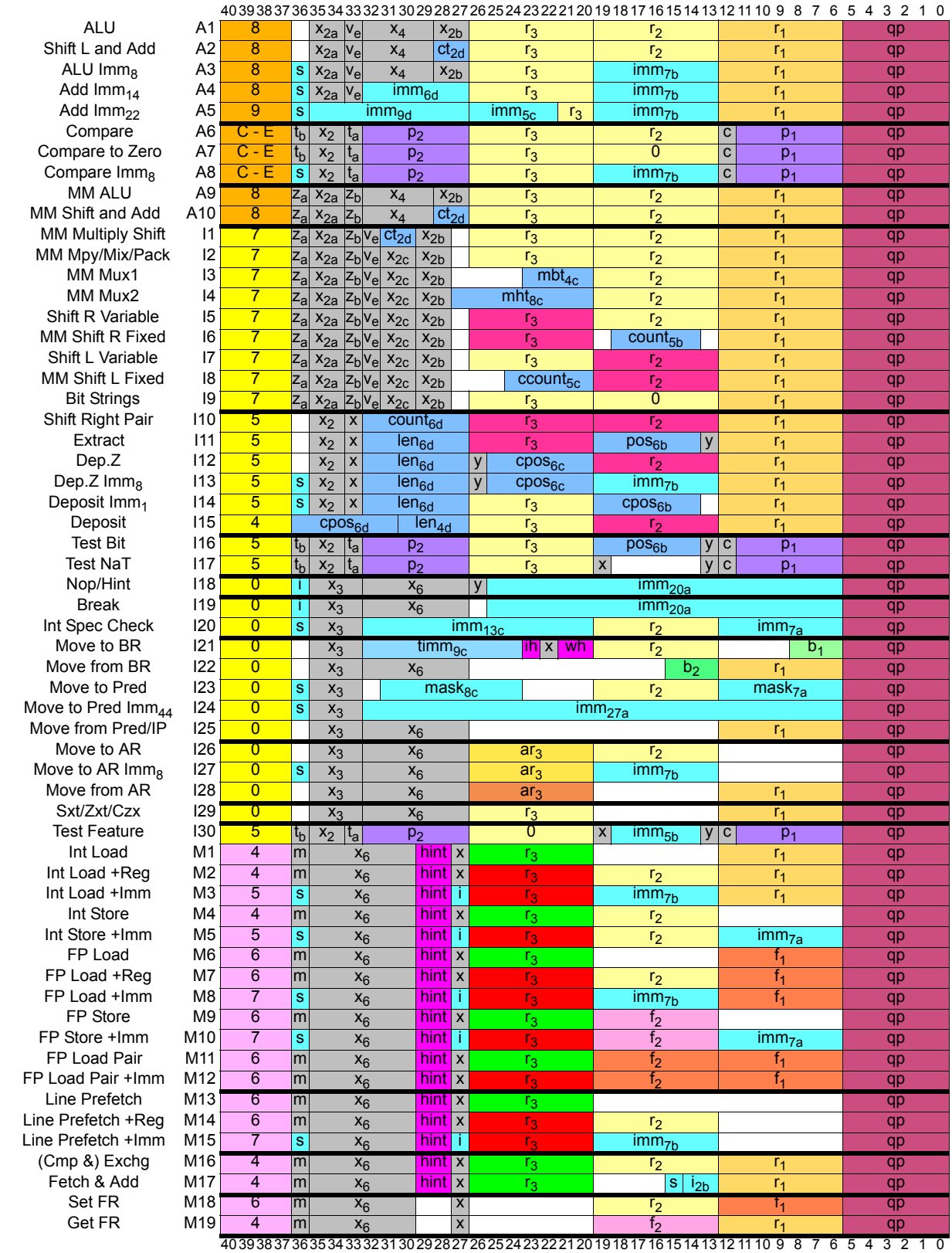


Table 4-4. Instruction Format Summary (Continued)

		40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Int Spec Check	M20	1				s		x_3											imm_{13c}						r_2						imm_{7a}						qp						
FP Spec Check	M21	1				s		x_3											imm_{13c}						f_2						imm_{7a}						qp						
Int ALAT Check	M22	0				s		x_3											imm_{20b}												r_1						qp						
FP ALAT Check	M23	0				s		x_3											imm_{20b}												f_1						qp						
Sync/Srlz/ALAT	M24	0						x_3		x_2			x_4																								qp						
RSE Control	M25	0						x_3		x_2			x_4																								0						
Int ALAT Inval	M26	0						x_3		x_2			x_4																		r_1						qp						
FP ALAT Inval	M27	0						x_3		x_2			x_4																		f_1						qp						
Flush Cache	M28	1		x				x_3				x_6				r_3																					qp						
Move to AR	M29	1						x_3				x_6				ar_3			r_2																		qp						
Move to AR Imm_8	M30	0		s				x_3		x_2			x_4			ar_3			imm_{7b}																		qp						
Move from AR	M31	1						x_3				x_6				ar_3															r_1						qp						
Move to CR	M32	1						x_3				x_6				cr_3			r_2																		qp						
Move from CR	M33	1						x_3				x_6				cr_3															r_1						qp						
Alloc	M34	1						x_3					sor			sol			sof											r_1						qp						
Move to PSR	M35	1						x_3				x_6							r_2																		qp						
Move from PSR	M36	1						x_3				x_6																			r_1						qp						
Break	M37	0		i				x_3		x_2			x_4																								qp						
Probe	M38	1						x_3				x_6				r_3			r_2												r_1						qp						
Probe Imm_2	M39	1						x_3				x_6				r_3													i_{2b}		r_1						qp						
Probe Fault Imm_2	M40	1						x_3				x_6				r_3													i_{2b}								qp						
TC Insert	M41	1						x_3				x_6							r_2																		qp						
Mv to Ind/TR Ins	M42	1						x_3				x_6				r_3			r_2																		qp						
Mv from Ind	M43	1						x_3				x_6				r_3															r_1						qp						
Set/Reset Mask	M44	0		i				x_3		i_{2d}			x_4			imm_{21a}																					qp						
Translation Purge	M45	1						x_3				x_6				r_3			r_2																		qp						
Translation Access	M46	1						x_3				x_6				r_3															r_1						qp						
TC Entry Purge	M47	1						x_3				x_6				r_3																					qp						
Nop/Hint	M48	0		i				x_3		x_2			x_4			y			imm_{20a}																		qp						
IP-Relative Branch	B1	4		s		d		wh											imm_{20b}											p			btype			qp						
Counted Branch	B2	4		s		d		wh											imm_{20b}											p			btype			qp						
IP-Relative Call	B3	5		s		d		wh											imm_{20b}											p			b_1			qp						
Indirect Branch	B4	0				d		wh				x_6																	b_2		p			btype			qp						
Indirect Call	B5	1				d		wh																					b_2		p			b_1			qp						
IP-Relative Predict	B6	7		s		ih		t_{2e}											imm_{20b}													$timm_{7a}$					wh						
Indirect Predict	B7	2				ih		t_{2e}				x_6																	b_2			$timm_{7a}$					wh						
Misc	B8	0										x_6																								0						
Break/Nop/Hint	B9	0/2		i								x_6							imm_{20a}																		qp						
FP Arithmetic	F1	8 - D		x		sf				f_4						f_3			f_2						f_1												qp						
Fixed Multiply Add	F2	E		x		x_2				f_4						f_3			f_2						f_1												qp						
FP Select	F3	E		x						f_4						f_3			f_2						f_1												qp						
FP Compare	F4	4		r_b		sf		r_a		p_2						f_3			f_2											t_a		p_1						qp						
FP Class	F5	5					fc_2			p_2						$fclass_{7c}$			f_2											t_a		p_1						qp						
FP Recip Approx	F6	0 - 1		q		sf		x		p_2						f_3			f_2						f_1												qp						
FP Recip Sqrt App	F7	0 - 1		q		sf		x		p_2						f_3									f_1												qp						
FP Min/Max/Pcmp	F8	0 - 1				sf		x		x_6						f_3			f_2						f_1												qp						
FP Merge/Logical	F9	0 - 1						x		x_6						f_3			f_2						f_1												qp						
Convert FP to Fixed	F10	0 - 1				sf		x		x_6									f_2						f_1												qp						
Convert Fixed to FP	F11	0						x		x_6									f_2						f_1												qp						
FP Set Controls	F12	0				sf		x		x_6						$omask_{7c}$			$amask_{7b}$																		qp						
FP Clear Flags	F13	0				sf		x		x_6																											qp						
FP Check Flags	F14	0		s		sf		x		x_6									imm_{20a}																		qp						
Break	F15	0		i				x		x_6									imm_{20a}																		qp						
Nop/Hint	F16	0		i				x		x_6						y			imm_{20a}																		qp						
Break	X1	0		i				x_3				x_6							imm_{20a}																		qp						imm_{41}
Move Imm_{64}	X2	6		i						imm_{9d}						imm_{5c}			i_c	v_c		imm_{7b}								r_1						qp						imm_{41}
Long Branch	X3	C		i		d		wh											imm_{20b}											p			btype			qp						imm_{39}
Long Call	X4	D		i		d		wh											imm_{20b}											p			b_1			qp						imm_{39}
Nop/Hint	X5	0		i				x_3				x_6				y			imm_{20a}																		qp						imm_{41}
		40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Table 4-5. Instruction Field Color Key

Field & Color	
ALU Instruction	Opcode Extension
Integer Instruction	Opcode Hint Extension
Memory Instruction	Immediate
Branch Instruction	Indirect Source
Floating-point Instruction	Predicate Destination
Integer Source	Integer Destination
Memory Source	Memory Source & Destination
Shift Source	Shift Immediate
Special Register Source	Special Register Destination
Floating-point Source	Floating-point Destination
Branch Source	Branch Destination
Address Source	Branch Tag Immediate
Qualifying Predicate	Reserved Instruction
Ignored Field/Instruction	Reserved Inst if PR[qp] is 1
	Reserved B-type Inst if PR[qp] is 1

Table 4-6. Instruction Field Names

Field Name	Description
ar_3	application register source/target
b_1, b_2	branch register source/target
btype	branch type opcode extension
c	complement compare relation opcode extension
$ccount_{5c}$	multimedia shift left complemented shift count immediate
$count_{5b}$, $count_{6d}$	multimedia shift right/shift right pair shift count immediate
$cpos_x$	deposit complemented bit position immediate
cr_3	control register source/target
ct_{2d}	multimedia multiply shift/shift and add shift count immediate
d	branch cache deallocation hint opcode extension
f_n	floating-point register source/target
fc_2, $fclass_{7c}$	floating-point class immediate
hint	memory reference hint opcode extension
i, i_{2b}, $i_{2d,}$ imm_x	immediate of length 1, 2, or *x*
ih	branch importance hint opcode extension
len_{4d}, len_{6d}	extract/deposit length immediate
m	memory reference post-modify opcode extension
$mask_x$	predicate immediate mask
mbt_{4c}, mht_{8c}	multimedia mux1/mux2 immediate
p	sequential prefetch hint opcode extension
p_1, p_2	predicate register target
pos_{6b}	test bit/extract bit position immediate
q	floating-point reciprocal/reciprocal square-root opcode extension
qp	qualifying predicate register source
r_n	general register source/target
s	immediate sign bit
sf	floating-point status field opcode extension

Table 4-6. Instruction Field Names (Continued)

Field Name	Description
sof, sol, sor	alloc size of frame, size of locals, size of rotating immediates
t_a, t_b	compare type opcode extension
t_{2e}, $timm_x$	branch predict tag immediate
v_x	reserved opcode extension field
wh	branch whether hint opcode extension
x, x_n	opcode extension of length 1 or *n*
y	extract/deposit/test bit/test NaT/hint opcode extension
z_a, z_b	multimedia operand size opcode extension

Table 4-7. Special Instruction Notations

Notation	Description
e	instruction ends an instruction group when taken, or for Reserved if PR[qp] is 1 (cyan) encodings and non-branch instructions with a qualifying predicate, when its PR[qp] is 1, or for Reserved (brown) encodings, unconditionally
f	instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0
i	instruction is allowed in the I slot of an MLI template
l	instruction must be the last in an instruction group
p	privileged instruction
t	instruction is only allowed in instruction slot 2

The remaining sections of this chapter present the detailed encodings of all instructions. The "A-Unit Instruction encodings" are presented first, followed by the "I-Unit Instruction Encodings" on page 3:310, "M-Unit Instruction Encodings" on page 3:323, "B-Unit Instruction Encodings" on page 3:349, "F-Unit Instruction Encodings" on page 3:356, and "X-Unit Instruction Encodings" on page 3:365.

Within each section, the instructions are grouped by function, and appear with their instruction format in the same order as in Table 4-4, "Instruction Format Summary" on page 3:296. The opcode extension fields are briefly described and tables present the opcode extension assignments. Unused instruction encodings (appearing as blank entries in the opcode extensions tables) behave in one of four ways:

- Ignored instructions (white color entries in the tables) execute as `nop` instructions.
- Reserved instructions (light gray color in the gray scale version of the tables, brown color in the color version) cause an Illegal Operation fault.
- Reserved if PR[qp] is 1 instructions (dark gray in the gray scale version of the tables, purple in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0.
- Reserved if PR[qp] is 1 B-unit instructions (medium gray in the gray scale version of the tables, cyan in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0. These differ from the Reserved if PR[qp] is 1 instructions (purple) only in their RAW dependency behavior (see "RAW Dependency Table" on page 3:374).

Some processors may implement the Reserved if PR[qp] is 1 (purple) and Reserved if PR[qp] is 1 B-unit (cyan) encodings in the L+X opcode space as Reserved (brown). These encodings appear in the L+X column of Table 4-3 on page 3:295, and in Table 4-69 on page 3:366, Table 4-70 on page 3:366, Table 4-71 on page 3:367, and Table 4-72 on page 3:367. On processors which implement these encodings as Reserved (brown), the operating system is required to provide an Illegal Operation fault handler which emulates them as Reserved if PR[qp] is 1 (cyan/purple) by decoding the reserved opcodes, checking the qualifying predicate, and returning to the next instruction if PR[qp] is 0.

Constant 0 fields in instructions must be 0 or undefined operation results. The undefined operation may include checking that the constant field is 0 and causing an Illegal Operation fault if it is not. If an instruction having a constant 0 field also has a qualifying predicate (qp field), the fault or other undefined operation must not occur if PR[qp] is 0. For constant 0 fields in instruction bits 5:0 (normally used for qp), the fault or other undefined operation may or may not depend on the PR addressed by those bits.

Ignored (white space) fields in instructions should be coded as 0. Although ignored in this revision of the architecture, future architecture revisions may define these fields as hint extensions. These hint extensions will be defined such that the 0 value in each field corresponds to the default hint. It is expected that assemblers will automatically set these fields to zero by default.

Unused opcode hint extension values (white color entries in Hint Completer tables) should not be used by software. Processors must perform the architected functional behavior of the instruction independent of the hint extension value (whether defined or unused), but different processor models may interpret unused opcode hint extension values in different ways, resulting in undesirable performance effects.

4.2 A-Unit Instruction Encodings

4.2.1 Integer ALU

All integer ALU instructions are encoded within major opcode 8 using a 2-bit opcode extension field in bits 35:34 (x_{2a}) and most have a second 2-bit opcode extension field in bits 28:27 (x_{2b}), a 4-bit opcode extension field in bits 32:29 (x_4), and a 1-bit reserved opcode extension field in bit 33 (v_e). Table 4-8 shows the 2-bit x_{2a} and 1-bit v_e assignments, Table 4-9 shows the integer ALU 4-bit+2-bit assignments, and Table 4-12 on page 3:306 shows the multimedia ALU 1-bit+2-bit assignments (which also share major opcode 8).

Table 4-8. Integer ALU 2-bit+1-bit Opcode Extensions

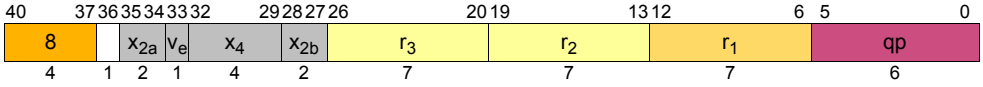
Opcode Bits 40:37	x_{2a} Bits 35:34	v_e Bit 33	
		0	1
8	0	Integer ALU 4-bit+2-bit Ext (Table 4-9)	
	1	Multimedia ALU 1-bit+2-bit Ext (Table 4-12)	
	2	adds – imm_{14} A4	
	3	addp4 – imm_{14} A4	

Table 4-9. Integer ALU 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	v_e Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
				0	1	2	3
8	0	0	0	add A1	add +1 A1		
			1	sub -1 A1	sub A1		
			2	addp4 A1			
			3	and A1	andcm A1	or A1	xor A1
			4	shladd A2			
			5				
			6	shladdp4 A2			
			7				
			8				
			9		sub – imm_8 A3		
			A				
			B	and – imm_8 A3	andcm – imm_8 A3	or – imm_8 A3	xor – imm_8 A3
			C				
			D				
			E				
			F				

4.2.1.1 Integer ALU – Register-Register

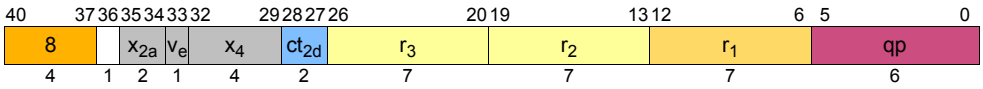
A1



Instruction	Operands	Opcode	Extension			
			x_{2a}	v_e	x_4	x_{2b}
add	$r_1 = r_2, r_3$	8	0	0	0	0
	$r_1 = r_2, r_3, 1$					1
sub	$r_1 = r_2, r_3$				1	1
	$r_1 = r_2, r_3, 1$					0
addp4	$r_1 = r_2, r_3$				2	0
and					3	0
andcm						1
or						2
xor						3

4.2.1.2 Shift Left and Add

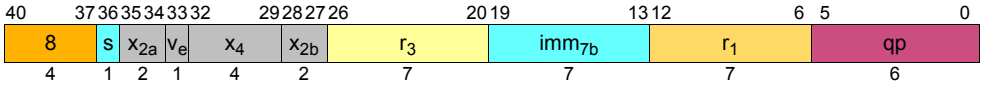
A2



Instruction	Operands	Opcode	Extension		
			x_{2a}	v_e	x_4
shladd	$r_1 = r_2, count_2, r_3$	8	0	0	4
shladdp4					6

4.2.1.3 Integer ALU – Immediate$_8$-Register

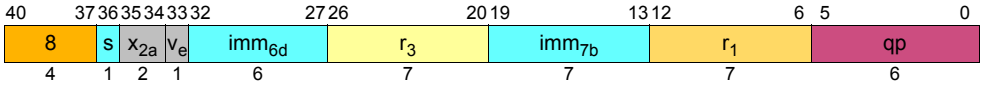
A3



Instruction	Operands	Opcode	Extension			
			x_{2a}	v_e	x_4	x_{2b}
sub	$r_1 = imm_8, r_3$	8	0	0	9	1
and					B	0
andcm						1
or						2
xor						3

4.2.1.4 Add Immediate$_{14}$

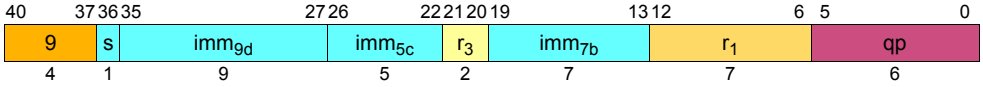
A4



Instruction	Operands	Opcode	Extension	
			x_{2a}	v_e
adds	$r_1 = imm_{14}, r_3$	8	2	0
addp4			3	

4.2.1.5 Add Immediate$_{22}$

A5



Instruction	Operands	Opcode
addl	$r_1 = imm_{22}, r_3$	9

4.2.2 Integer Compare

The integer compare instructions are encoded within major opcodes C - E using a 2-bit opcode extension field (x_2) in bits 35:34 and three 1-bit opcode extension fields in bits 33 (t_a), 36 (t_b), and 12 (c), as shown in Table 4-10. The integer compare immediate instructions are encoded within major opcodes C - E using a 2-bit opcode extension field (x_2) in bits 35:34 and two 1-bit opcode extension fields in bits 33 (t_a) and 12 (c), as shown in Table 4-11.

Table 4-10. Integer Compare Opcode Extensions

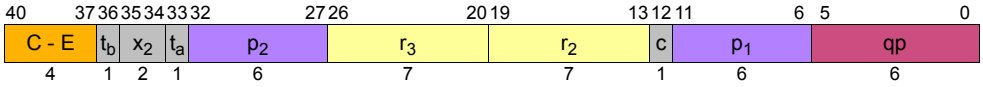
x_2 Bits 35:34	t_b Bit 36	t_a Bit 33	c Bit 12	Opcode Bits 40:37		
				C	D	E
0	0	0	0	cmp.lt A6	cmp.ltu A6	cmp.eq A6
			1	cmp.lt.unc A6	cmp.ltu.unc A6	cmp.eq.unc A6
		1	0	cmp.eq.and A6	cmp.eq.or A6	cmp.eq.or.andcm A6
			1	cmp.ne.and A6	cmp.ne.or A6	cmp.ne.or.andcm A6
	1	0	0	cmp.gt.and A7	cmp.gt.or A7	cmp.gt.or.andcm A7
			1	cmp.le.and A7	cmp.le.or A7	cmp.le.or.andcm A7
		1	0	cmp.ge.and A7	cmp.ge.or A7	cmp.ge.or.andcm A7
			1	cmp.lt.and A7	cmp.lt.or A7	cmp.lt.or.andcm A7
1	0	0	0	cmp4.lt A6	cmp4.ltu A6	cmp4.eq A6
			1	cmp4.lt.unc A6	cmp4.ltu.unc A6	cmp4.eq.unc A6
		1	0	cmp4.eq.and A6	cmp4.eq.or A6	cmp4.eq.or.andcm A6
			1	cmp4.ne.and A6	cmp4.ne.or A6	cmp4.ne.or.andcm A6
	1	0	0	cmp4.gt.and A7	cmp4.gt.or A7	cmp4.gt.or.andcm A7
			1	cmp4.le.and A7	cmp4.le.or A7	cmp4.le.or.andcm A7
		1	0	cmp4.ge.and A7	cmp4.ge.or A7	cmp4.ge.or.andcm A7
			1	cmp4.lt.and A7	cmp4.lt.or A7	cmp4.lt.or.andcm A7

Table 4-11. Integer Compare Immediate Opcode Extensions

x_2 Bits 35:34	t_a Bit 33	c Bit 12	Opcode Bits 40:37		
			C	D	E
2	0	0	cmp.lt – imm_8 A8	cmp.ltu – imm_8 A8	cmp.eq – imm_8 A8
		1	cmp.lt.unc – imm_8 A8	cmp.ltu.unc – imm_8 A8	cmp.eq.unc – imm_8 A8
	1	0	cmp.eq.and – imm_8 A8	cmp.eq.or – imm_8 A8	cmp.eq.or.andcm – imm_8 A8
		1	cmp.ne.and – imm_8 A8	cmp.ne.or – imm_8 A8	cmp.ne.or.andcm – imm_8 A8
3	0	0	cmp4.lt – imm_8 A8	cmp4.ltu – imm_8 A8	cmp4.eq – imm_8 A8
		1	cmp4.lt.unc – imm_8 A8	cmp4.ltu.unc – imm_8 A8	cmp4.eq.unc – imm_8 A8
	1	0	cmp4.eq.and – imm_8 A8	cmp4.eq.or – imm_8 A8	cmp4.eq.or.andcm – imm_8 A8
		1	cmp4.ne.and – imm_8 A8	cmp4.ne.or – imm_8 A8	cmp4.ne.or.andcm – imm_8 A8

4.2.2.1 Integer Compare – Register-Register

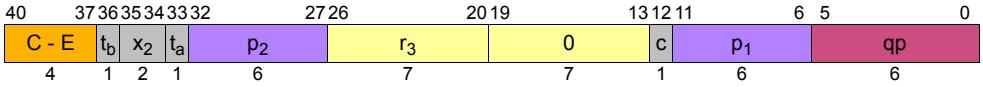
A6



Instruction	Operands	Opcode	Extension			
			x_2	t_b	t_a	c
cmp.lt	$p_1, p_2 = r_2, r_3$	C	0	0	0	0
cmp.ltu		D				
cmp.eq		E				
cmp.lt.unc		C				1
cmp.ltu.unc		D				
cmp.eq.unc		E				
cmp.eq.and		C			1	0
cmp.eq.or		D				
cmp.eq.or.andcm		E				
cmp.ne.and		C				1
cmp.ne.or		D				
cmp.ne.or.andcm		E				
cmp4.lt		C	1	0	0	0
cmp4.ltu		D				
cmp4.eq		E				
cmp4.lt.unc		C				1
cmp4.ltu.unc		D				
cmp4.eq.unc		E				
cmp4.eq.and		C			1	0
cmp4.eq.or		D				
cmp4.eq.or.andcm		E				
cmp4.ne.and		C				1
cmp4.ne.or		D				
cmp4.ne.or.andcm		E				

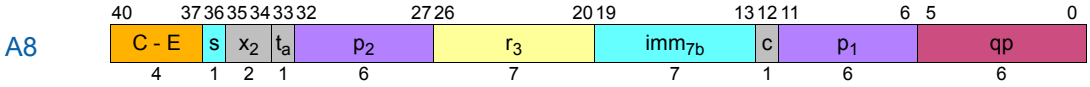
4.2.2.2 Integer Compare to Zero – Register

A7



Instruction	Operands	Opcode	Extension			
			x_2	t_b	t_a	c
cmp.gt.and	p_1, p_2 = r0, r_3	C	0	1	0	0
cmp.gt.or		D				
cmp.gt.or.andcm		E				
cmp.le.and		C				1
cmp.le.or		D				
cmp.le.or.andcm		E				
cmp.ge.and		C			1	0
cmp.ge.or		D				
cmp.ge.or.andcm		E				
cmp.lt.and		C				1
cmp.lt.or		D				
cmp.lt.or.andcm		E				
cmp4.gt.and		C	1		0	0
cmp4.gt.or		D				
cmp4.gt.or.andcm		E				
cmp4.le.and		C				1
cmp4.le.or		D				
cmp4.le.or.andcm		E				
cmp4.ge.and		C			1	0
cmp4.ge.or		D				
cmp4.ge.or.andcm		E				
cmp4.lt.and		C				1
cmp4.lt.or		D				
cmp4.lt.or.andcm		E				

4.2.2.3 Integer Compare – Immediate-Register



Instruction	Operands	Opcode	Extension		
			x_2	t_a	c
cmp.lt	$p_1, p_2 = imm_8, r_3$	C	2	0	0
cmp.ltu		D			
cmp.eq		E			
cmp.lt.unc		C			1
cmp.ltu.unc		D			
cmp.eq.unc		E			
cmp.eq.and		C		1	0
cmp.eq.or		D			
cmp.eq.or.andcm		E			
cmp.ne.and		C			1
cmp.ne.or		D			
cmp.ne.or.andcm		E			
cmp4.lt		C	3	0	0
cmp4.ltu		D			
cmp4.eq		E			
cmp4.lt.unc		C			1
cmp4.ltu.unc		D			
cmp4.eq.unc		E			
cmp4.eq.and		C		1	0
cmp4.eq.or		D			
cmp4.eq.or.andcm		E			
cmp4.ne.and		C			1
cmp4.ne.or		D			
cmp4.ne.or.andcm		E			

4.2.3 Multimedia

All multimedia ALU instructions are encoded within major opcode 8 using two 1-bit opcode extension fields in bits 36 (z_a) and 33 (z_b) and a 2-bit opcode extension field in bits 35:34 (x_{2a}) as shown in Table 4-12. The multimedia ALU instructions also have a 4-bit opcode extension field in bits 32:29 (x_4), and a 2-bit opcode extension field in bits 28:27 (x_{2b}) as shown in Table 4-13 on page 3:307.

Table 4-12. Multimedia ALU 2-bit+1-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	
8	1	0	0	Multimedia ALU Size 1 (Table 4-13)
			1	Multimedia ALU Size 2 (Table 4-14)
		1	0	Multimedia ALU Size 4 (Table 4-15)
			1	

Table 4-13. Multimedia ALU Size 1 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	0	0	0	padd1 A9	padd1.sss A9	padd1.uuu A9	padd1.uus A9
				1	psub1 A9	psub1.sss A9	psub1.uuu A9	psub1.uus A9
				2			pavg1 A9	pavg1.raz A9
				3			pavgsub1 A9	
				4				
				5				
				6				
				7				
				8				
				9	pcmp1.eq A9	pcmp1.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

Table 4-14. Multimedia ALU Size 2 4-bit+2-bit Opcode Extensions

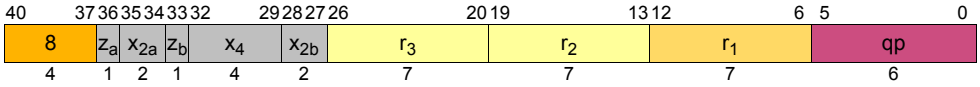
Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	0	1	0	padd2 A9	padd2.sss A9	padd2.uuu A9	padd2.uus A9
				1	psub2 A9	psub2.sss A9	psub2.uuu A9	psub2.uus A9
				2			pavg2 A9	pavg2.raz A9
				3			pavgsub2 A9	
				4	pshladd2 A10			
				5				
				6	pshradd2 A10			
				7				
				8				
				9	pcmp2.eq A9	pcmp2.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

Table 4-15. Multimedia ALU Size 4 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	1	0	0	padd4 A9			
				1	psub4 A9			
				2				
				3				
				4				
				5				
				6				
				7				
				8				
				9	pcmp4.eq A9	pcmp4.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

4.2.3.1 Multimedia ALU

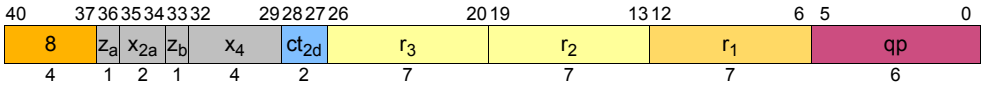
A9



Instruction	Operands	Opcode	Extension				
			x_{2a}	z_a	z_b	x_4	x_{2b}
padd1	$r_1 = r_2, r_3$	8	1	0	0	0	0
padd2					1		
padd4				1	0		
padd1.sss				0	0		1
padd2.sss					1		
padd1.uuu				0	0		2
padd2.uuu					1		
padd1.uus				0	0		3
padd2.uus					1		
psub1				0	0	1	0
psub2					1		
psub4				1	0		
psub1.sss				0	0		1
psub2.sss					1		
psub1.uuu				0	0		2
psub2.uuu					1		
psub1.uus				0	0		3
psub2.uus					1		
pavg1				0	0	2	2
pavg2					1		
pavg1.raz				0	0		3
pavg2.raz					1		
pavgsub1				0	0	3	2
pavgsub2					1		
pcmp1.eq				0	0	9	0
pcmp2.eq					1		
pcmp4.eq				1	0		
pcmp1.gt				0	0		1
pcmp2.gt					1		
pcmp4.gt				1	0		

4.2.3.2 Multimedia Shift and Add

A10



Instruction	Operands	Opcode	Extension			
			x_{2a}	z_a	z_b	x_4
pshladd2	$r_1 = r_2, count_2, r_3$	8	1	0	1	4
pshradd2						6

4.3 I-Unit Instruction Encodings

4.3.1 Multimedia and Variable Shifts

All multimedia multiply/shift/max/min/mix/mux/pack/unpack and variable shift instructions are encoded within major opcode 7 using two 1-bit opcode extension fields in bits 36 (z_a) and 33 (z_b) and a 1-bit reserved opcode extension in bit 32 (v_e) as shown in Table 4-16. They also have a 2-bit opcode extension field in bits 35:34 (x_{2a}) and a 2-bit field in bits 29:28 (x_{2b}) and most have a 2-bit field in bits 31:30 (x_{2c}) as shown in Table 4-17.

Table 4-16. Multimedia and Variable Shift 1-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	
			0	1
7	0	0	Multimedia Size 1 (Table 4-17)	
		1	Multimedia Size 2 (Table 4-18)	
	1	0	Multimedia Size 4 (Table 4-19)	
		1	Variable Shift (Table 4-20)	

Table 4-17. Multimedia Opcode 7 Size 1 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	0	0	0	0	0				
					1				
					2				
					3				
				1	0				
					1				
					2				
					3				
				2	0		unpack1.h I2	mix1.r I2	
					1	pmin1.u I2	pmax1.u I2		
					2		unpack1.l I2	mix1.l I2	
					3			psad1 I2	
				3	0				
					1				
					2			mux1 I3	
					3				

Table 4-18. Multimedia Opcode 7 Size 2 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	0	1	0	0	0	pshr2.u – var I5	pshl2 – var I7		
					1	pmpyshr2.u I1			
					2	pshr2 – var I5			
					3	pmpyshr2 I1			
				1	0				
					1	pshr2.u – fixed I6		popcnt I9	clz I9
					2				
					3	pshr2 – fixed I6			
				2	0	pack2.uss I2	unpack2.h I2	mix2.r I2	
					1				pmpy2.r I2
					2	pack2.sss I2	unpack2.l I2	mix2.l I2	
					3	pmin2 I2	pmax2 I2		pmpy2.l I2
				3	0				
					1		pshl2 – fixed I8		
					2			mux2 I4	
					3				

Table 4-19. Multimedia Opcode 7 Size 4 2-bit Opcode Extensions

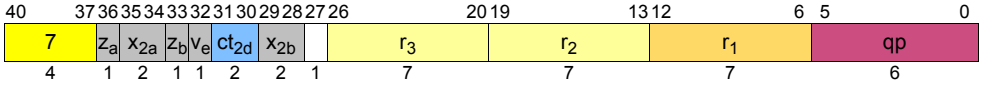
Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	1	0	0	0	0	pshr4.u – var I5	pshl4 – var I7		
					1				mpy4 I2
					2	pshr4 – var I5			
					3				mpyshl4 I2
				1	0				
					1	pshr4.u – fixed I6			
					2				
					3	pshr4 – fixed I6			
				2	0		unpack4.h I2	mix4.r I2	
					1				
					2	pack4.sss I2	unpack4.l I2	mix4.l I2	
					3				
				3	0				
					1		pshl4 – fixed I8		
					2				
					3				

Table 4-20. Variable Shift Opcode 7 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	1	1	0	0	0	shr.u – var I5	shl – var I7		
					1				
					2	shr – var I5			
					3				
				1	0				
					1				
					2				
					3				
				2	0				
					1				
					2				
					3				
				3	0				
					1				
					2				
					3				

4.3.1.1 Multimedia Multiply and Shift

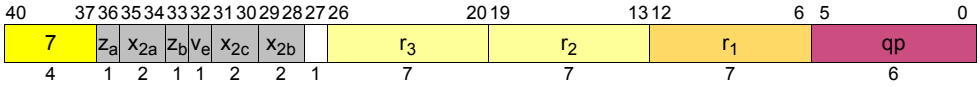
I1



Instruction	Operands	Opcode	Extension				
			z_a	z_b	v_e	x_{2a}	x_{2b}
pmpyshr2	$r_1 = r_2, r_3, count_2$	7	0	1	0	0	3
pmpyshr2.u							1

4.3.1.2 Multimedia Multiply/Mix/Pack/Unpack

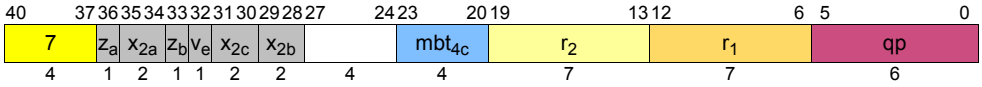
I2



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mpy4 mpyshl4	$r_1 = r_2, r_3$	7	1	0	0	0	1 3	3
pmpy2.r pmpy2.l			0	1			1 3	3
mix1.r mix2.r mix4.r			0 0 1	0 1 0			0	2
mix1.l mix2.l mix4.l			0 0 1	0 1 0			2	
pack2.uss			0	1			0	0
pack2.sss pack4.sss			0 1	1 0			2	
unpack1.h unpack2.h unpack4.h			0 0 1	0 1 0			0	1
unpack1.l unpack2.l unpack4.l			0 0 1	0 1 0			2	
pmin1.u pmax1.u			0	0			1	0 1
pmin2 pmax2			0	1			3	0 1
psad1			0	0			3	2

4.3.1.3 Multimedia Mux1

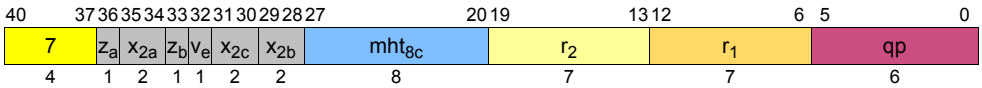
I3



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mux1	$r_1 = r_2$, $mbtype_4$	7	0	0	0	3	2	2

4.3.1.4 Multimedia Mux2

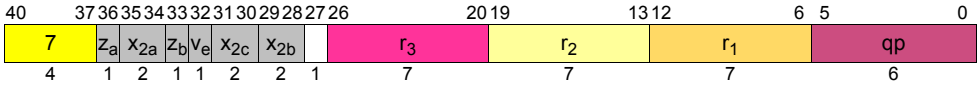
I4



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mux2	$r_1 = r_2$, $mhtype_8$	7	0	1	0	3	2	2

4.3.1.5 Shift Right – Variable

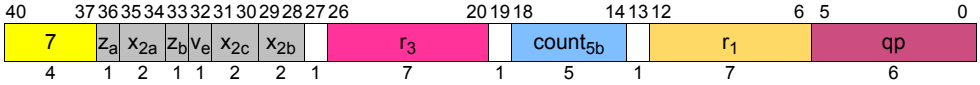
I5



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshr2	$r_1 = r_3, r_2$	7	0	1	0	0	2	0
pshr4			1	0				
shr			1	1				
pshr2.u			0	1			0	
pshr4.u			1	0				
shr.u			1	1				

4.3.1.6 Multimedia Shift Right – Fixed

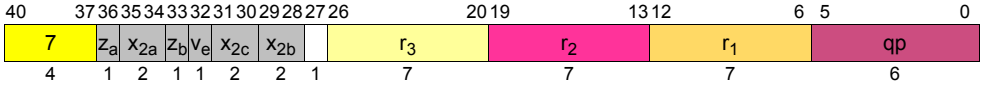
I6



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshr2	$r_1 = r_3, count_5$	7	0	1	0	1	3	0
pshr4			1	0				
pshr2.u			0	1			1	
pshr4.u			1	0				

4.3.1.7 Shift Left – Variable

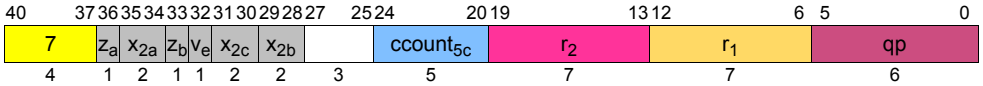
I7



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshl2	$r_1 = r_2, r_3$	7	0	1	0	0	0	1
pshl4			1	0				
shl			1	1				

4.3.1.8 Multimedia Shift Left – Fixed

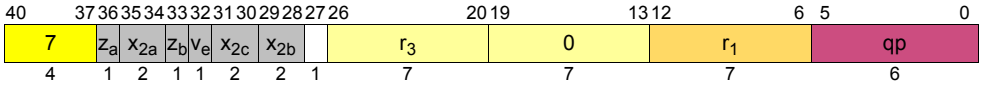
I8



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshl2	$r_1 = r_2, count_5$	7	0	1	0	3	1	1
pshl4			1	0				

4.3.1.9 Bit Strings

I9



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
popcnt	$r_1 = r_3$	7	0	1	0	1	1	2
clz								3

4.3.2 Integer Shifts

The integer shift, test bit, and test NaT instructions are encoded within major opcode 5 using a 2-bit opcode extension field in bits 35:34 (x_2) and a 1-bit opcode extension field in bit 33 (x). The extract and test bit instructions also have a 1-bit opcode extension field in bit 13 (y). Table 4-21 shows the test bit, extract, and shift right pair assignments.

Table 4-21. Integer Shift/Test Bit/Test NaT 2-bit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	x Bit 33	y Bit 13	
			0	1
5	0	0	Test Bit (Table 4-23)	Test NaT/Test Feature (Table 4-23)
	1		extr.u I11	extr I11
	2			
	3		shrp I10	

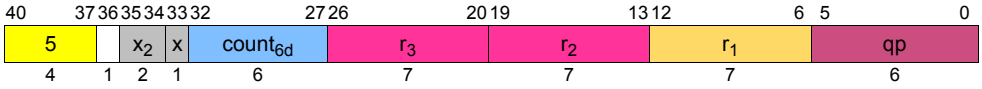
Most deposit instructions also have a 1-bit opcode extension field in bit 26 (y). Table 4-22 shows these assignments.

Table 4-22. Deposit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	x Bit 33	y Bit 26	
			0	1
5	0	1	Test Bit/Test NaT/Test Feature (Table 4-23)	
	1		dep.z I12	dep.z – imm_8 I13
	2			
	3		dep – imm_1 I14	

4.3.2.1 Shift Right Pair

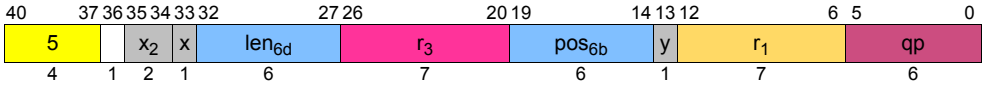
I10



Instruction	Operands	Opcode	Extension	
			x_2	x
shrp	$r_1 = r_2, r_3, count_6$	5	3	0

4.3.2.2 Extract

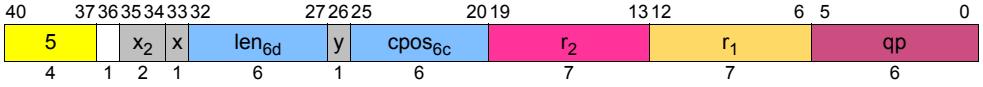
I11



Instruction	Operands	Opcode	Extension		
			x_2	x	y
extr.u extr	$r_1 = r_3, pos_6, len_6$	5	1	0	0 1

4.3.2.3 Zero and Deposit

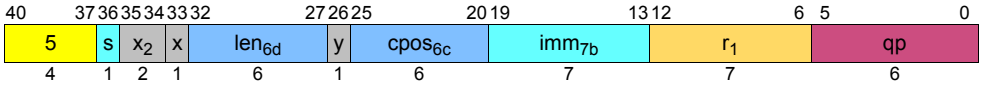
I12



Instruction	Operands	Opcode	Extension		
			x_2	x	y
dep.z	$r_1 = r_2, pos_6, len_6$	5	1	1	0

4.3.2.4 Zero and Deposit Immediate$_8$

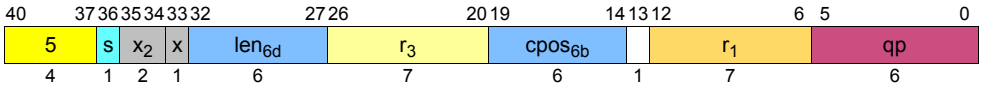
I13



Instruction	Operands	Opcode	Extension		
			x_2	x	y
dep.z	$r_1 = imm_8, pos_6, len_6$	5	1	1	1

4.3.2.5 Deposit Immediate$_1$

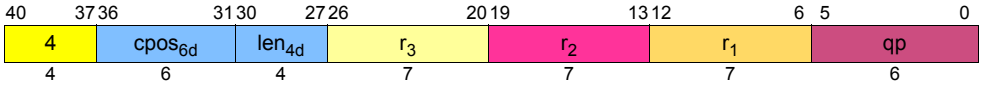
I14



Instruction	Operands	Opcode	Extension	
			x_2	x
dep	$r_1 = imm_1, r_3, pos_6, len_6$	5	3	1

4.3.2.6 Deposit

I15



Instruction	Operands	Opcode
dep	$r_1 = r_2, r_3, pos_6, len_4$	4

4.3.3 Test Bit

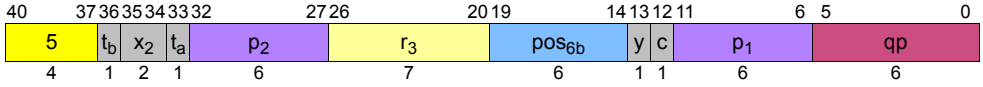
All test bit instructions are encoded within major opcode 5 using a 2-bit opcode extension field in bits 35:34 (x_2) plus five 1-bit opcode extension fields in bits 33 (t_a), 36 (t_b), 12 (c), 13 (y) and 19 (x). Table 4-23 summarizes these assignments.

Table 4-23. Test Bit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	t_a Bit 33	t_b Bit 36	c Bit 12	y Bit 13	x Bit 19	
						0	1
5	0	0	0	0	0	tbit.z I16	
					1	tnat.z I17	tf.z I30
				1	0	tbit.z.unc I16	
					1	tnat.z.unc I17	tf.z.unc I30
			1	0	0	tbit.z.and I16	
					1	tnat.z.and I17	tf.z.and I30
				1	0	tbit.nz.and I16	
					1	tnat.nz.and I17	tf.nz.and I30
		1	0	0	0	tbit.z.or I16	
					1	tnat.z.or I17	tf.z.or I30
				1	0	tbit.nz.or I16	
					1	tnat.nz.or I17	tf.nz.or I30
			1	0	0	tbit.z.or.andcm I16	
					1	tnat.z.or.andcm I17	tf.z.or.andcm I30
				1	0	tbit.nz.or.andcm I16	
					1	tnat.nz.or.andcm I17	tf.nz.or.andcm I30

4.3.3.1 Test Bit

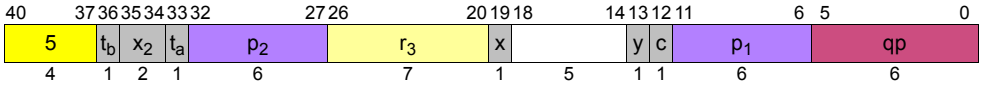
I16



Instruction	Operands	Opcode	Extension				
			x_2	t_a	t_b	y	c
tbit.z	$p_1, p_2 = r_3, pos_6$	5	0	0	0	0	0
tbit.z.unc							1
tbit.z.and					1		0
tbit.nz.and							1
tbit.z.or				1	0		0
tbit.nz.or							1
tbit.z.or.andcm					1		0
tbit.nz.or.andcm							1

4.3.3.2 Test NaT

I17



Instruction	Operands	Opcode	Extension					
			x_2	t_a	t_b	y	x	c
tnat.z	$p_1, p_2 = r_3$	5	0	0	0	1	0	0
tnat.z.unc								1
tnat.z.and					1			0
tnat.nz.and								1
tnat.z.or				1	0			0
tnat.nz.or								1
tnat.z.or.andcm					1			0
tnat.nz.or.andcm								1

4.3.4 Miscellaneous I-Unit Instructions

The miscellaneous I-unit instructions are encoded in major opcode 0 using a 3-bit opcode extension field (x_3) in bits 35:33. Some also have a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-24 shows the 3-bit assignments and Table 4-25 summarizes the 6-bit assignments.

Table 4-24. Misc I-Unit 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	6-bit Ext (Table 4-25)
	1	chk.s.i – int I20
	2	mov to pr.rot – imm_{44} I24
	3	mov to pr I23
	4	
	5	
	6	
	7	mov to b I21

Table 4-25. Misc I-Unit 6-bit Opcode Extensions

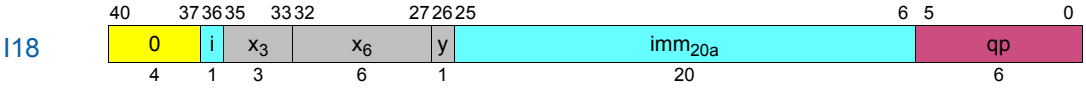
Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 30:27	x_6 Bits 32:31 = 0	x_6 Bits 32:31 = 1	x_6 Bits 32:31 = 2	x_6 Bits 32:31 = 3
0	0	0	break.i I19	zxt1 I29		mov from ip I25
		1	1-bit Ext (Table 4-26)	zxt2 I29		mov from b I22
		2		zxt4 I29		mov.i from ar I28
		3				mov from pr I25
		4		sxt1 I29		
		5		sxt2 I29		
		6		sxt4 I29		
		7				
		8		czx1.l I29		
		9		czx2.l I29		
		A	mov.i to ar – imm_8 I27		mov.i to ar I26	
		B				
		C		czx1.r I29		
		D		czx2.r I29		
		E				
		F				

4.3.4.1 Nop/Hint (I-Unit)

I-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 6-bit opcode extension field in bits 32:27 (x_6), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-26.

Table 4-26. Misc I-Unit 1-bit Opcode Extensions

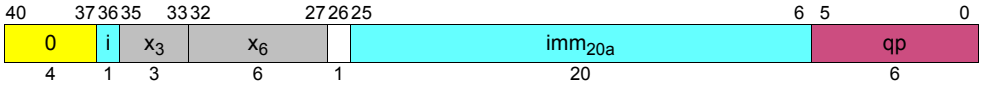
Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 32:27	y Bit 26	
0	0	01	0	nop.i
			1	hint.i



Instruction	Operands	Opcode	Extension x_3	Extension x_6	Extension y
nop.i [i]	imm_{21}	0	0	01	0
hint.i					1

4.3.4.2 Break (I-Unit)

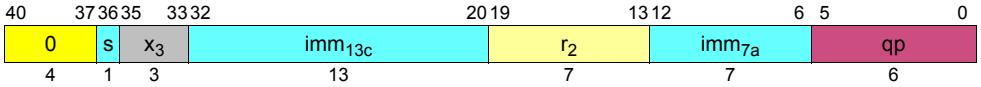
I19



Instruction	Operands	Opcode	Extension	
			x_3	x_6
break.i [i]	*imm*$_{21}$	0	0	00

4.3.4.3 Integer Speculation Check (I-Unit)

I20



Instruction	Operands	Opcode	Extension
			x_3
chk.s.i	*r*$_2$, *target*$_{25}$	0	1

4.3.5 GR/BR Moves

The GR/BR move instructions are encoded in major opcode 0. See "Miscellaneous I-Unit Instructions" on page 3:318 for a summary of the opcode extensions. The mov to BR instruction uses a 2-bit "whether" prediction hint field in bits 21:20 (wh) as shown in Table 4-27.

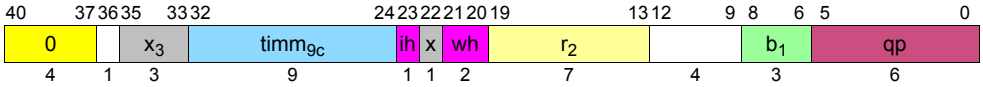
Table 4-27. Move to BR Whether Hint Completer

wh Bits 21:20	*mwh*
0	.sptk
1	*none*
2	.dptk
3	

The mov to BR instruction also uses a 1-bit opcode extension field (x) in bit 22 to distinguish the return form from the normal form, and a 1-bit hint extension in bit 23 (ih) (see Table 4-56 on page 3:354).

4.3.5.1 Move to BR

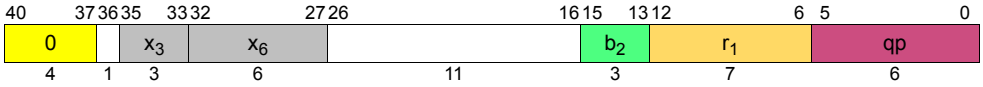
I21



Instruction	Operands	Opcode	Extension			
			x_3	x	ih	wh
mov.*mwh.ih* mov.ret.*mwh.ih*	*b*$_1$ = *r*$_2$, *tag*$_{13}$	0	7	0 1	See Table 4-56 on page 3:354	See Table 4-27 on page 3:320

4.3.5.2 Move from BR

I22



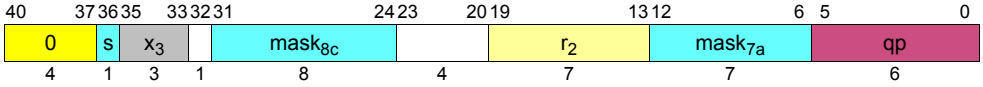
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	$r_1 = b_2$	0	0	31

4.3.6 GR/Predicate/IP Moves

The GR/Predicate/IP move instructions are encoded in major opcode 0. See "Miscellaneous I-Unit Instructions" on page 3:318 for a summary of the opcode extensions.

4.3.6.1 Move to Predicates – Register

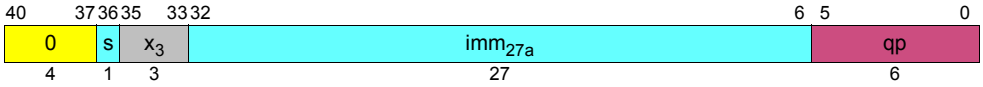
I23



Instruction	Operands	Opcode	Extension
			x_3
mov	pr = r_2, $mask_{17}$	0	3

4.3.6.2 Move to Predicates – Immediate$_{44}$

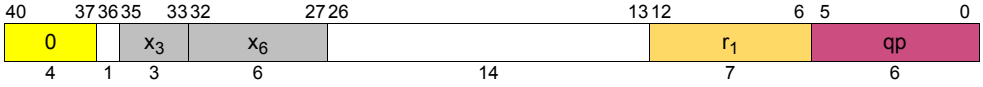
I24



Instruction	Operands	Opcode	Extension
			x_3
mov	pr.rot = imm_{44}	0	2

4.3.6.3 Move from Predicates/IP

I25



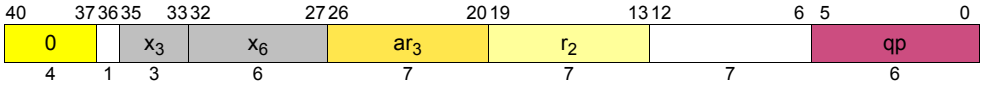
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	r_1 = ip r_1 = pr	0	0	30 33

4.3.7 GR/AR Moves (I-Unit)

The I-Unit GR/AR move instructions are encoded in major opcode 0. (Some ARs are accessed using system/memory management instructions on the M-unit. See "GR/AR Moves (M-Unit)" on page 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)

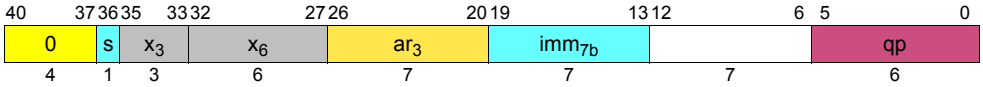
I26



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$ar_3 = r_2$	0	0	2A

4.3.7.2 Move to AR – Immediate$_8$ (I-Unit)

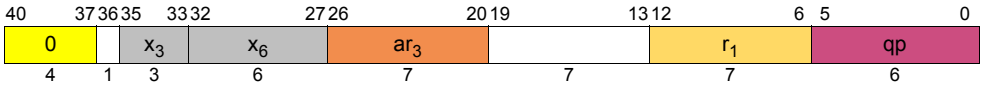
I27



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$ar_3 = imm_8$	0	0	0A

4.3.7.3 Move from AR (I-Unit)

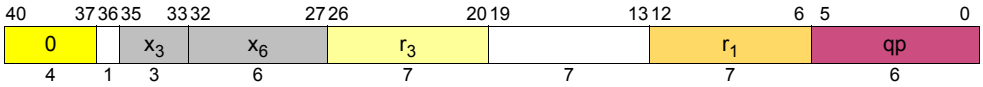
I28



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$r_1 = ar_3$	0	0	32

4.3.8 Sign/Zero Extend/Compute Zero Index

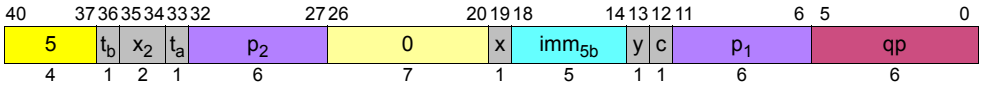
I29



Instruction	Operands	Opcode	Extension	
			x_3	x_6
zxt1	$r_1 = r_3$	0	0	10
zxt2				11
zxt4				12
sxt1				14
sxt2				15
sxt4				16
czx1.l				18
czx2.l				19
czx1.r				1C
czx2.r				1D

4.3.9 Test Feature

I30



Instruction	Operands	Opcode	Extension					
			x_2	t_a	t_b	y	x	c
tf.z	$p_1, p_2 = imm_5$	5	0	0	0	1	1	0
tf.z.unc								1
tf.z.and					1			0
tf.nz.and								1
tf.z.or				1	0			0
tf.nz.or								1
tf.z.or.andcm					1			0
tf.nz.or.andcm								1

4.4 M-Unit Instruction Encodings

4.4.1 Loads and Stores

All load and store instructions are encoded within major opcodes 4, 5, 6, and 7 using a 6-bit opcode extension field in bits 35:30 (x_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

Opcode Bits 40:37	m Bit 36	x Bit 27	
4	0	0	Load/Store (Table 4-30)
	0	1	Semaphore/get FR (Table 4-33)
	1	0	Load +Reg (Table 4-31)
	1	1	

Table 4-29. Floating-point Load/Store/Load Pair/Set FR 1-bit Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	
6	0	0	FP Load/Store (Table 4-34)
	0	1	FP Load Pair/set FR (Table 4-37)
	1	0	FP Load +Reg (Table 4-35)
	1	1	FP Load Pair +Imm (Table 4-38)

The integer load/store opcode extensions are summarized in Table 4-30 on page 3:324, Table 4-31 on page 3:324, and Table 4-32 on page 3:325, and the semaphore and get FR opcode extensions in Table 4-33 on page 3:325. The floating-point load/store

opcode extensions are summarized in Table 4-34 on page 3:326, Table 4-35 on page 3:326, and Table 4-36 on page 3:327, the floating-point load pair and set FR opcode extensions in Table 4-37 on page 3:327 and Table 4-38 on page 3:328.

Table 4-30. Integer Load/Store Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	0	0	ld1 M2	ld2 M2	ld4 M2	ld8 M2
			1	ld1.s M2	ld2.s M2	ld4.s M2	ld8.s M2
			2	ld1.a M2	ld2.a M2	ld4.a M2	ld8.a M2
			3	ld1.sa M2	ld2.sa M2	ld4.sa M2	ld8.sa M2
			4	ld1.bias M2	ld2.bias M2	ld4.bias M2	ld8.bias M2
			5	ld1.acq M2	ld2.acq M2	ld4.acq M2	ld8.acq M2
			6				ld8.fill M2
			7				
			8	ld1.c.clr M2	ld2.c.clr M2	ld4.c.clr M2	ld8.c.clr M2
			9	ld1.c.nc M2	ld2.c.nc M2	ld4.c.nc M2	ld8.c.nc M2
			A	ld1.c.clr.acq M2	ld2.c.clr.acq M2	ld4.c.clr.acq M2	ld8.c.clr.acq M2
			B				
			C	st1 M6	st2 M6	st4 M6	st8 M6
			D	st1.rel M6	st2.rel M6	st4.rel M6	st8.rel M6
			E				st8.spill M6
			F				

Table 4-31. Integer Load +Reg Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	1	0	0	ld1 M2	ld2 M2	ld4 M2	ld8 M2
			1	ld1.s M2	ld2.s M2	ld4.s M2	ld8.s M2
			2	ld1.a M2	ld2.a M2	ld4.a M2	ld8.a M2
			3	ld1.sa M2	ld2.sa M2	ld4.sa M2	ld8.sa M2
			4	ld1.bias M2	ld2.bias M2	ld4.bias M2	ld8.bias M2
			5	ld1.acq M2	ld2.acq M2	ld4.acq M2	ld8.acq M2
			6				ld8.fill M2
			7				
			8	ld1.c.clr M2	ld2.c.clr M2	ld4.c.clr M2	ld8.c.clr M2
			9	ld1.c.nc M2	ld2.c.nc M2	ld4.c.nc M2	ld8.c.nc M2
			A	ld1.c.clr.acq M2	ld2.c.clr.acq M2	ld4.c.clr.acq M2	ld8.c.clr.acq M2
			B				
			C				
			D				
			E				
			F				

Table 4-32. Integer Load/Store +Imm Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 35:32	Bits 31:30			
		0	1	2	3
5	0	ld1 M3	ld2 M3	ld4 M3	ld8 M3
	1	ld1.s M3	ld2.s M3	ld4.s M3	ld8.s M3
	2	ld1.a M3	ld2.a M3	ld4.a M3	ld8.a M3
	3	ld1.sa M3	ld2.sa M3	ld4.sa M3	ld8.sa M3
	4	ld1.bias M3	ld2.bias M3	ld4.bias M3	ld8.bias M3
	5	ld1.acq M3	ld2.acq M3	ld4.acq M3	ld8.acq M3
	6				ld8.fill M3
	7				
	8	ld1.c.clr M3	ld2.c.clr M3	ld4.c.clr M3	ld8.c.clr M3
	9	ld1.c.nc M3	ld2.c.nc M3	ld4.c.nc M3	ld8.c.nc M3
	A	ld1.c.clr.acq M3	ld2.c.clr.acq M3	ld4.c.clr.acq M3	ld8.c.clr.acq M3
	B				
	C	st1 M5	st2 M5	st4 M5	st8 M5
	D	st1.rel M5	st2.rel M5	st4.rel M5	st8.rel M5
	E				st8.spill M5
	F				

Table 4-33. Semaphore/Get FR/16-Byte Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	1	0	cmpxchg1.acq M16	cmpxchg2.acq M16	cmpxchg4.acq M16	cmpxchg8.acq M16
			1	cmpxchg1.rel M16	cmpxchg2.rel M16	cmpxchg4.rel M16	cmpxchg8.rel M16
			2	xchg1 M16	xchg2 M16	xchg4 M16	xchg8 M16
			3				
			4			fetchadd4.acq M17	fetchadd8.acq M17
			5			fetchadd4.rel M17	fetchadd8.rel M17
			6				
			7	getf.sig M19	getf.exp M19	getf.s M19	getf.d M19
			8	cmp8xchg16.acq M16			
			9	cmp8xchg16.rel M16			
			A	ld16 M2			
			B	ld16.acq M2			
			C	st16 M6			
			D	st16.rel M6			
			E				
			F				

Table 4-34. Floating-point Load/Store/Lfetch Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	0	0	0	ldfe M9	ldf8 M9	ldfs M9	ldfd M9
			1	ldfe.s M9	ldf8.s M9	ldfs.s M9	ldfd.s M9
			2	ldfe.a M9	ldf8.a M9	ldfs.a M9	ldfd.a M9
			3	ldfe.sa M9	ldf8.sa M9	ldfs.sa M9	ldfd.sa M9
			4				
			5				
			6				ldf.fill M9
			7				
			8	ldfe.c.clr M9	ldf8.c.clr M9	ldfs.c.clr M9	ldfd.c.clr M9
			9	ldfe.c.nc M9	ldf8.c.nc M9	ldfs.c.nc M9	ldfd.c.nc M9
			A				
			B	lfetch M18	lfetch.excl M18	lfetch.fault M18	lfetch.fault.excl M18
			C	stfe M13	stf8 M13	stfs M13	stfd M13
			D				
			E				stf.spill M13
			F				

Table 4-35. Floating-point Load/Lfetch +Reg Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	1	0	0	ldfe M7	ldf8 M7	ldfs M7	ldfd M7
			1	ldfe.s M7	ldf8.s M7	ldfs.s M7	ldfd.s M7
			2	ldfe.a M7	ldf8.a M7	ldfs.a M7	ldfd.a M7
			3	ldfe.sa M7	ldf8.sa M7	ldfs.sa M7	ldfd.sa M7
			4				
			5				
			6				ldf.fill M7
			7				
			8	ldfe.c.clr M7	ldf8.c.clr M7	ldfs.c.clr M7	ldfd.c.clr M7
			9	ldfe.c.nc M7	ldf8.c.nc M7	ldfs.c.nc M7	ldfd.c.nc M7
			A				
			B	lfetch M20	lfetch.excl M20	lfetch.fault M20	lfetch.fault.excl M20
			C				
			D				
			E				
			F				

Table 4-36. Floating-point Load/Store/Lfetch +Imm Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 35:32	Bits 31:30			
		0	1	2	3
7	0	ldfe M8	ldf8 M8	ldfs M8	ldfd M8
	1	ldfe.s M8	ldf8.s M8	ldfs.s M8	ldfd.s M8
	2	ldfe.a M8	ldf8.a M8	ldfs.a M8	ldfd.a M8
	3	ldfe.sa M8	ldf8.sa M8	ldfs.sa M8	ldfd.sa M8
	4				
	5				
	6				ldf.fill M8
	7				
	8	ldfe.c.clr M8	ldf8.c.clr M8	ldfs.c.clr M8	ldfd.c.clr M8
	9	ldfe.c.nc M8	ldf8.c.nc M8	ldfs.c.nc M8	ldfd.c.nc M8
	A				
	B	lfetch M22	lfetch.excl M22	lfetch.fault M22	lfetch.fault.excl M22
	C	stfe M10	stf8 M10	stfs M10	stfd M10
	D				
	E				stf.spill M10
	F				

Table 4-37. Floating-point Load Pair/Set FR Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	0	1	0		ldfp8 M11	ldfps M11	ldfpd M11
			1		ldfp8.s M11	ldfps.s M11	ldfpd.s M11
			2		ldfp8.a M11	ldfps.a M11	ldfpd.a M11
			3		ldfp8.sa M11	ldfps.sa M11	ldfpd.sa M11
			4				
			5				
			6				
			7	setf.sig M18	setf.exp M18	setf.s M18	setf.d M18
			8		ldfp8.c.clr M11	ldfps.c.clr M11	ldfpd.c.clr M11
			9		ldfp8.c.nc M11	ldfps.c.nc M11	ldfpd.c.nc M11
			A				
			B				
			C				
			D				
			E				
			F				

Table 4-38. Floating-point Load Pair +Imm Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	1	1	0		ldfp8 M12	ldfps M12	ldfpd M12
			1		ldfp8.s M12	ldfps.s M12	ldfpd.s M12
			2		ldfp8.a M12	ldfps.a M12	ldfpd.a M12
			3		ldfp8.sa M12	ldfps.sa M12	ldfpd.sa M12
			4				
			5				
			6				
			7				
			8		ldfp8.c.clr M12	ldfps.c.clr M12	ldfpd.c.clr M12
			9		ldfp8.c.nc M12	ldfps.c.nc M12	ldfpd.c.nc M12
			A				
			B				
			C				
			D				
			E				
			F				

The load and store instructions all have a 2-bit cache locality opcode hint extension field in bits 29:28 (hint). Table 4-39 and Table 4-40 summarize these assignments.

Table 4-39. Load Hint Completer

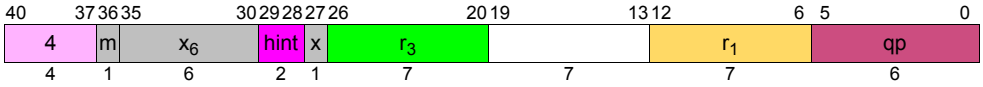
hint Bits 29:28	*ldhint*
0	*none*
1	.nt1
2	
3	.nta

Table 4-40. Store Hint Completer

hint Bits 29:28	*sthint*
0	*none*
1	
2	
3	.nta

4.4.1.1 Integer Load

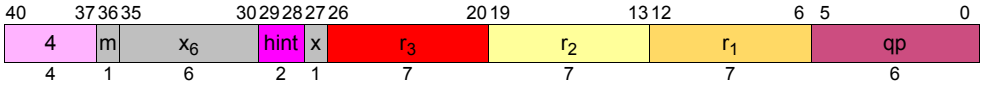
M2



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ld1.*ldhint*	$r_1 = [r_3]$	4	0	0	00	See Table 4-39 on page 3:328
ld2.*ldhint*					01	
ld4.*ldhint*					02	
ld8.*ldhint*					03	
ld1.s.*ldhint*					04	
ld2.s.*ldhint*					05	
ld4.s.*ldhint*					06	
ld8.s.*ldhint*					07	
ld1.a.*ldhint*					08	
ld2.a.*ldhint*					09	
ld4.a.*ldhint*					0A	
ld8.a.*ldhint*					0B	
ld1.sa.*ldhint*					0C	
ld2.sa.*ldhint*					0D	
ld4.sa.*ldhint*					0E	
ld8.sa.*ldhint*					0F	
ld1.bias.*ldhint*					10	
ld2.bias.*ldhint*					11	
ld4.bias.*ldhint*					12	
ld8.bias.*ldhint*					13	
ld1.acq.*ldhint*					14	
ld2.acq.*ldhint*					15	
ld4.acq.*ldhint*					16	
ld8.acq.*ldhint*					17	
ld8.fill.*ldhint*					1B	
ld1.c.clr.*ldhint*					20	
ld2.c.clr.*ldhint*					21	
ld4.c.clr.*ldhint*					22	
ld8.c.clr.*ldhint*					23	
ld1.c.nc.*ldhint*					24	
ld2.c.nc.*ldhint*					25	
ld4.c.nc.*ldhint*					26	
ld8.c.nc.*ldhint*					27	
ld1.c.clr.acq.*ldhint*					28	
ld2.c.clr.acq.*ldhint*					29	
ld4.c.clr.acq.*ldhint*					2A	
ld8.c.clr.acq.*ldhint*					2B	
ld16.*ldhint*	r_1, ar.csd = $[r_3]$		0	1	28	
ld16.acq.*ldhint*					2C	

4.4.1.2 Integer Load – Increment by Register

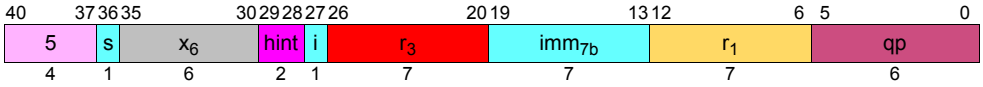
M2



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ld1.*ldhint*	$r_1 = [r_3], r_2$	4	1	0	00	See Table 4-39 on page 3:328
ld2.*ldhint*					01	
ld4.*ldhint*					02	
ld8.*ldhint*					03	
ld1.s.*ldhint*					04	
ld2.s.*ldhint*					05	
ld4.s.*ldhint*					06	
ld8.s.*ldhint*					07	
ld1.a.*ldhint*					08	
ld2.a.*ldhint*					09	
ld4.a.*ldhint*					0A	
ld8.a.*ldhint*					0B	
ld1.sa.*ldhint*					0C	
ld2.sa.*ldhint*					0D	
ld4.sa.*ldhint*					0E	
ld8.sa.*ldhint*					0F	
ld1.bias.*ldhint*					10	
ld2.bias.*ldhint*					11	
ld4.bias.*ldhint*					12	
ld8.bias.*ldhint*					13	
ld1.acq.*ldhint*					14	
ld2.acq.*ldhint*					15	
ld4.acq.*ldhint*					16	
ld8.acq.*ldhint*					17	
ld8.fill.*ldhint*					1B	
ld1.c.clr.*ldhint*					20	
ld2.c.clr.*ldhint*					21	
ld4.c.clr.*ldhint*					22	
ld8.c.clr.*ldhint*					23	
ld1.c.nc.*ldhint*					24	
ld2.c.nc.*ldhint*					25	
ld4.c.nc.*ldhint*					26	
ld8.c.nc.*ldhint*					27	
ld1.c.clr.acq.*ldhint*					28	
ld2.c.clr.acq.*ldhint*					29	
ld4.c.clr.acq.*ldhint*					2A	
ld8.c.clr.acq.*ldhint*					2B	

4.4.1.3 Integer Load – Increment by Immediate

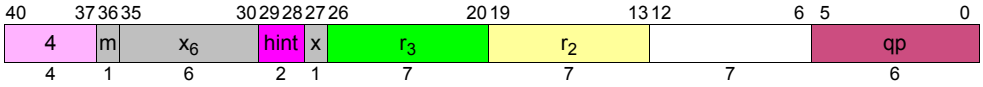
M3



Instruction	Operands	Opcode	Extension	
			x_6	hint
ld1.*ldhint*	$r_1 = [r_3]$, imm_9	5	00	See Table 4-39 on page 3:328
ld2.*ldhint*			01	
ld4.*ldhint*			02	
ld8.*ldhint*			03	
ld1.s.*ldhint*			04	
ld2.s.*ldhint*			05	
ld4.s.*ldhint*			06	
ld8.s.*ldhint*			07	
ld1.a.*ldhint*			08	
ld2.a.*ldhint*			09	
ld4.a.*ldhint*			0A	
ld8.a.*ldhint*			0B	
ld1.sa.*ldhint*			0C	
ld2.sa.*ldhint*			0D	
ld4.sa.*ldhint*			0E	
ld8.sa.*ldhint*			0F	
ld1.bias.*ldhint*			10	
ld2.bias.*ldhint*			11	
ld4.bias.*ldhint*			12	
ld8.bias.*ldhint*			13	
ld1.acq.*ldhint*			14	
ld2.acq.*ldhint*			15	
ld4.acq.*ldhint*			16	
ld8.acq.*ldhint*			17	
ld8.fill.*ldhint*			1B	
ld1.c.clr.*ldhint*			20	
ld2.c.clr.*ldhint*			21	
ld4.c.clr.*ldhint*			22	
ld8.c.clr.*ldhint*			23	
ld1.c.nc.*ldhint*			24	
ld2.c.nc.*ldhint*			25	
ld4.c.nc.*ldhint*			26	
ld8.c.nc.*ldhint*			27	
ld1.c.clr.acq.*ldhint*			28	
ld2.c.clr.acq.*ldhint*			29	
ld4.c.clr.acq.*ldhint*			2A	
ld8.c.clr.acq.*ldhint*			2B	

4.4.1.4 Integer Store

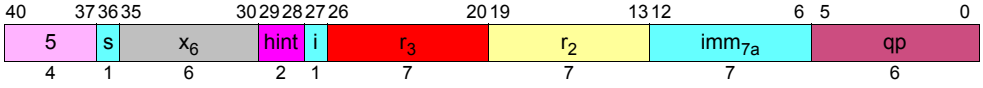
M6



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
st1.*sthint*	$[r_3] = r_2$	4	0	0	30	See Table 4-40 on page 3:328
st2.*sthint*					31	
st4.*sthint*					32	
st8.*sthint*					33	
st1.rel.*sthint*					34	
st2.rel.*sthint*					35	
st4.rel.*sthint*					36	
st8.rel.*sthint*					37	
st8.spill.*sthint*					3B	
st16.*sthint*	$[r_3] = r_2$, ar.csd		0	1	30	
st16.rel.*sthint*					34	

4.4.1.5 Integer Store – Increment by Immediate

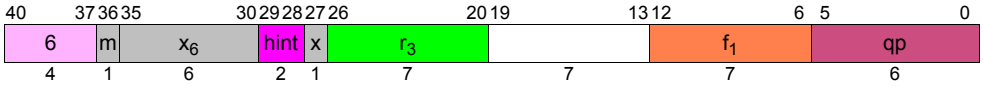
M5



Instruction	Operands	Opcode	Extension	
			x_6	hint
st1.*sthint*	$[r_3] = r_2, imm_9$	5	30	See Table 4-40 on page 3:328
st2.*sthint*			31	
st4.*sthint*			32	
st8.*sthint*			33	
st1.rel.*sthint*			34	
st2.rel.*sthint*			35	
st4.rel.*sthint*			36	
st8.rel.*sthint*			37	
st8.spill.*sthint*			3B	

4.4.1.6 Floating-point Load

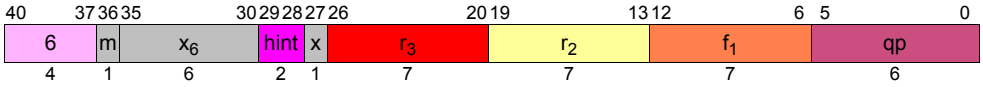
M9



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfs.*ldhint*	$f_1 = [r_3]$	6	0	0	02	See Table 4-39 on page 3:328
ldfd.*ldhint*					03	
ldf8.*ldhint*					01	
ldfe.*ldhint*					00	
ldfs.s.*ldhint*					06	
ldfd.s.*ldhint*					07	
ldf8.s.*ldhint*					05	
ldfe.s.*ldhint*					04	
ldfs.a.*ldhint*					0A	
ldfd.a.*ldhint*					0B	
ldf8.a.*ldhint*					09	
ldfe.a.*ldhint*					08	
ldfs.sa.*ldhint*					0E	
ldfd.sa.*ldhint*					0F	
ldf8.sa.*ldhint*					0D	
ldfe.sa.*ldhint*					0C	
ldf.fill.*ldhint*					1B	
ldfs.c.clr.*ldhint*					22	
ldfd.c.clr.*ldhint*					23	
ldf8.c.clr.*ldhint*					21	
ldfe.c.clr.*ldhint*					20	
ldfs.c.nc.*ldhint*					26	
ldfd.c.nc.*ldhint*					27	
ldf8.c.nc.*ldhint*					25	
ldfe.c.nc.*ldhint*					24	

4.4.1.7 Floating-point Load – Increment by Register

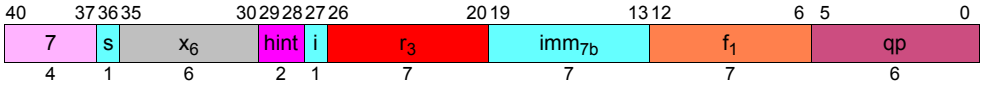
M7



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfs.*ldhint*	f_1 = [r_3], r_2	6	1	0	02	See Table 4-39 on page 3:328
ldfd.*ldhint*					03	
ldf8.*ldhint*					01	
ldfe.*ldhint*					00	
ldfs.s.*ldhint*					06	
ldfd.s.*ldhint*					07	
ldf8.s.*ldhint*					05	
ldfe.s.*ldhint*					04	
ldfs.a.*ldhint*					0A	
ldfd.a.*ldhint*					0B	
ldf8.a.*ldhint*					09	
ldfe.a.*ldhint*					08	
ldfs.sa.*ldhint*					0E	
ldfd.sa.*ldhint*					0F	
ldf8.sa.*ldhint*					0D	
ldfe.sa.*ldhint*					0C	
ldf.fill.*ldhint*					1B	
ldfs.c.clr.*ldhint*					22	
ldfd.c.clr.*ldhint*					23	
ldf8.c.clr.*ldhint*					21	
ldfe.c.clr.*ldhint*					20	
ldfs.c.nc.*ldhint*					26	
ldfd.c.nc.*ldhint*					27	
ldf8.c.nc.*ldhint*					25	
ldfe.c.nc.*ldhint*					24	

4.4.1.8 Floating-point Load – Increment by Immediate

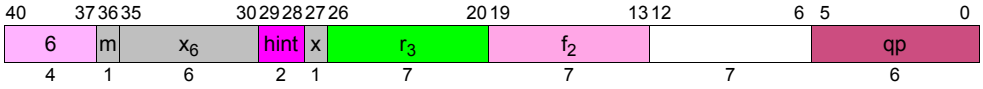
M8



Instruction	Operands	Opcode	Extension	
			x_6	hint
ldfs.*ldhint*	$f_1 = [r_3]$, imm_9	7	02	See Table 4-39 on page 3:328
ldfd.*ldhint*			03	
ldf8.*ldhint*			01	
ldfe.*ldhint*			00	
ldfs.s.*ldhint*			06	
ldfd.s.*ldhint*			07	
ldf8.s.*ldhint*			05	
ldfe.s.*ldhint*			04	
ldfs.a.*ldhint*			0A	
ldfd.a.*ldhint*			0B	
ldf8.a.*ldhint*			09	
ldfe.a.*ldhint*			08	
ldfs.sa.*ldhint*			0E	
ldfd.sa.*ldhint*			0F	
ldf8.sa.*ldhint*			0D	
ldfe.sa.*ldhint*			0C	
ldf.fill.*ldhint*			1B	
ldfs.c.clr.*ldhint*			22	
ldfd.c.clr.*ldhint*			23	
ldf8.c.clr.*ldhint*			21	
ldfe.c.clr.*ldhint*			20	
ldfs.c.nc.*ldhint*			26	
ldfd.c.nc.*ldhint*			27	
ldf8.c.nc.*ldhint*			25	
ldfe.c.nc.*ldhint*			24	

4.4.1.9 Floating-point Store

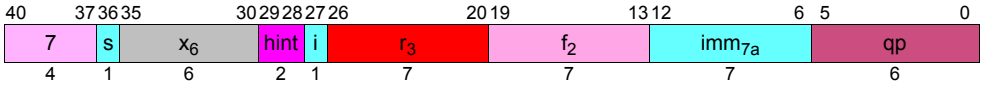
M13



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
stfs.*sthint*	$[r_3] = f_2$	6	0	0	32	See Table 4-40 on page 3:328
stfd.*sthint*					33	
stf8.*sthint*					31	
stfe.*sthint*					30	
stf.spill.*sthint*					3B	

4.4.1.10 Floating-point Store – Increment by Immediate

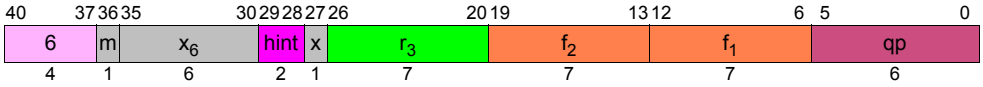
M10



Instruction	Operands	Opcode	Extension	
			x_6	hint
stfs.*sthint*	$[r_3] = f_2, imm_9$	7	32	See Table 4-40 on page 3:328
stfd.*sthint*			33	
stf8.*sthint*			31	
stfe.*sthint*			30	
stf.spill.*sthint*			3B	

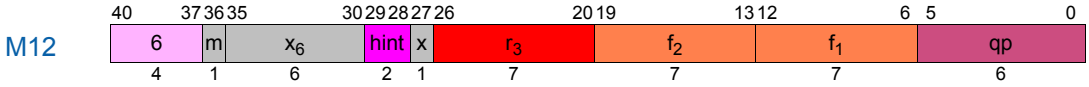
4.4.1.11 Floating-point Load Pair

M11



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfps.*ldhint*	$f_1, f_2 = [r_3]$	6	0	1	02	See Table 4-39 on page 3:328
ldfpd.*ldhint*					03	
ldfp8.*ldhint*					01	
ldfps.s.*ldhint*					06	
ldfpd.s.*ldhint*					07	
ldfp8.s.*ldhint*					05	
ldfps.a.*ldhint*					0A	
ldfpd.a.*ldhint*					0B	
ldfp8.a.*ldhint*					09	
ldfps.sa.*ldhint*					0E	
ldfpd.sa.*ldhint*					0F	
ldfp8.sa.*ldhint*					0D	
ldfps.c.clr.*ldhint*					22	
ldfpd.c.clr.*ldhint*					23	
ldfp8.c.clr.*ldhint*					21	
ldfps.c.nc.*ldhint*					26	
ldfpd.c.nc.*ldhint*					27	
ldfp8.c.nc.*ldhint*					25	

4.4.1.12 Floating-point Load Pair – Increment by Immediate



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfps.*ldhint*	f_1, f_2 = [r_3], 8	6	1	1	02	See Table 4-39 on page 3:328
ldfpd.*ldhint*	f_1, f_2 = [r_3], 16				03	
ldfp8.*ldhint*					01	
ldfps.s.*ldhint*	f_1, f_2 = [r_3], 8				06	
ldfpd.s.*ldhint*	f_1, f_2 = [r_3], 16				07	
ldfp8.s.*ldhint*					05	
ldfps.a.*ldhint*	f_1, f_2 = [r_3], 8				0A	
ldfpd.a.*ldhint*	f_1, f_2 = [r_3], 16				0B	
ldfp8.a.*ldhint*					09	
ldfps.sa.*ldhint*	f_1, f_2 = [r_3], 8				0E	
ldfpd.sa.*ldhint*	f_1, f_2 = [r_3], 16				0F	
ldfp8.sa.*ldhint*					0D	
ldfps.c.clr.*ldhint*	f_1, f_2 = [r_3], 8				22	
ldfpd.c.clr.*ldhint*	f_1, f_2 = [r_3], 16				23	
ldfp8.c.clr.*ldhint*					21	
ldfps.c.nc.*ldhint*	f_1, f_2 = [r_3], 8				26	
ldfpd.c.nc.*ldhint*	f_1, f_2 = [r_3], 16				27	
ldfp8.c.nc.*ldhint*					25	

4.4.2 Line Prefetch

The line prefetch instructions are encoded in major opcodes 6 and 7 along with the floating-point load/store instructions. See "Loads and Stores" on page 3:323 for a summary of the opcode extensions.

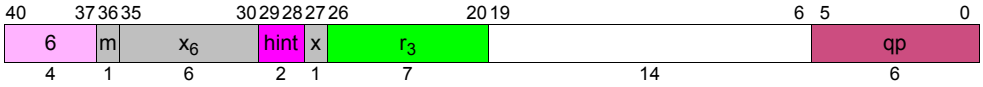
The line prefetch instructions all have a 2-bit cache locality opcode hint extension field in bits 29:28 (hint) as shown in Table 4-44.

Table 4-41. Line Prefetch Hint Completer

hint Bits 29:28	*lfhint*
0	*none*
1	.nt1
2	.nt2
3	.nta

4.4.2.1 Line Prefetch

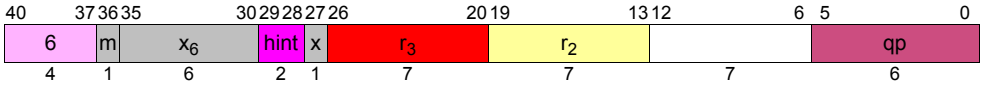
M13



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
lfetch.excl.*lfhint* lfetch.fault.*lfhint* lfetch.fault.excl.*lfhint*	[r_3]	6	0	0	2D 2E 2F	See Table 4-41 on page 3:337

4.4.2.2 Line Prefetch – Increment by Register

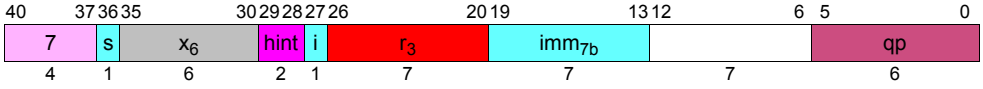
M14



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
lfetch.*lfhint* lfetch.excl.*lfhint* lfetch.fault.*lfhint* lfetch.fault.excl.*lfhint*	[r_3], r_2	6	1	0	2C 2D 2E 2F	See Table 4-41 on page 3:337

4.4.2.3 Line Prefetch – Increment by Immediate

M15



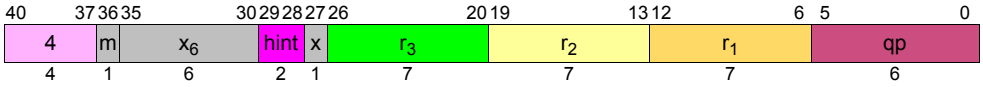
Instruction	Operands	Opcode	Extension	
			x_6	hint
lfetch.*lfhint* lfetch.excl.*lfhint* lfetch.fault.*lfhint* lfetch.fault.excl.*lfhint*	[r_3], imm_9	7	2C 2D 2E 2F	See Table 4-41 on page 3:337

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

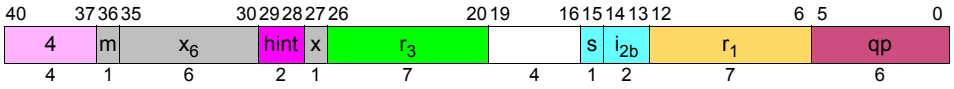
M16



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
cmpxchg1.acq.*ldhint*	r_1 = [r_3], r_2, ar.ccv	4	0	1	00	See Table 4-39 on page 3:328
cmpxchg2.acq.*ldhint*					01	
cmpxchg4.acq.*ldhint*					02	
cmpxchg8.acq.*ldhint*					03	
cmpxchg1.rel.*ldhint*					04	
cmpxchg2.rel.*ldhint*					05	
cmpxchg4.rel.*ldhint*					06	
cmpxchg8.rel.*ldhint*					07	
cmp8xchg16.acq.*ldhint*	r_1 = [r_3], r_2, ar.csd, ar.ccv				20	
cmp8xchg16.rel.*ldhint*					24	
xchg1.*ldhint*	r_1 = [r_3], r_2				08	
xchg2.*ldhint*					09	
xchg4.*ldhint*					0A	
xchg8.*ldhint*					0B	

4.4.3.2 Fetch and Add – Immediate

M17



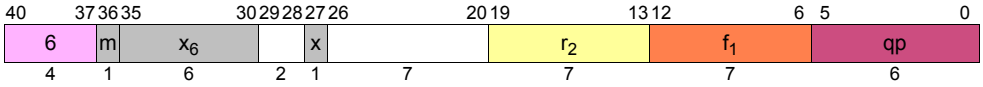
Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
fetchadd4.acq.*ldhint*	r_1 = [r_3], inc_3	4	0	1	12	See Table 4-39 on page 3:328
fetchadd8.acq.*ldhint*					13	
fetchadd4.rel.*ldhint*					16	
fetchadd8.rel.*ldhint*					17	

4.4.4 Set/Get FR

The set FR instructions are encoded in major opcode 6 along with the floating-point load/store instructions. The get FR instructions are encoded in major opcode 4 along with the integer load/store instructions. See "Loads and Stores" on page 3:323 for a summary of the opcode extensions.

4.4.4.1 Set FR

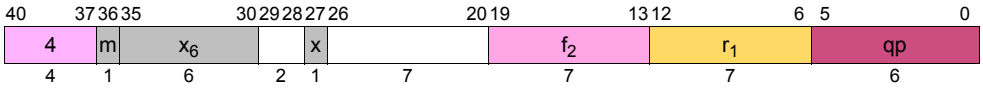
M18



Instruction	Operands	Opcode	Extension		
			m	x	x_6
setf.sig setf.exp setf.s setf.d	$f_1 = r_2$	6	0	1	1C 1D 1E 1F

4.4.4.2 Get FR

M19



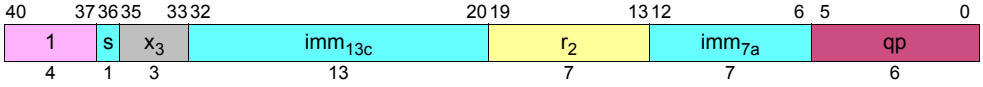
Instruction	Operands	Opcode	Extension		
			m	x	x_6
getf.sig getf.exp getf.s getf.d	$r_1 = f_2$	4	0	1	1C 1D 1E 1F

4.4.5 Speculation and Advanced Load Checks

The speculation and advanced load check instructions are encoded in major opcodes 0 and 1 along with the system/memory management instructions. See "System/Memory Management" on page 3:345 for a summary of the opcode extensions.

4.4.5.1 Integer Speculation Check (M-Unit)

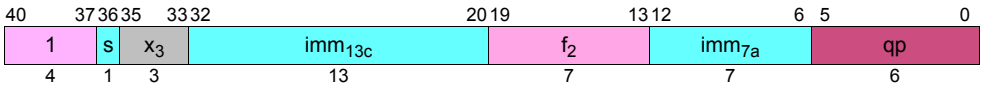
M20



Instruction	Operands	Opcode	Extension
			x_3
chk.s.m	r_2, $target_{25}$	1	1

4.4.5.2 Floating-point Speculation Check

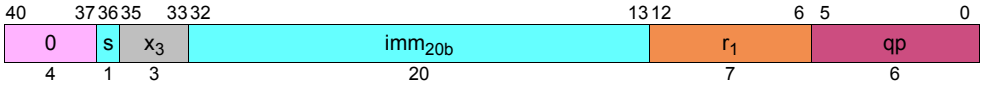
M21



Instruction	Operands	Opcode	Extension
			x_3
chk.s	f_2, $target_{25}$	1	3

4.4.5.3 Integer Advanced Load Check

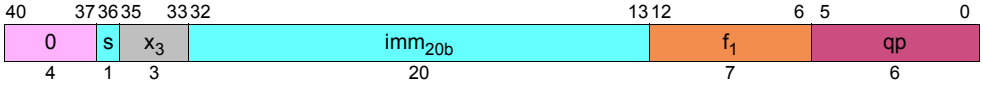
M22



Instruction	Operands	Opcode	Extension
			x_3
chk.a.nc	r_1, $target_{25}$	0	4
chk.a.clr			5

4.4.5.4 Floating-point Advanced Load Check

M23



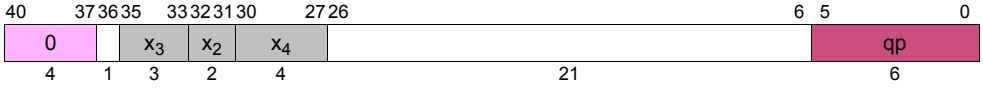
Instruction	Operands	Opcode	Extension
			x_3
chk.a.nc	f_1, $target_{25}$	0	6
chk.a.clr			7

4.4.6 Cache/Synchronization/RSE/ALAT

The cache/synchronization/RSE/ALAT instructions are encoded in major opcode 0 along with the memory management instructions. See "System/Memory Management" on page 3:345 for a summary of the opcode extensions.

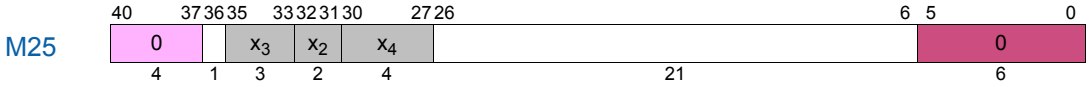
4.4.6.1 Sync/Fence/Serialize/ALAT Control

M24



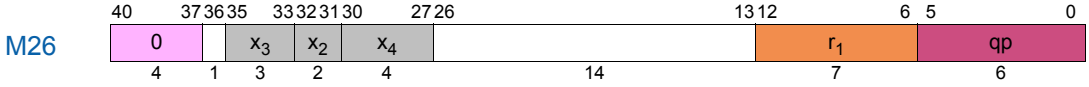
Instruction	Opcode	Extension		
		x_3	x_4	x_2
invala	0	0	0	1
fwb			0	
mf			2	2
mf.a			3	
srlz.d			0	
srlz.i			1	3
sync.i			3	

4.4.6.2 RSE Control



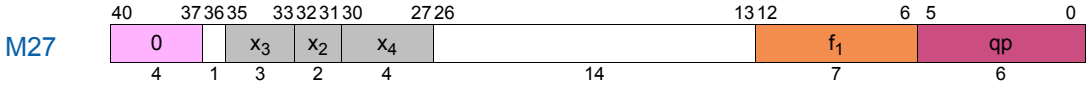
Instruction	Opcode	Extension		
		x_3	x_4	x_2
flushrs [f] loadrs [f]	0	0	C A	0

4.4.6.3 Integer ALAT Entry Invalidate



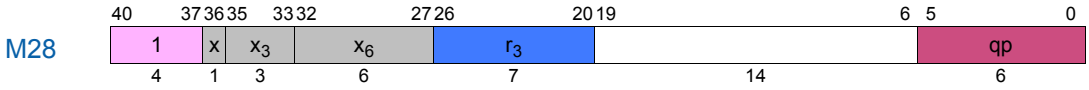
Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
invala.e	r_1	0	0	2	1

4.4.6.4 Floating-point ALAT Entry Invalidate



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
invala.e	f_1	0	0	3	1

4.4.6.5 Flush Cache



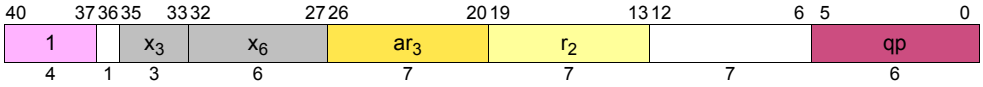
Instruction	Operands	Opcode	Extension		
			x_3	x_6	x
fc fc.i	r_3	1	0	30	0 1

4.4.7 GR/AR Moves (M-Unit)

The M-Unit GR/AR move instructions are encoded in major opcode 0 along with the system/memory management instructions. (Some ARs are accessed using system control instructions on the I-unit. See "GR/AR Moves (I-Unit)" on page 3:321.) See "System/Memory Management" on page 3:345 for a summary of the M-Unit GR/AR opcode extensions.

4.4.7.1 Move to AR – Register (M-Unit)

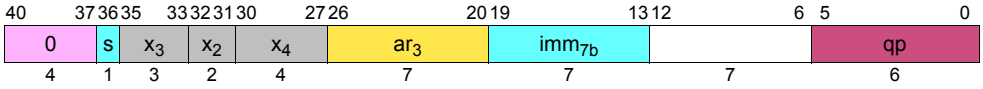
M29



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.m	$ar_3 = r_2$	1	0	2A

4.4.7.2 Move to AR – Immediate$_8$ (M-Unit)

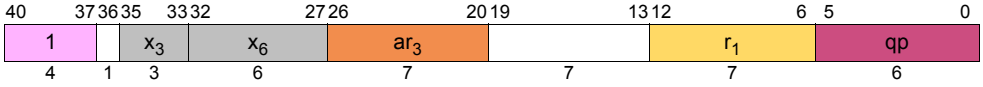
M30



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
mov.m	$ar_3 = imm_8$	0	0	8	2

4.4.7.3 Move from AR (M-Unit)

M31



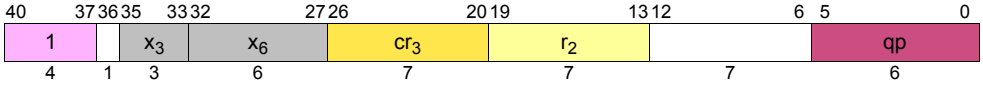
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.m	$r_1 = ar_3$	1	0	22

4.4.8 GR/CR Moves

The GR/CR move instructions are encoded in major opcode 0 along with the system/memory management instructions. See "System/Memory Management" on page 3:345 for a summary of the opcode extensions.

4.4.8.1 Move to CR

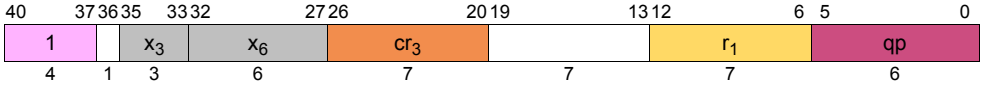
M32



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov ^{p}	$cr_3 = r_2$	1	0	2C

4.4.8.2 Move from CR

M33



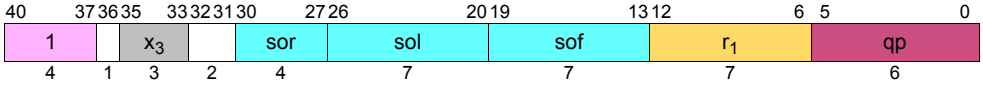
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov ^{p}	$r_1 = cr_3$	1	0	24

4.4.9 Miscellaneous M-Unit Instructions

The miscellaneous M-unit instructions are encoded in major opcode 0 along with the system/memory management instructions. See "System/Memory Management" on page 3:345 for a summary of the opcode extensions.

4.4.9.1 Allocate Register Stack Frame

M34

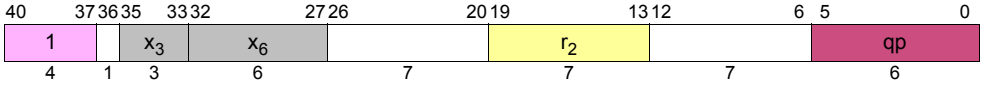


Instruction	Operands	Opcode	Extension
			x_3
alloc [f]	r_1 = ar.pfs, *i*, *l*, *o*, *r*	1	6

Note: The three immediates in the instruction encoding are formed from the operands as follows:
sof = *i* + *l* + *o*
sol = *i* + *l*
sor = *r* >> 3

4.4.9.2 Move to PSR

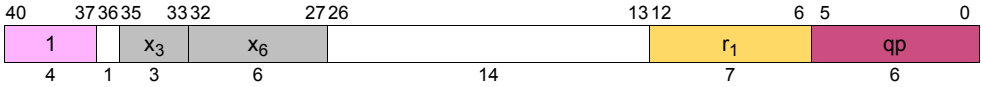
M35



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	psr.l = r_2	1	0	2D
mov	psr.um = r_2			29

4.4.9.3 Move from PSR

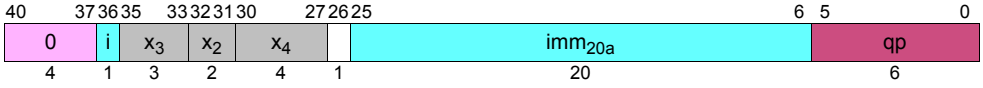
M36



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	r_1 = psr	1	0	25
mov	r_1 = psr.um			21

4.4.9.4 Break (M-Unit)

M37



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
break.m	imm_{21}	0	0	0	0

4.4.10 System/Memory Management

All system/memory management instructions are encoded within major opcodes 0 and 1 using a 3-bit opcode extension field (x_3) in bits 35:33. Some instructions also have a 4-bit opcode extension field (x_4) in bits 30:27, or a 6-bit opcode extension field (x_6) in bits 32:27. Most of the instructions having a 4-bit opcode extension field also have a 2-bit extension field (x_2) in bits 32:31. Table 4-42 shows the 3-bit assignments for opcode 0, Table 4-43 summarizes the 4-bit+2-bit assignments for opcode 0, Table 4-44 shows the 3-bit assignments for opcode 1, and Table 4-45 summarizes the 6-bit assignments for opcode 1.

Table 4-42. Opcode 0 System/Memory Management 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	System/Memory Management 4-bit+2-bit Ext (Table 4-43)
	1	
	2	
	3	
	4	chk.a.nc – int M22
	5	chk.a.clr – int M22
	6	chk.a.nc – fp M23
	7	chk.a.clr – fp M23

Table 4-43. Opcode 0 System/Memory Management 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_4 Bits 30:27	x_2 Bits 32:31			
			0	1	2	3
0	0	0	break.m M37	invala M24	fwb M24	srlz.d M24
		1	1-bit Ext (Table 4-46)			srlz.i M24
		2		invala.e – int M26	mf M24	
		3		invala.e – fp M27	mf.a M24	sync.i M24
		4	sum M44			
		5	rum M44			
		6	ssm M44			
		7	rsm M44			
		8			mov.m to ar – imm_8 M30	
		9				
		A	loadrs M25			
		B				
		C	flushrs M25			
		D				
		E				
		F				

Table 4-44. Opcode 1 System/Memory Management 3-bit Opcode Extensions

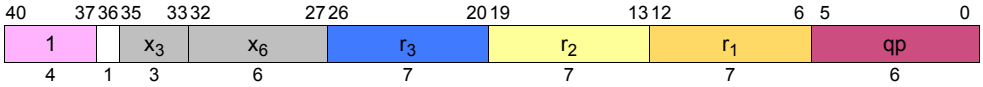
Opcode Bits 40:37	x_3 Bits 35:33	
1	0	System/Memory Management 6-bit Ext (Table 4-45)
	1	chk.s.m – int M20
	2	
	3	chk.s – fp M21
	4	
	5	
	6	alloc M34
	7	

Table 4-45. Opcode 1 System/Memory Management 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
1	0	0	mov to rr M42	mov from rr M43		fc M28
		1	mov to dbr M42	mov from dbr M43	mov from psr.um M36	probe.rw.fault – imm_2 M40
		2	mov to ibr M42	mov from ibr M43	mov.m from ar M31	probe.r.fault – imm_2 M40
		3	mov to pkr M42	mov from pkr M43		probe.w.fault – imm_2 M40
		4	mov to pmc M42	mov from pmc M43	mov from cr M33	ptc.e M47
		5	mov to pmd M42	mov from pmd M43	mov from psr M36	
		6				
		7		mov from cpuid M43		
		8		probe.r – imm_2 M39		probe.r M38
		9	ptc.l M45	probe.w – imm_2 M39	mov to psr.um M35	probe.w M38
		A	ptc.g M45	thash M46	mov.m to ar M29	
		B	ptc.ga M45	ttag M46		
		C	ptr.d M45		mov to cr M32	
		D	ptr.i M45		mov to psr.l M35	
		E	itr.d M42	tpa M46	itc.d M41	
		F	itr.i M42	tak M46	itc.i M41	

4.4.10.1 Probe – Register

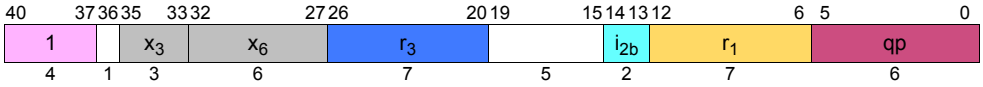
M38



Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.r probe.w	$r_1 = r_3, r_2$	1	0	38 39

4.4.10.2 Probe – Immediate$_2$

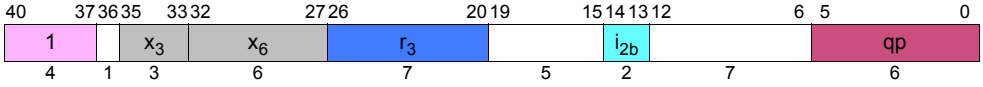
M39



Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.r	$r_1 = r_3, imm_2$	1	0	18
probe.w				19

4.4.10.3 Probe Fault – Immediate$_2$

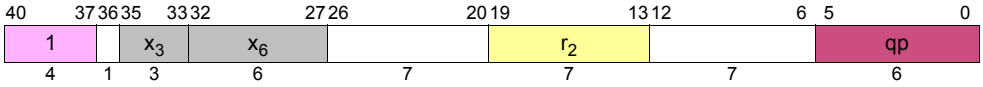
M40



Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.rw.fault	r_3, imm_2	1	0	31
probe.r.fault				32
probe.w.fault				33

4.4.10.4 Translation Cache Insert

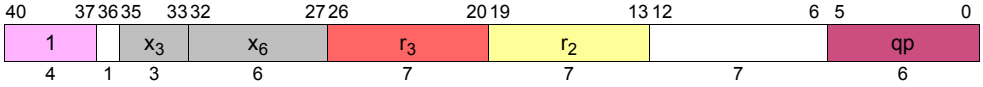
M41



Instruction	Operands	Opcode	Extension	
			x_3	x_6
itc.d [l p]	r_2	1	0	2E
itc.i [l p]				2F

4.4.10.5 Move to Indirect Register/Translation Register Insert

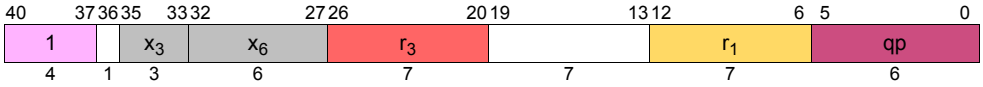
M42



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	rr[r_3] = r_2	1	0	00
	dbr[r_3] = r_2			01
	ibr[r_3] = r_2			02
	pkr[r_3] = r_2			03
	pmc[r_3] = r_2			04
	pmd[r_3] = r_2			05
itr.d [p]	dtr[r_3] = r_2			0E
itr.i [p]	itr[r_3] = r_2			0F

4.4.10.6 Move from Indirect Register

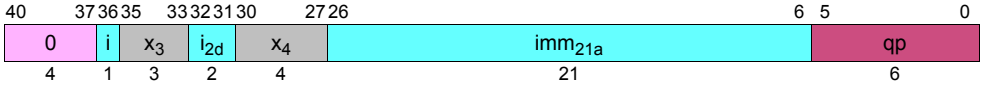
M43



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	r_1 = rr[r_3]	1	0	10
	r_1 = dbr[r_3]			11
	r_1 = ibr[r_3]			12
	r_1 = pkr[r_3]			13
	r_1 = pmc[r_3]			14
mov	r_1 = pmd[r_3]			15
	r_1 = cpuid[r_3]			17

4.4.10.7 Set/Reset User/System Mask

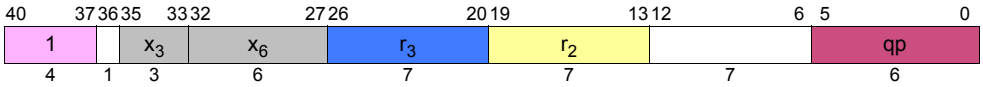
M44



Instruction	Operands	Opcode	Extension	
			x_3	x_4
sum	imm_{24}	0	0	4
rum				5
ssm [p]				6
rsm [p]				7

4.4.10.8 Translation Purge

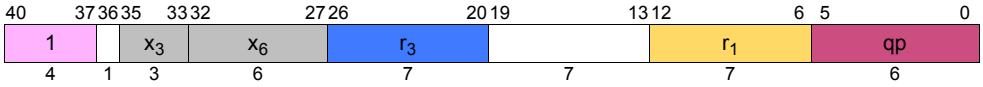
M45



Instruction	Operands	Opcode	Extension	
			x_3	x_6
ptc.l [p]	r_3, r_2	1	0	09
ptc.g [l p]				0A
ptc.ga [l p]				0B
ptr.d [p]				0C
ptr.i [p]				0D

4.4.10.9 Translation Access

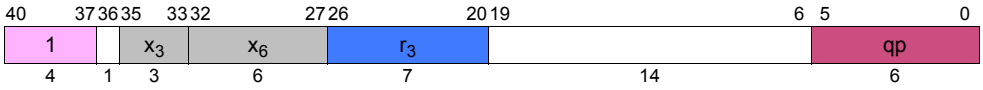
M46



Instruction	Operands	Opcode	Extension	
			x_3	x_6
thash ttag tpa [p] tak [p]	$r_1 = r_3$	1	0	1A 1B 1E 1F

4.4.10.10 Purge Translation Cache Entry

M47



Instruction	Operands	Opcode	Extension	
			x_3	x_6
ptc.e [p]	r_3	1	0	34

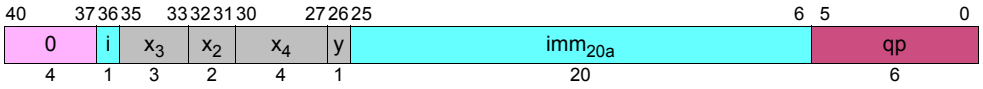
4.4.11 Nop/Hint (M-Unit)

M-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 2-bit opcode extension field in bits 32:31 (x_2), a 4-bit opcode extension field in bits 30:27 (x_4), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-46.

Table 4-46. Misc M-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_4 Bits 30:27	x_2 Bits 32:31	y Bit 26	
0	0	1	0	0	nop.m
				1	hint.m

M48



Instruction	Operands	Opcode	Extension			
			x_3	x_4	x_2	y
nop.m hint.m	imm_{21}	0	0	1	0	0 1

4.5 B-Unit Instruction Encodings

The branch-unit includes branch, predict, and miscellaneous instructions.

4.5.1 Branches

Opcode 0 is used for indirect branch, opcode 1 for indirect call, opcode 4 for IP-relative branch, and opcode 5 for IP-relative call.

The IP-relative branch instructions encoded within major opcode 4 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-47.

Table 4-47. IP-Relative Branch Types

Opcode Bits 40:37	btype Bits 8:6	
4	0	br.cond B1
	1	e
	2	br.wexit B1
	3	br.wtop B1
	4	e
	5	br.cloop B2
	6	br.cexit B2
	7	br.ctop B2

The indirect branch, indirect return, and miscellaneous branch-unit instructions are encoded within major opcode 0 using a 6-bit opcode extension field in bits 32:27 (x_6). Table 4-48 summarizes these assignments.

Table 4-48. Indirect/Miscellaneous Branch Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 30:27	Bits 32:31			
		0	1	2	3
0	0	break.b B9	epc B8	Indirect Branch (Table 4-49)	e
	1		e	Indirect Return (Table 4-50)	e
	2	cover B8	e	e	e
	3	e	e	e	e
	4	clrrrb B8	e	e	e
	5	clrrrb.pr B8	e	e	e
	6	e	e	e	e
	7	e	e	e	e
	8	rfi B8	vmsw.0 B8	e	e
	9		vmsw.1 B8	e	e
	A	e	e	e	e
	B	e	e	e	e
	C	bsw.0 B8	e	e	e
	D	bsw.1 B8	e	e	e
	E	e	e	e	e
	F	e	e	e	e

The indirect branch instructions encoded within major opcodes 0 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-49.

Table 4-49. Indirect Branch Types

Opcode Bits 40:37	x_6 Bits 32:27	btype Bits 8:6	
0	20	0	br.cond B4
		1	br.ia B4
		2	e
		3	e
		4	e
		5	e
		6	e
		7	e

The indirect return branch instructions encoded within major opcodes 0 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-50.

Table 4-50. Indirect Return Branch Types

Opcode Bits 40:37	x_6 Bits 32:27	btype Bits 8:6	
0	21	0	e
		1	e
		2	e
		3	e
		4	br.ret B4
		5	e
		6	e
		7	e

All of the branch instructions have a 1-bit sequential prefetch opcode hint extension field, p, in bit 12. Table 4-51 summarizes these assignments.

Table 4-51. Sequential Prefetch Hint Completer

p Bit 12	*ph*
0	.few
1	.many

The IP-relative and indirect branch instructions all have a 2-bit branch prediction "whether" opcode hint extension field in bits 34:33 (wh) as shown in Table 4-52. Indirect call instructions have a 3-bit "whether" opcode hint extension field in bits 34:32 (wh) as shown in Table 4-53.

Table 4-52. Branch Whether Hint Completer

wh Bits 34:33	*bwh*
0	.sptk
1	.spnt
2	.dptk
3	.dpnt

Table 4-53. Indirect Call Whether Hint Completer

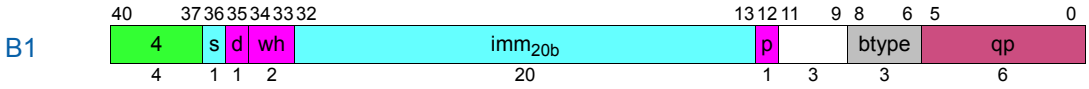
wh Bits 34:32	*bwh*
0	
1	.sptk
2	
3	.spnt
4	
5	.dptk
6	
7	.dpnt

The branch instructions also have a 1-bit branch cache deallocation opcode hint extension field in bit 35 (d) as shown in Table 4-54.

Table 4-54. Branch Cache Deallocation Hint Completer

d Bit 35	*dh*
0	*none*
1	.clr

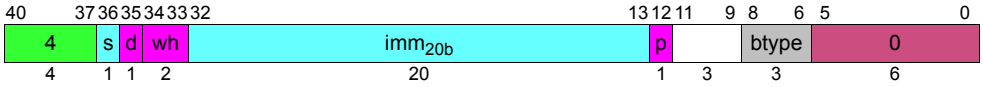
4.5.1.1 IP-Relative Branch



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cond.*bwh.ph.dh* [e] br.wexit.*bwh.ph.dh* [e t] br.wtop.*bwh.ph.dh* [e t]	*target*$_{25}$	4	0 2 3	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

4.5.1.2 IP-Relative Counted Branch

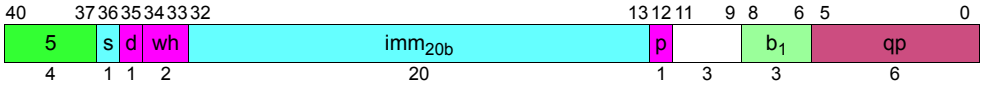
B2



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cloop.*bwh.ph.dh* e t br.cexit.*bwh.ph.dh* e t br.ctop.*bwh.ph.dh* e t	$target_{25}$	4	5 6 7	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

4.5.1.3 IP-Relative Call

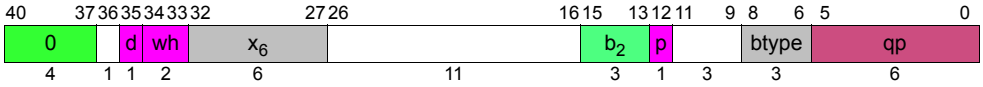
B3



Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.*bwh.ph.dh* e	$b_1 = target_{25}$	5	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

4.5.1.4 Indirect Branch

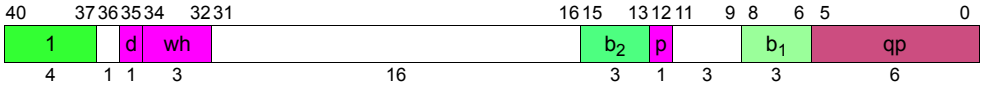
B4



Instruction	Operands	Opcode	Extension				
			x_6	btype	p	wh	d
br.cond.*bwh.ph.dh* e br.ia.*bwh.ph.dh* e br.ret.*bwh.ph.dh* e	b_2	0	20 21	0 1 4	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

4.5.1.5 Indirect Call

B5



Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.*bwh.ph.dh* e	$b_1 = b_2$	1	See Table 4-51 on page 3:351	See Table 4-53 on page 3:352	See Table 4-54 on page 3:352

4.5.2 Branch Predict/Nop/Hint

The branch predict, nop, and hint instructions are encoded in major opcodes 2 (Indirect Predict/Nop/Hint) and 7 (IP-relative Predict). The indirect predict, nop, and hint instructions in major opcode 2 use a 6-bit opcode extension field in bits 32:27 (x_6). Table 4-55 summarizes these assignments.

Table 4-55. Indirect Predict/Nop/Hint Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 30:27	Bits 32:31			
		0	1	2	3
2	0	nop.b B9	brp B7		
	1	hint.b B9	brp.ret B7		
	2				
	3				
	4				
	5				
	6				
	7				
	8				
	9				
	A				
	B				
	C				
	D				
	E				
	F				

The branch predict instructions all have a 1-bit branch importance opcode hint extension field in bit 35 (ih). The mov to BR instruction (page 3:320) also has this hint in bit 23. Table 4-56 shows these assignments.

Table 4-56. Branch Importance Hint Completer

ih Bit 23 or Bit 35	*ih*
0	*none*
1	.imp

The IP-relative branch predict instructions have a 2-bit branch prediction "whether" opcode hint extension field in bits 4:3 (wh) as shown in Table 4-57. Note that the combination of the .loop or .exit whether hint completer with the *none* importance hint completer is undefined.

Table 4-57. IP-Relative Predict Whether Hint Completer

wh Bits 4:3	*ipwh*
0	.sptk
1	.loop
2	.dptk
3	.exit

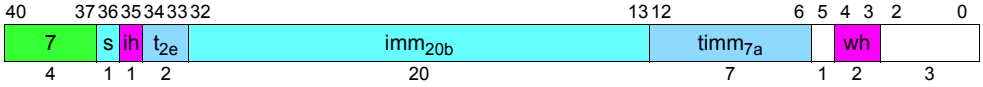
The indirect branch predict instructions have a 2-bit branch prediction "whether" opcode hint extension field in bits 4:3 (wh) as shown in Table 4-58.

Table 4-58. Indirect Predict Whether Hint Completer

wh Bits 4:3	*indwh*
0	.sptk
1	
2	.dptk
3	

4.5.2.1 IP-Relative Predict

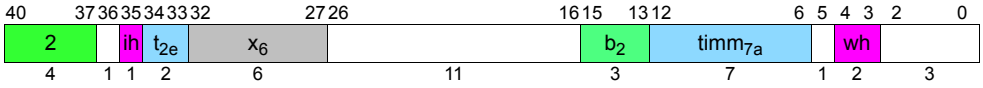
B6



Instruction	Operands	Opcode	Extension	
			ih	wh
brp.*ipwh.ih*	*target*$_{25}$, *tag*$_{13}$	7	See Table 4-56 on page 3:354	See Table 4-57 on page 3:354

4.5.2.2 Indirect Predict

B7



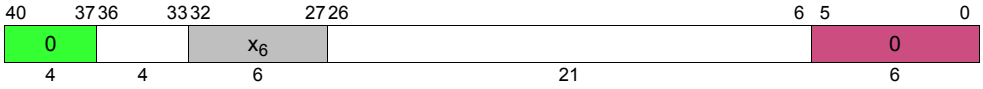
Instruction	Operands	Opcode	Extension		
			x_6	ih	wh
brp.*indwh.ih*	*b*$_2$, *tag*$_{13}$	2	10	See Table 4-56 on page 3:354	See Table 4-58 on page 3:355
brp.ret.*indwh.ih*			11		

4.5.3 Miscellaneous B-Unit Instructions

The miscellaneous branch-unit instructions include a number of instructions encoded within major opcode 0 using a 6-bit opcode extension field in bits 32:27 (x_6) as described in Table 4-48 on page 3:350.

4.5.3.1 Miscellaneous (B-Unit)

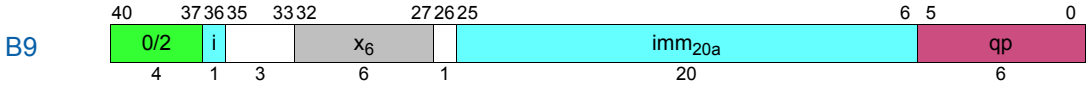
B8



Instruction	Opcode	Extension
		x_6
cover [l]	0	02
clrrrb [l]		04
clrrrb.pr [l]		05
rfi [e l p]		08
bsw.0 [l p]		0C
bsw.1 [l p]		0D
epc		10

Instruction	Opcode	Extension
		x_6
vmsw.0 [p]	0	18
vmsw.1 [p]		19

4.5.3.2 Break/Nop/Hint (B-Unit)



Instruction	Operands	Opcode	Extension
			x_6
break.b [e]	*imm*$_{21}$	0	00
nop.b		2	
hint.b			01

4.6 F-Unit Instruction Encodings

The floating-point instructions are encoded in major opcodes 8 – E for floating-point and fixed-point arithmetic, opcode 4 for floating-point compare, opcode 5 for floating-point class, and opcodes 0 and 1 for miscellaneous floating-point instructions.

The miscellaneous and reciprocal approximation floating-point instructions are encoded within major opcodes 0 and 1 using a 1-bit opcode extension field (x) in bit 33 and either a second 1-bit extension field in bit 36 (q) or a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-59 shows the 1-bit x assignments, Table 4-62 shows the additional 1-bit q assignments for the reciprocal approximation instructions; Table 4-60 and Table 4-61 summarize the 6-bit x_6 assignments.

Table 4-59. Miscellaneous Floating-point 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	
0	0	6-bit Ext (Table 4-60)
	1	Reciprocal Approximation (Table 4-62)
1	0	6-bit Ext (Table 4-61)
	1	Reciprocal Approximation (Table 4-62)

Table 4-60. Opcode 0 Miscellaneous Floating-point 6-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	x_6 Bits 30:27	x_6 Bits 32:31 = 0	x_6 Bits 32:31 = 1	x_6 Bits 32:31 = 2	x_6 Bits 32:31 = 3
0	0	0	break.f F15	fmerge.s F9		
		1	1-bit Ext (Table 4-68)	fmerge.ns F9		
		2		fmerge.se F9		
		3				
		4	fsetc F12	fmin F8		fswap F9
		5	fclrf F13	fmax F8		fswap.nl F9
		6		famin F8		fswap.nr F9
		7		famax F8		
		8	fchkf F14	fcvt.fx F10	fpack F9	
		9		fcvt.fxu F10		fmix.lr F9
		A		fcvt.fx.trunc F10		fmix.r F9
		B		fcvt.fxu.trunc F10		fmix.l F9
		C		fcvt.xf F11	fand F9	fsxt.r F9
		D			fandcm F9	fsxt.l F9
		E			for F9	
		F			fxor F9	

Table 4-61. Opcode 1 Miscellaneous Floating-point 6-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	x_6 Bits 30:27	x_6 Bits 32:31 = 0	x_6 Bits 32:31 = 1	x_6 Bits 32:31 = 2	x_6 Bits 32:31 = 3
1	0	0		fpmerge.s F9		fpcmp.eq F8
		1		fpmerge.ns F9		fpcmp.lt F8
		2		fpmerge.se F9		fpcmp.le F8
		3				fpcmp.unord F8
		4		fpmin F8		fpcmp.neq F8
		5		fpmax F8		fpcmp.nlt F8
		6		fpamin F8		fpcmp.nle F8
		7		fpamax F8		fpcmp.ord F8
		8		fpcvt.fx F10		
		9		fpcvt.fxu F10		
		A		fpcvt.fx.trunc F10		
		B		fpcvt.fxu.trunc F10		
		C				
		D				
		E				
		F				

Table 4-62. Reciprocal Approximation 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	q Bit 36	
0	1	0	frcpa F6
		1	frsqrta F7
1		0	fprcpa F6
		1	fprsqrta F7

Most floating-point instructions have a 2-bit opcode extension field in bits 35:34 (sf) which encodes the FPSR status field to be used. Table 4-63 summarizes these assignments.

Table 4-63. Floating-point Status Field Completer

sf Bits 35:34	*sf*
0	.s0
1	.s1
2	.s2
3	.s3

4.6.1 Arithmetic

The floating-point arithmetic instructions are encoded within major opcodes 8 – D using a 1-bit opcode extension field (x) in bit 36 and a 2-bit opcode extension field (sf) in bits 35:34. The opcode and x assignments are shown in Table 4-64.

Table 4-64. Floating-point Arithmetic 1-bit Opcode Extensions

x Bit 36	Opcode Bits 40:37					
	8	9	A	B	C	D
0	fma F1	fma.d F1	fms F1	fms.d F1	fnma F1	fnma.d F1
1	fma.s F1	fpma F1	fms.s F1	fpms F1	fnma.s F1	fpnma F1

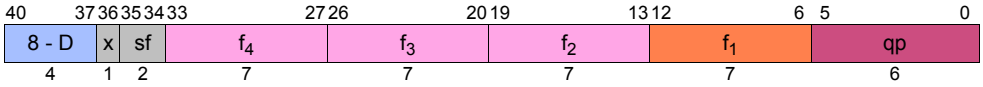
The fixed-point arithmetic and parallel floating-point select instructions are encoded within major opcode E using a 1-bit opcode extension field (x) in bit 36. The fixed-point arithmetic instructions also have a 2-bit opcode extension field (x_2) in bits 35:34. These assignments are shown in Table 4-65.

Table 4-65. Fixed-point Multiply Add and Select Opcode Extensions

Opcode Bits 40:37	x Bit 36	x_2 Bits 35:34			
		0	1	2	3
E	0	fselect F3			
	1	xma.l F2		xma.hu F2	xma.h F2

4.6.1.1 Floating-point Multiply Add

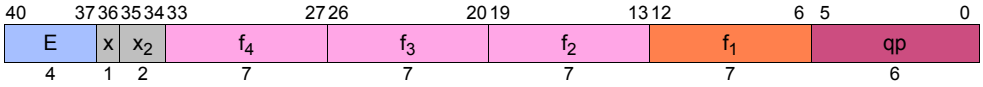
F1



Instruction	Operands	Opcode	Extension	
			x	sf
fma.*sf* fma.s.*sf* fma.d.*sf* fpma.*sf*	$f_1 = f_3, f_4, f_2$	8 9	0 1 0 1	See Table 4-63 on page 3:358
fms.*sf* fms.s.*sf* fms.d.*sf* fpms.*sf*		A B	0 1 0 1	
fnma.*sf* fnma.s.*sf* fnma.d.*sf* fpnma.*sf*		C D	0 1 0 1	

4.6.1.2 Fixed-point Multiply Add

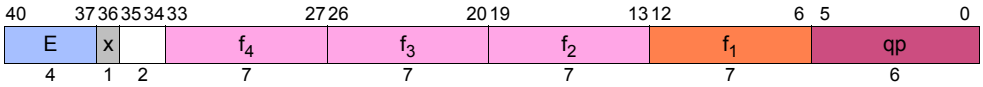
F2



Instruction	Operands	Opcode	Extension	
			x	x_2
xma.l xma.h xma.hu	$f_1 = f_3, f_4, f_2$	E	1	0 3 2

4.6.2 Parallel Floating-point Select

F3



Instruction	Operands	Opcode	Extension
			x
fselect	$f_1 = f_3, f_4, f_2$	E	0

4.6.3 Compare and Classify

The predicate setting floating-point compare instructions are encoded within major opcode 4 using three 1-bit opcode extension fields in bits 33 (r_a), 36 (r_b), and 12 (t_a), and a 2-bit opcode extension field (sf) in bits 35:34. The opcode, r_a, r_b, and t_a assignments are shown in Table 4-66. The sf assignments are shown in Table 4-63 on page 3:358.

The parallel floating-point compare instructions are described on page 3:362.

Table 4-66. Floating-point Compare Opcode Extensions

Opcode Bits 40:37	r_a Bit 33	r_b Bit 36	t_a Bit 12	
			0	1
4	0	0	fcmp.eq F4	fcmp.eq.unc F4
		1	fcmp.lt F4	fcmp.lt.unc F4
	1	0	fcmp.le F4	fcmp.le.unc F4
		1	fcmp.unord F4	fcmp.unord.unc F4

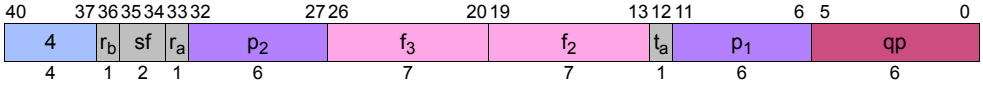
The floating-point class instructions are encoded within major opcode 5 using a 1-bit opcode extension field in bit 12 (t_a) as shown in Table 4-67.

Table 4-67. Floating-point Class 1-bit Opcode Extensions

Opcode Bits 40:37	t_a Bit 12	
5	0	fclass.m F5
	1	fclass.m.unc F5

4.6.3.1 Floating-point Compare

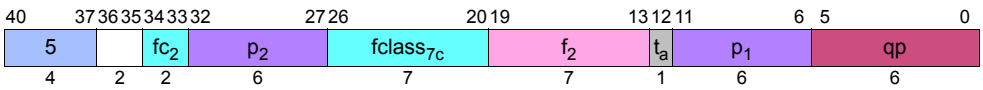
F4



Instruction	Operands	Opcode	Extension			
			r_a	r_b	t_a	sf
fcmp.eq.*sf*	$p_1, p_2 = f_2, f_3$	4	0	0	0	See Table 4-63 on page 3:358
fcmp.lt.*sf*				1		
fcmp.le.*sf*			1	0		
fcmp.unord.*sf*				1		
fcmp.eq.unc.*sf*			0	0	1	
fcmp.lt.unc.*sf*				1		
fcmp.le.unc.*sf*			1	0		
fcmp.unord.unc.*sf*				1		

4.6.3.2 Floating-point Class

F5



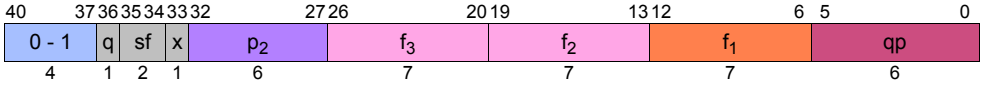
Instruction	Operands	Opcode	Extension
			t_a
fclass.m	$p_1, p_2 = f_2, fclass_9$	5	0
fclass.m.unc			1

4.6.4 Approximation

4.6.4.1 Floating-point Reciprocal Approximation

There are two Reciprocal Approximation instructions. The first, in major op 0, encodes the full register variant. The second, in major op 1, encodes the parallel variant.

F6

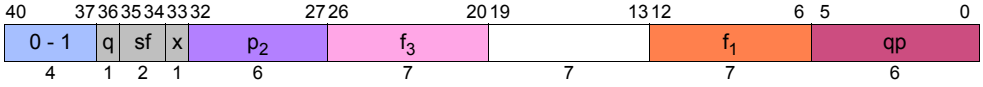


Instruction	Operands	Opcode	Extension		
			x	q	sf
frcpa.*sf* fprcpa.*sf*	$f_1, p_2 = f_2, f_3$	0 1	1	0	See Table 4-63 on page 3:358

4.6.4.2 Floating-point Reciprocal Square Root Approximation

There are two Reciprocal Square Root Approximation instructions. The first, in major op 0, encodes the full register variant. The second, in major op 1, encodes the parallel variant.

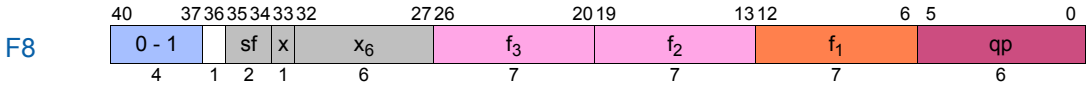
F7



Instruction	Operands	Opcode	Extension		
			x	q	sf
frsqrta.*sf* fprsqrta.*sf*	$f_1, p_2 = f_3$	0 1	1	1	See Table 4-63 on page 3:358

4.6.5 Minimum/Maximum and Parallel Compare

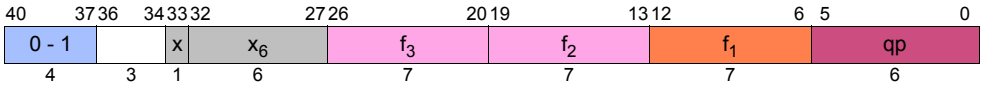
There are two groups of Minimum/Maximum instructions. The first group, in major op 0, encodes the full register variants. The second group, in major op 1, encodes the parallel variants. The parallel compare instructions are all encoded in major op 1.



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fmin.*sf*	$f_1 = f_2, f_3$	0	0	14	See Table 4-63 on page 3:358
fmax.*sf*				15	
famin.*sf*				16	
famax.*sf*				17	
fpmin.*sf*		1		14	
fpmax.*sf*				15	
fpamin.*sf*				16	
fpamax.*sf*				17	
fpcmp.eq.*sf*				30	
fpcmp.lt.*sf*				31	
fpcmp.le.*sf*				32	
fpcmp.unord.*sf*				33	
fpcmp.neq.*sf*				34	
fpcmp.nlt.*sf*				35	
fpcmp.nle.*sf*				36	
fpcmp.ord.*sf*				37	

4.6.6 Merge and Logical

F9

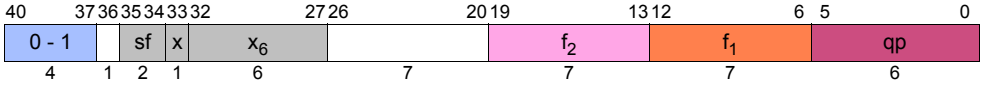


Instruction	Operands	Opcode	Extension	
			x	x_6
fmerge.s	$f_1 = f_2, f_3$	0	0	10
fmerge.ns				11
fmerge.se				12
fmix.lr				39
fmix.r				3A
fmix.l				3B
fsxt.r				3C
fsxt.l				3D
fpack				28
fswap				34
fswap.nl				35
fswap.nr				36
fand				2C
fandcm				2D
for				2E
fxor				2F
fpmerge.s		1		10
fpmerge.ns				11
fpmerge.se				12

4.6.7 Conversion

4.6.7.1 Convert Floating-point to Fixed-point

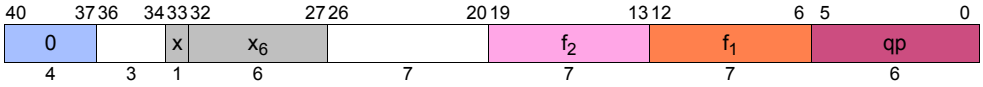
F10



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fcvt.fx.*sf*	$f_1 = f_2$	0	0	18	See Table 4-63 on page 3:358
fcvt.fxu.*sf*				19	
fcvt.fx.trunc.*sf*				1A	
fcvt.fxu.trunc.*sf*				1B	
fpcvt.fx.*sf*		1		18	
fpcvt.fxu.*sf*				19	
fpcvt.fx.trunc.*sf*				1A	
fpcvt.fxu.trunc.*sf*				1B	

4.6.7.2 Convert Fixed-point to Floating-point

F11

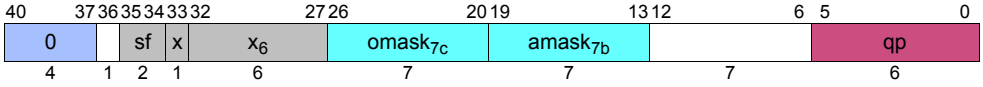


Instruction	Operands	Opcode	Extension	
			x	x_6
fcvt.xf	$f_1 = f_2$	0	0	1C

4.6.8 Status Field Manipulation

4.6.8.1 Floating-point Set Controls

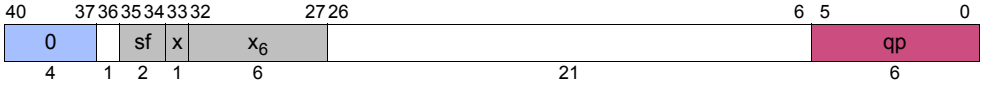
F12



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fsetc.*sf*	$amask_7$, $omask_7$	0	0	04	See Table 4-63 on page 3:358

4.6.8.2 Floating-point Clear Flags

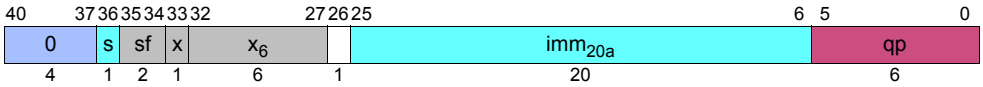
F13



Instruction	Opcode	Extension		
		x	x_6	sf
fclrf.*sf*	0	0	05	See Table 4-63 on page 3:358

4.6.8.3 Floating-point Check Flags

F14

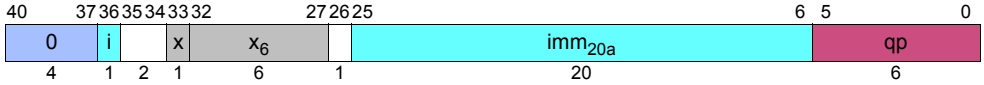


Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fchkf.*sf*	$target_{25}$	0	0	08	See Table 4-63 on page 3:358

4.6.9 Miscellaneous F-Unit Instructions

4.6.9.1 Break (F-Unit)

F15



Instruction	Operands	Opcode	Extension	
			x	x_6
break.f	imm_{21}	0	0	00

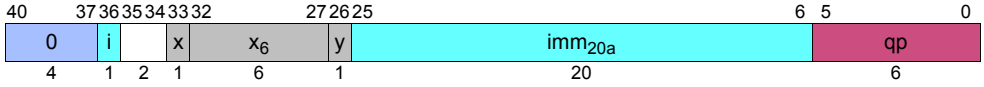
4.6.9.2 Nop/Hint (F-Unit)

F-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_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 4-46.

Table 4-68. Misc F-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit :33	x_6 Bits 32:27	y Bit 26	
0	0	01	0	nop.f
			1	hint.f

F16



Instruction	Operands	Opcode	Extension		
			x	x_6	y
nop.f hint.f	imm_{21}	0	0	01	0 1

4.7 X-Unit Instruction Encodings

The X-unit instructions occupy two instruction slots, L+X. The major opcode, opcode extensions and hints, qp, and small immediate fields occupy the X instruction slot. For movl, break.x, and nop.x, the imm_{41} field occupies the L instruction slot. For brl, the imm_{39} field and a 2-bit Ignored field occupy the L instruction slot.

4.7.1 Miscellaneous X-Unit Instructions

The miscellaneous X-unit instructions are encoded in major opcode 0 using a 3-bit opcode extension field (x_3) in bits 35:33 and a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-69 shows the 3-bit assignments and Table 4-70 summarizes the 6-bit assignments. These instructions are executed by an I-unit.

Table 4-69. Misc X-Unit 3-bit Opcode Extensions

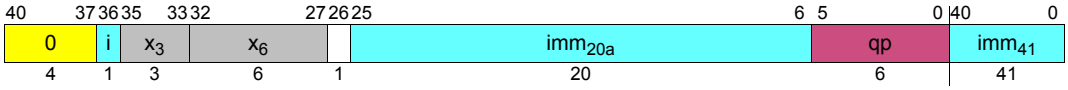
Opcode Bits 40:37	x_3 Bits 35:33	
0	0	6-bit Ext (Table 4-70)
	1	
	2	
	3	
	4	
	5	
	6	
	7	

Table 4-70. Misc X-Unit 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 30:27	x_6 Bits 32:31 = 0	1	2	3
0	0	0	break.x X1			
		1	1-bit Ext (Table 4-73)			
		2				
		3				
		4				
		5				
		6				
		7				
		8				
		9				
		A				
		B				
		C				
		D				
		E				
		F				

4.7.1.1 Break (X-Unit)

X1



Instruction	Operands	Opcode	Extension	
			x_3	x_6
break.x	*imm*$_{62}$	0	0	00

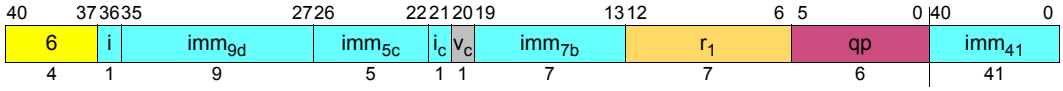
4.7.2 Move Long Immediate$_{64}$

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.

Table 4-71. Move Long 1-bit Opcode Extensions

Opcode Bits 40:37	v_c Bit 20	
6	0	movl X2
	1	

X2



Instruction	Operands	Opcode	Extension
			v_c
movl [i]	$r_1 = imm_{64}$	6	0

4.7.3 Long Branches

Long branches are executed by a B-unit. Opcode C is used for long branch and opcode D for long call.

The long branch instructions encoded within major opcode C use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-72.

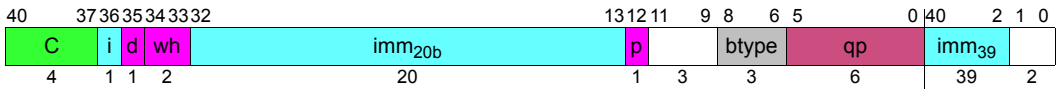
Table 4-72. Long Branch Types

Opcode Bits 40:37	btype Bits 8:6	
C	0	brl.cond X3
	1	
	2	
	3	
	4	
	5	
	6	
	7	

The long branch instructions have the same opcode hint fields in bit 12 (p), bits 34:33 (wh), and bit 35 (d) as normal IP-relative branches. These are shown in Table 4-51 on page 3:351, Table 4-52 on page 3:352, and Table 4-54 on page 3:352.

4.7.3.1 Long Branch

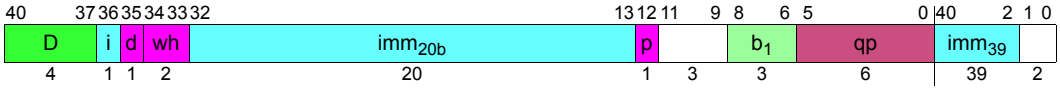
X3



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
brl.cond.*bwh.ph.dh* [e] [l]	$target_{64}$	C	0	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

4.7.3.2 Long Call

X4



Instruction	Operands	Opcode	Extension		
			p	wh	d
brl.call.*bwh.ph.dh* [e l]	$b_1 = target_{64}$	D	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

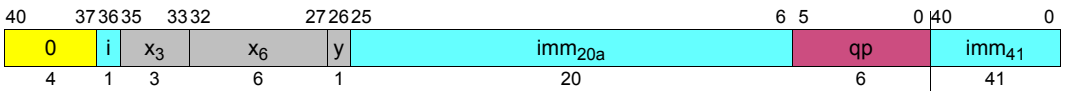
4.7.4 Nop/Hint (X-Unit)

X-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 6-bit opcode extension field in bits 32:27 (x_6), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-73. These instructions are executed by an I-unit.

Table 4-73. Misc X-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 32:27	y Bit 26	
0	0	01	0	nop.x
			1	hint.x

X5



Instruction	Operands	Opcode	Extension		
			x_3	x_6	y
nop.x	*imm*$_{62}$	0	0	01	0
hint.x					1

4.8 Immediate Formation

Table 4-74 shows, for each instruction format that has one or more immediates, how those immediates are formed. In each equation, the symbol to the left of the equals is the assembly language name for the immediate. The symbols to the right are the field names in the instruction encoding.

Table 4-74. Immediate Formation

Instruction Format	Immediate Formation
A2	$count_2 = ct_{2d} + 1$
A3 A8 I27 M30	$imm_8 = sign\_ext(s << 7 \mid imm_{7b}, 8)$
A4	$imm_{14} = sign\_ext(s << 13 \mid imm_{6d} << 7 \mid imm_{7b}, 14)$
A5	$imm_{22} = sign\_ext(s << 21 \mid imm_{5c} << 16 \mid imm_{9d} << 7 \mid imm_{7b}, 22)$
A10	$count_2 = (ct_{2d} > 2)$? reservedQP[a] : $ct_{2d} + 1$
I1	$count_2 = (ct_{2d} == 0)$? 0 : $(ct_{2d} == 1)$? 7 : $(ct_{2d} == 2)$? 15 : 16

Table 4-74. Immediate Formation (Continued)

Instruction Format	Immediate Formation
I3	$mbtype_4$ = (mbt_{4c} == 0) ? @brcst : (mbt_{4c} == 8) ? @mix : (mbt_{4c} == 9) ? @shuf : (mbt_{4c} == 0xA) ? @alt : (mbt_{4c} == 0xB) ? @rev : reservedQP[a]
I4	$mhtype_8$ = mht_{8c}
I6	$count_5$ = $count_{5b}$
I8	$count_5$ = 31 – $ccount_{5c}$
I10	$count_6$ = $count_{6d}$
I11	len_6 = len_{6d} + 1 pos_6 = pos_{6b}
I12	len_6 = len_{6d} + 1 pos_6 = 63 – $cpos_{6c}$
I13	len_6 = len_{6d} + 1 pos_6 = 63 – $cpos_{6c}$ imm_8 = sign_ext(s << 7 | imm_{7b}, 8)
I14	len_6 = len_{6d} + 1 pos_6 = 63 – $cpos_{6b}$ imm_1 = sign_ext(s, 1)
I15	len_4 = len_{4d} + 1 pos_6 = 63 – $cpos_{6d}$
I16	pos_6 = pos_{6b}
I18 I19 M37 M48	imm_{21} = i << 20 | imm_{20a}
I21	tag_{13} = IP + (sign_ext($timm_{9c}$, 9) << 4)
I23	$mask_{17}$ = sign_ext(s << 16 | $mask_{8c}$ << 8 | $mask_{7a}$ << 1, 17)
I24	imm_{44} = sign_ext(s << 43 | imm_{27a} << 16, 44)
I30	imm_5 = imm_{5b} + 32
M3 M8 M22	imm_9 = sign_ext(s << 8 | i << 7 | imm_{7b}, 9)
M5 M10	imm_9 = sign_ext(s << 8 | i << 7 | imm_{7a}, 9)
M17	inc_3 = sign_ext(((s) ? –1 : 1) * ((i_{2b} == 3) ? 1 : 1 << (4 – i_{2b})), 6)
I20 M20 M21	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{13c} << 7 | imm_{7a}, 21) << 4)
M22 M23	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20b}, 21) << 4)
M34	il = sol o = sof – sol r = sor << 3
M39 M40	imm_2 = i_{2b}
M44	imm_{24} = i << 23 | i_{2d} << 21 | imm_{21a}
B1 B2 B3	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20b}, 21) << 4)
B6	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20b}, 21) << 4) tag_{13} = IP + (sign_ext(t_{2e} << 7 | $timm_{7a}$, 9) << 4)
B7	tag_{13} = IP + (sign_ext(t_{2e} << 7 | $timm_{7a}$, 9) << 4)
B9	imm_{21} = i << 20 | imm_{20a}
F5	$fclass_9$ = $fclass_{7c}$ << 2 | fc_2
F12	$amask_7$ = $amask_{7b}$ $omask_7$ = $omask_{7c}$
F14	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20a}, 21) << 4)
F15 F16	imm_{21} = i << 20 | imm_{20a}
X1 X5	imm_{62} = imm_{41} << 21 | i << 20 | imm_{20a}
X2	imm_{64} = i << 63 | imm_{41} << 22 | i_c << 21 | imm_{5c} << 16 | imm_{9d} << 7 | imm_{7b}
X3 X4	$target_{64}$ = IP + ((i << 59 | imm_{39} << 20 | imm_{20b}) << 4)

a. This encoding causes an Illegal Operation fault if the value of the qualifying predicate is 1.

§

Resource and Dependency Semantics 5

5.1 Reading and Writing Resources

An Itanium instruction is said to be a **reader** of a resource if the instruction's qualifying predicate is 1 or it has no qualifying predicate or is one of the instructions that reads a resource even when its qualifying predicate is 0, and the execution of the instruction depends on that resource.

An Itanium instruction is said to be an **writer** of a resource if the instruction's qualifying predicate is 1 or it has no qualifying predicate or writes the resource even when the qualifying predicate is 0, and the execution of the instruction writes that resource.

An Itanium instruction is said to be a reader or writer of a resource even if it only sometimes depends on that resource and it cannot be determined statically whether the resource will be read or written. For example, `cover` only writes CR[IFS] when PSR.ic is 0, but for purposes of dependency, it is treated as if it always writes the resource since this condition cannot be determined statically. On the other hand, `rsm` conditionally writes several bits in the PSR depending on a mask which is encoded as an immediate in the instruction. Since the PSR bits to be written can be determined by examining the encoded instruction, the instruction is treated as only writing those bits which have a corresponding mask bit set. All exceptions to these general rules are described in this appendix.

5.2 Dependencies and Serialization

A **RAW** (Read-After-Write) dependency is a sequence of two events where the first is a writer of a resource and the second is a reader of the same resource. Events may be instructions, interruptions, or other 'uses' of the resource such as instruction stream fetches and VHPT walks. Table 5-2 covers only dependencies based on instruction readers and writers.

A **WAW** (Write-After-Write) dependency is a sequence of two events where both events write the resource in question. Events may be instructions, interruptions, or other 'updates' of the resource. Table 5-3 covers only dependencies based on instruction writers.

A **WAR** (Write-After-Read) dependency is a sequence of two instructions, where the first is a reader of a resource and the second is a writer of the same resource. Such dependencies are always allowed except as indicated in Table 5-4 and only those related to instruction readers and writers are included.

A **RAR** (Read-After-Read) dependency is a sequence of two instructions where both are readers of the same resource. Such dependencies are always allowed.

RAW and WAW dependencies are generally not allowed without some type of serialization event (an implied, data, or instruction serialization after the first writing instruction. (See Section 3.2, “Serialization” on page 2:17 for details on serialization.) The tables and associated rules in this appendix provide a comprehensive list of readers and writers of resources and describe the serialization required for the dependency to be observed and possible outcomes if the required serialization is not met. Even when targeting code for machines which do not check for particular disallowed dependencies, such code sequences are considered architecturally undefined and may cause code to behave differently across processors, operating systems, or even separate executions of the code sequence during the same program run. In some cases, different serializations may yield different, but well-defined results.

The serialization of application level (non-privileged) resources is always implied. This means that if a writer of that resource and a subsequent read of that same resource are in different instruction groups, then the reader will see the value written. In addition, for dependencies on PRs and BRs, where the writer is a non-branch instruction and the reader is a branch instruction, the writer and reader may be in the same instruction group.

System resources generally require explicit serialization, i.e., the use of a `srlz.i` or `srlz.d` instruction, between the writing and the reading of that resource. Note that RAW accesses to CRs are not exceptional – they require explicit data or instruction serialization. However, in some cases (other than CRs) where pairs of instructions explicitly encode the same resource, serialization is implied.

There are cases where it is architecturally allowed to omit a serialization, and that the response from the CPU must be atomic (act as if either the old or the new state were fully in place). The tables in this appendix indicate dependency requirements under the assumption that the desired result is for the dependency to always be observed. In some such cases, the programmer may not care if the old or new state is used; such situations are allowed, but the value seen is not deterministic.

On the other hand, if an *impliedF* dependency is violated, then the program is incorrectly coded and the processor's behavior is undefined.

5.3 Resource and Dependency Table Format Notes

- The “Writers” and “Readers” columns of the dependency tables contain instruction class names and instruction mnemonic prefixes as given in the format section of each instruction page. To avoid ambiguity, instruction classes are shown in bold, while instruction mnemonic prefixes are in regular font. For instruction mnemonic prefixes, all instructions that exactly match the name specified or those that begin with the specified text and are followed by a ‘.’ and then followed by any other text will match.
- The dependency on a listed instruction is in effect no matter what values are encoded in the instruction or what dynamic values occur in operands, unless a superscript is present or one of the special case instruction rules in Section 5.3.1 applies. Instructions listed are still subject to rules regarding qualifying predicates.
- Instruction classes are groups of related instructions. Such names appear in boldface for clarity. The list of all instruction classes is contained in Table 5-5. Note that an instruction may appear in multiple instruction classes, instruction classes

may expand to contain other classes, and that when fully expanded, a set of classes (e.g., the readers of some resource) may contain the same instruction multiple times.

- The syntax '**x****y**' where **x** and **y** are both instruction classes, indicates an unnamed instruction class that includes all instructions in instruction class **x** but that are not in instruction class **y**. Similarly, the notation '**x****y****z**' means all instructions in instruction class **x**, but that are not in either instruction class **y** or instruction class **z**.
- Resources on separate rows of a table are independent resources. This means that there are no serialization requirements for an event which references one of them followed by an event which uses a different resource. In cases where resources are broken into subrows, dependencies only apply between instructions within a subrow. Instructions that do not appear in a subrow together have no dependencies (reader/writer or writer/writer dependencies) for the resource in question, although they may still have dependencies on some other resource.
- The dependencies listed for pairs of instructions on each resource are not unique – the same pair of instructions might also have a dependency on some other resource with a different semantics of dependency. In cases where there are multiple resource dependencies for the same pair of instructions, the most stringent semantics are assumed: *instr* overrides *data* which overrides *impliedF* which overrides *implied* which overrides *none*.
- Arrays of numbered resources are represented in a single row of a table using the % notation as a substitute for the number of the resource. In such cases, the semantics of the table are as if each numbered resource had its own row in that table and is thus an independent resource. The range of values that the % can take are given in the "Resource Name" column.
- An asterisk '*' in the "Resource Name" column indicates that this resource may not have a physical resource associated with it, but is added to enforce special dependencies.
- A pound sign '#' in the "Resource Name" column indicates that this resource is an array of resources that are indexed by a value in a GR. The number of individual elements in the array is described in the detailed description of each resource.
- The "Semantics of Dependency" column describes the outcome given various serialization and instruction group boundary conditions. The exact definition for each keyword is given in Table 5-1.

Table 5-1. Semantics of Dependency Codes

Semantics of Dependency Code	Serialization Type Required	Effects of Serialization Violation
instr	Instruction Serialization (See "Instruction Serialization" on page 2:18).	Atomic: Any attempt to read a resource after one or more insufficiently serialized writes is either the value previously in the register (before any of the unserialized writes) or the value of one of any unserialized writes. Which value is returned is unpredictable and multiple insufficiently serialized reads may see different results. No fault will be caused by the insufficient serialization.
data	Data Serialization (See "Data Serialization" on page 2:18)	
implied	Instruction Group Break. Writer and reader must be in separate instruction groups. (See "Instruction Sequencing Considerations" on page 1:39).	

Table 5-1. Semantics of Dependency Codes (Continued)

Semantics of Dependency Code	Serialization Type Required	Effects of Serialization Violation
impliedF	Instruction Group Break (same as above).	An undefined value is returned, or an Illegal Operation fault may be taken. If no fault is taken, the value returned is unpredictable, and may be unrelated to past writes, but will not be data which could not be accessed by the current process (e.g., if PSR.cpl != 0, the undefined value to return cannot be read from some control register).
stop	Stop. Writer and reader must be separated by a stop.	
none	None	N/A
specific	Implementation Specific	
SC	Special Case	Described elsewhere in book, see referenced section in the entry.

5.3.1 Special Case Instruction Rules

The following rules apply to the specified instructions when they appear in Table 5-2, Table 5-3, Table 5-4, or Table 5-5:

- An instruction always reads a given resource if its qualifying predicate is 1 and it appears in the "Reader" column of the table (except as noted). An instruction always writes a given resource if its qualifying predicate is 1 and it appears in the "Writer" column of the table (except as noted). An instruction never reads or writes the specified resource if its qualifying predicate is 0 (except as noted). These rules include branches and their qualifying predicate. Instructions in the **unpredicatable-instructions** class have no qualifying predicate and thus always read or write their resources (except as noted).
- An instruction of type **mov-from-PR** reads all PRs if its PR[qp] is true. If the PR[qp] is false, then only the PR[qp] is read.
- An instruction of type **mov-to-PR** writes only those PRs as indicated by the immediate mask encoded in the instruction.
- A `st8.spill` only writes AR[UNAT]{*X*} where *X* equals the value in bits 8:3 of the store's data address. A `ld8.fill` instruction only reads AR[UNAT]{*Y*} where *Y* equals the value in bits 8:3 of the load's data address.
- Instructions of type **mod-sched-brs** always read AR[EC] and the rotating register base registers in CFM, and always write AR[EC], the rotating register bases in CFM, and PR[63] even if they do not change their values or if their PR[qp] is false.
- Instructions of type **mod-sched-brs-counted** always read and write AR[LC], even if they do not change its value.
- For instructions of type **pr-or-writers** or **pr-and-writers**, if their completer is `or.andcm`, then only the first target predicate is an or-compare and the second target predicate is an and-compare. Similarly, if their completer is `and.orcm`, then only the second target predicate is an or-compare and the first target predicate is an and-compare.
- `rum` and `sum` only read PSR.sp when the bit corresponding to PSR.up (bit 2) is set in the immediate field of the instruction.

5.3.2 RAW Dependency Table

Table 5-2 architecturally defines the following information:

- A list of all architecturally-defined, independently-writable resources in the Itanium architecture. Each row represents an 'atomic' resource. Thus, for each row in the table, hardware will probably require a separate write-enable control signal.
- For each resource, a complete list of readers and writers.
- For each instruction, a complete list of all resources read and written. Such a list can be obtained by taking the union of all the rows in which each instruction appears.

Table 5-2. RAW Dependencies Organized by Resource

Resource Name	Writers	Readers	Semantics of Dependency
ALAT	chk.a.clr, **mem-readers-alat**, **mem-writers**, **invala-all**	**mem-readers-alat**, **mem-writers**, **chk-a**, invala.e	none
AR[BSP]	br.call, brl.call, br.ret, cover, **mov-to-AR-BSPSTORE**, rfi	br.call, brl.call, br.ia, br.ret, cover, flushrs, loadrs, **mov-from-AR-BSP**, rfi	impliedF
AR[BSPSTORE]	alloc, loadrs, flushrs, **mov-to-AR-BSPSTORE**	alloc, br.ia, flushrs, **mov-from-AR-BSPSTORE**	impliedF
AR[CCV]	**mov-to-AR-CCV**	br.ia, **cmpxchg**, **mov-from-AR-CCV**	impliedF
AR[CFLG]	**mov-to-AR-CFLG**	br.ia, **mov-from-AR-CFLG**	impliedF
AR[CSD]	ld16, **mov-to-AR-CSD**	br.ia, cmp8xchg16, **mov-from-AR-CSD**, st16	impliedF
AR[EC]	**mod-sched-brs**, br.ret, **mov-to-AR-EC**	br.call, brl.call, br.ia, **mod-sched-brs**, **mov-from-AR-EC**	impliedF
AR[EFLAG]	**mov-to-AR-EFLAG**	br.ia, **mov-from-AR-EFLAG**	impliedF
AR[FCR]	**mov-to-AR-FCR**	br.ia, **mov-from-AR-FCR**	impliedF
AR[FDR]	**mov-to-AR-FDR**	br.ia, **mov-from-AR-FDR**	impliedF
AR[FIR]	**mov-to-AR-FIR**	br.ia, **mov-from-AR-FIR**	impliedF
AR[FPSR].sf0.controls	**mov-to-AR-FPSR**, fsetc.s0	br.ia, **fp-arith-s0**, **fcmp-s0**, **fpcmp-s0**, fsetc, **mov-from-AR-FPSR**	impliedF
AR[FPSR].sf1.controls	**mov-to-AR-FPSR**, fsetc.s1	br.ia, **fp-arith-s1**, **fcmp-s1**, **fpcmp-s1**, **mov-from-AR-FPSR**	
AR[FPSR].sf2.controls	**mov-to-AR-FPSR**, fsetc.s2	br.ia, **fp-arith-s2**, **fcmp-s2**, **fpcmp-s2**, **mov-from-AR-FPSR**	
AR[FPSR].sf3.controls	**mov-to-AR-FPSR**, fsetc.s3	br.ia, **fp-arith-s3**, **fcmp-s3**, **fpcmp-s3**, **mov-from-AR-FPSR**	
AR[FPSR].sf0.flags	**fp-arith-s0**, fclrf.s0, **fcmp-s0**, **fpcmp-s0**, **mov-to-AR-FPSR**	br.ia, fchkf, **mov-from-AR-FPSR**	impliedF
AR[FPSR].sf1.flags	**fp-arith-s1**, fclrf.s1, **fcmp-s1**, **fpcmp-s1**, **mov-to-AR-FPSR**	br.ia, fchkf.s1, **mov-from-AR-FPSR**	
AR[FPSR].sf2.flags	**fp-arith-s2**, fclrf.s2, **fcmp-s2**, **fpcmp-s2**, **mov-to-AR-FPSR**	br.ia, fchkf.s2, **mov-from-AR-FPSR**	
AR[FPSR].sf3.flags	**fp-arith-s3**, fclrf.s3, **fcmp-s3**, **fpcmp-s3**, **mov-to-AR-FPSR**	br.ia, fchkf.s3, **mov-from-AR-FPSR**	
AR[FPSR].traps	**mov-to-AR-FPSR**	br.ia, **fp-arith**, fchkf, fcmp, fpcmp, **mov-from-AR-FPSR**	impliedF
AR[FPSR].rv	**mov-to-AR-FPSR**	br.ia, **fp-arith**, fchkf, fcmp, fpcmp, **mov-from-AR-FPSR**	impliedF
AR[FSR]	**mov-to-AR-FSR**	br.ia, **mov-from-AR-FSR**	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
AR[ITC]	**mov-to-AR-ITC**	br.ia, **mov-from-AR-ITC**	impliedF
AR[K%], % in 0 - 7	**mov-to-AR-K**[1]	br.ia, **mov-from-AR-K**[1]	impliedF
AR[LC]	**mod-sched-brs-counted**, **mov-to-AR-LC**	br.ia, **mod-sched-brs-counted**, **mov-from-AR-LC**	impliedF
AR[PFS]	br.call, brl.call	alloc, br.ia, br.ret, epc, **mov-from-AR-PFS**	impliedF
	mov-to-AR-PFS	alloc, br.ia, epc, **mov-from-AR-PFS**	impliedF
		br.ret	none
AR[RNAT]	alloc, flushrs, loadrs, **mov-to-AR-RNAT**, **mov-to-AR-BSPSTORE**	alloc, br.ia, flushrs, loadrs, **mov-from-AR-RNAT**	impliedF
AR[RSC]	**mov-to-AR-RSC**	alloc, br.ia, flushrs, loadrs, **mov-from-AR-RSC**, **mov-from-AR-BSPSTORE**, **mov-to-AR-RNAT**, **mov-from-AR-RNAT**, **mov-to-AR-BSPSTORE**	impliedF
AR[RUC]	**mov-to-AR-RUC**	br.ia, **mov-from-AR-RUC**	impliedF
AR[SSD]	**mov-to-AR-SSD**	br.ia, **mov-from-AR-SSD**	impliedF
AR[UNAT]{%}, % in 0 - 63	**mov-to-AR-UNAT**, st8.spill	br.ia, ld8.fill, **mov-from-AR-UNAT**	impliedF
AR%, % in 8-15, 20, 22-23, 31, 33-35, 37-39, 41-43, 46-47, 67-111	**none**	br.ia, **mov-from-AR-rv**[1]	none
AR%, % in 48-63, 112-127	**mov-to-AR-ig**[1]	br.ia, **mov-from-AR-ig**[1]	impliedF
BR%, % in 0 - 7	br.call[1], brl.call[1]	**indirect-brs**[1], **indirect-brp**[1], **mov-from-BR**[1]	impliedF
	mov-to-BR[1]	**indirect-brs**[1]	none
		indirect-brp[1], **mov-from-BR**[1]	impliedF
CFM	**mod-sched-brs**	**mod-sched-brs**	impliedF
		cover, alloc, rfi, loadrs, br.ret, br.call, brl.call	impliedF
		cfm-readers[2]	impliedF
	br.call, brl.call, br.ret, clrrrb, cover, rfi	**cfm-readers**	impliedF
	alloc	**cfm-readers**	none
CPUID#	**none**	**mov-from-IND-CPUID**[3]	specific
CR[CMCV]	**mov-to-CR-CMCV**	**mov-from-CR-CMCV**	data
CR[DCR]	**mov-to-CR-DCR**	**mov-from-CR-DCR**, **mem-readers-spec**	data

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
CR[EOI]	**mov-to-CR-EOI**	**none**	SC Section 5.8.3.4, “End of External Interrupt Register (EOI – CR67)” on page 2:124
CR[IFA]	**mov-to-CR-IFA**	itc.i, itc.d, itr.i, itr.d	implied
		mov-from-CR-IFA	data
CR[IFS]	**mov-to-CR-IFS**	**mov-from-CR-IFS**	data
		rfi	implied
	cover	rfi, **mov-from-CR-IFS**	implied
CR[IHA]	**mov-to-CR-IHA**	**mov-from-CR-IHA**	data
CR[IIB%], % in 0 - 1	**mov-to-CR-IIB**	**mov-from-CR-IIB**	data
CR[IIM]	**mov-to-CR-IIM**	**mov-from-CR-IIM**	data
CR[IIP]	**mov-to-CR-IIP**	**mov-from-CR-IIP**	data
		rfi	implied
CR[IIPA]	**mov-to-CR-IIPA**	**mov-from-CR-IIPA**	data
CR[IPSR]	**mov-to-CR-IPSR**	**mov-from-CR-IPSR**	data
		rfi	implied
CR[IRR%], % in 0 - 3	**mov-from-CR-IVR**	**mov-from-CR-IRR**[1]	data
CR[ISR]	**mov-to-CR-ISR**	**mov-from-CR-ISR**	data
CR[ITIR]	**mov-to-CR-ITIR**	**mov-from-CR-ITIR**	data
		itc.i, itc.d, itr.i, itr.d	implied
CR[ITM]	**mov-to-CR-ITM**	**mov-from-CR-ITM**	data
CR[ITO]	**mov-to-CR-ITO**	**mov-from-AR-ITC**, **mov-from-CR-ITO**	data
CR[ITV]	**mov-to-CR-ITV**	**mov-from-CR-ITV**	data
CR[IVA]	**mov-to-CR-IVA**	**mov-from-CR-IVA**	instr
CR[IVR]	**none**	**mov-from-CR-IVR**	SC Section 5.8.3.2, “External Interrupt Vector Register (IVR – CR65)” on page 2:123
CR[LID]	**mov-to-CR-LID**	**mov-from-CR-LID**	SC Section 5.8.3.1, “Local ID (LID – CR64)” on page 2:122
CR[LRR%], % in 0 - 1	**mov-to-CR-LRR**[1]	**mov-from-CR-LRR**[1]	data
CR[PMV]	**mov-to-CR-PMV**	**mov-from-CR-PMV**	data
CR[PTA]	**mov-to-CR-PTA**	**mov-from-CR-PTA**, **mem-readers**, **mem-writers**, **non-access**, thash	data

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
CR[TPR]	**mov-to-CR-TPR**	**mov-from-CR-TPR**, **mov-from-CR-IVR**	data
		mov-to-PSR-l[17], ssm[17]	SC Section 5.8.3.3, “Task Priority Register (TPR – CR66)” on page 2:123
		rfi	implied
CR%, % in 3, 5-7, 10-15, 18, 28-63, 75-79, 82-127	**none**	**mov-from-CR-rv**[1]	none
DBR#	**mov-to-IND-DBR**[3]	**mov-from-IND-DBR**[3]	impliedF
		probe-all, **lfetch-all**, **mem-readers**, **mem-writers**	data
DTC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	**mem-readers**, **mem-writers**, **non-access**	data
	itc.i, itc.d, itr.i, itr.d	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	impliedF
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	none
		itc.i, itc.d, itr.i, itr.d	impliedF
DTC_LIMIT*	ptc.g, ptc.ga	ptc.g, ptc.ga	impliedF
DTR	itr.d	**mem-readers**, **mem-writers**, **non-access**	data
		ptc.g, ptc.ga, ptc.l, ptr.d, itr.d	impliedF
	ptr.d	**mem-readers**, **mem-writers**, **non-access**	data
		ptc.g, ptc.ga, ptc.l, ptr.d	none
		itr.d, itc.d	impliedF
FR%, % in 0 - 1	**none**	**fr-readers**[1]	none
FR%, % in 2 - 127	**fr-writers**[1]**ldf-c**[1]**ldfp-c**[1]	**fr-readers**[1]	impliedF
	ldf-c[1], **ldfp-c**[1]	**fr-readers**[1]	none
GR0	**none**	**gr-readers**[1]	none
GR%, % in 1 - 127	**ld-c**[1,13]	**gr-readers**[1]	none
	gr-writers[1]**ld-c**[1,13]	**gr-readers**[1]	impliedF
IBR#	**mov-to-IND-IBR**[3]	**mov-from-IND-IBR**[3]	impliedF
InService*	**mov-to-CR-EOI**	**mov-from-CR-IVR**	data
	mov-from-CR-IVR	**mov-from-CR-IVR**	impliedF
	mov-to-CR-EOI	**mov-to-CR-EOI**	impliedF
IP	**all**	**all**	none
ITC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	epc, vmsw	instr
		itc.i, itc.d, itr.i, itr.d	impliedF
		ptr.i, ptr.d, ptc.e, ptc.g, ptc.ga, ptc.l	none
	itc.i, itc.d, itr.i, itr.d	epc, vmsw	instr
		itc.d, itc.i, itr.d, itr.i, ptr.d, ptr.i, ptc.g, ptc.ga, ptc.l	impliedF
ITC_LIMIT*	ptc.g, ptc.ga	ptc.g, ptc.ga	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
ITR	itr.i	itr.i, itc.i, ptc.g, ptc.ga, ptc.l, ptr.i	impliedF
		epc, vmsw	instr
	ptr.i	itc.i, itr.i	impliedF
		ptc.g, ptc.ga, ptc.l, ptr.i	none
		epc, vmsw	instr
memory	**mem-writers**	**mem-readers**	none
PKR#	**mov-to-IND-PKR**[3]	**mem-readers**, **mem-writers**, **mov-from-IND-PKR**[4], **probe-all**	data
		mov-to-IND-PKR[4]	none
		mov-from-IND-PKR[3]	impliedF
		mov-to-IND-PKR[3]	impliedF
PMC#	**mov-to-IND-PMC**[3]	**mov-from-IND-PMC**[3]	impliedF
		mov-from-IND-PMD[3]	SC Section 7.2.1, “Generic Performance Counter Registers” for PMC[0].fr on page 2:156
PMD#	**mov-to-IND-PMD**[3]	**mov-from-IND-PMD**[3]	impliedF
PR0	**pr-writers**[1]	**pr-readers-br**[1], **pr-readers-nobr-nomovpr**[1], **mov-from-PR**[12], **mov-to-PR**[12]	none
PR%, % in 1 - 15	**pr-writers**[1], **mov-to-PR-allreg**[7]	**pr-readers-nobr-nomovpr**[1], **mov-from-PR**, **mov-to-PR**[12]	impliedF
	pr-writers-fp[1]	**pr-readers-br**[1]	impliedF
	pr-writers-int[1], **mov-to-PR-allreg**[7]	**pr-readers-br**[1]	none
PR%, % in 16 - 62	**pr-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-nobr-nomovpr**[1], **mov-from-PR**, **mov-to-PR**[12]	impliedF
	pr-writers-fp[1]	**pr-readers-br**[1]	impliedF
	pr-writers-int[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-br**[1]	none
PR63	**mod-sched-brs**, **pr-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-nobr-nomovpr**[1], **mov-from-PR**, **mov-to-PR**[12]	impliedF
	pr-writers-fp[1], **mod-sched-brs**	**pr-readers-br**[1]	impliedF
	pr-writers-int[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-br**[1]	none

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
PSR.ac	**user-mask-writers-partial**[7], **mov-to-PSR-um**	**mem-readers**, **mem-writers**	implied
	sys-mask-writers-partial[7], **mov-to-PSR-l**	**mem-readers**, **mem-writers**	data
	user-mask-writers-partial[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**, **mov-from-PSR-um**	impliedF
	rfi	**mem-readers**, **mem-writers**, **mov-from-PSR**, **mov-from-PSR-um**	impliedF
PSR.be	**user-mask-writers-partial**[7], **mov-to-PSR-um**	**mem-readers**, **mem-writers**	implied
	sys-mask-writers-partial[7], **mov-to-PSR-l**	**mem-readers**, **mem-writers**	data
	user-mask-writers-partial[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**, **mov-from-PSR-um**	impliedF
	rfi	**mem-readers**, **mem-writers**, **mov-from-PSR**, **mov-from-PSR-um**	impliedF
PSR.bn	bsw, rfi	**gr-readers**[10], **gr-writers**[10]	impliedF
PSR.cpl	epc, br.ret	**priv-ops**, br.call, brl.call, epc, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-to-AR-ITC**, **mov-to-AR-RSC**, **mov-to-AR-RUC**, **mov-to-AR-K**, **mov-from-IND-PMD**, **probe-all**, **mem-readers**, **mem-writers**, **lfetch-all**	implied
	rfi	**priv-ops**, br.call, brl.call, epc, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-to-AR-ITC**, **mov-to-AR-RSC**, **mov-to-AR-RUC**, **mov-to-AR-K**, **mov-from-IND-PMD**, **probe-all**, **mem-readers**, **mem-writers**, **lfetch-all**	impliedF
PSR.da	rfi	**mem-readers**, **lfetch-all**, **mem-writers**, **probe-fault**	impliedF
PSR.db	**mov-to-PSR-l**	**lfetch-all**, **mem-readers**, **mem-writers**, **probe-fault**	data
		mov-from-PSR	impliedF
	rfi	**lfetch-all**, **mem-readers**, **mem-writers**, **mov-from-PSR**, **probe-fault**	impliedF
PSR.dd	rfi	**lfetch-all**, **mem-readers**, **probe-fault**, **mem-writers**	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
PSR.dfh	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**fr-readers**[8], **fr-writers**[8]	data
		mov-from-PSR	impliedF
	rfi	**fr-readers**[8], **fr-writers**[8], **mov-from-PSR**	impliedF
PSR.dfl	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**fr-writers**[8], **fr-readers**[8]	data
		mov-from-PSR	impliedF
	rfi	**fr-writers**[8], **fr-readers**[8], **mov-from-PSR**	impliedF
PSR.di	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	br.ia	data
		mov-from-PSR	impliedF
	rfi	br.ia, **mov-from-PSR**	impliedF
PSR.dt	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mem-readers**, **mem-writers**, **non-access**	data
		mov-from-PSR	impliedF
	rfi	**mem-readers**, **mem-writers**, **non-access**, **mov-from-PSR**	impliedF
PSR.ed	rfi	**lfetch-all**, **mem-readers-spec**	impliedF
PSR.i	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR**	impliedF
PSR.ia	rfi	**all**	none
PSR.ic	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**	impliedF
		cover, itc.i, itc.d, itr.i, itr.d, **mov-from-interruption-CR**, **mov-to-interruption-CR**	data
	rfi	**mov-from-PSR**, cover, itc.i, itc.d, itr.i, itr.d, **mov-from-interruption-CR**, **mov-to-interruption-CR**	impliedF
PSR.id	rfi	**all**	none
PSR.is	br.ia, rfi	**none**	none
PSR.it	rfi	**branches**, **mov-from-PSR**, chk, epc, fchkf, vmsw	impliedF
PSR.lp	**mov-to-PSR-l**	**mov-from-PSR**	impliedF
		br.ret	data
	rfi	**mov-from-PSR**, br.ret	impliedF
PSR.mc	rfi	**mov-from-PSR**	impliedF
PSR.mfh	**fr-writers**[9], **user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR-um**, **mov-from-PSR**	impliedF
PSR.mfl	**fr-writers**[9], **user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR-um**, **mov-from-PSR**	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
PSR.pk	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**lfetch-all**, **mem-readers**, **mem-writers**, **probe-all**	data
		mov-from-PSR	impliedF
	rfi	**lfetch-all**, **mem-readers**, **mem-writers**, **mov-from-PSR**, **probe-all**	impliedF
PSR.pp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR**	impliedF
PSR.ri	rfi	**all**	none
PSR.rt	**mov-to-PSR-l**	**mov-from-PSR**	impliedF
		alloc, flushrs, loadrs	data
	rfi	**mov-from-PSR**, alloc, flushrs, loadrs	impliedF
PSR.si	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**	impliedF
		mov-from-AR-ITC, **mov-from-AR-RUC**	data
	rfi	**mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-from-PSR**	impliedF
PSR.sp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**	impliedF
		mov-from-IND-PMD, **mov-to-PSR-um**, rum, sum	data
	rfi	**mov-from-IND-PMD**, **mov-from-PSR**, **mov-to-PSR-um**, rum, sum	impliedF
PSR.ss	rfi	**all**	impliedF
PSR.tb	**mov-to-PSR-l**	**branches**, chk, fchkf	data
		mov-from-PSR	impliedF
	rfi	**branches**, chk, fchkf, **mov-from-PSR**	impliedF
PSR.up	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR-um**, **mov-from-PSR**	impliedF
PSR.vm	vmsw	**mem-readers**, **mem-writers**, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-from-IND-CPUID**, **mov-to-AR-ITC**, **mov-to-AR-RUC**, **priv-ops**\vmsw, cover, thash, ttag	implied
	rfi	**mem-readers**, **mem-writers**, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-from-IND-CPUID**, **mov-to-AR-ITC**, **mov-to-AR-RUC**, **priv-ops**\vmsw, cover, thash, ttag	impliedF
RR#	**mov-to-IND-RR**[6]	**mem-readers**, **mem-writers**, itc.i, itc.d, itr.i, itr.d, **non-access**, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, thash, ttag	data
		mov-from-IND-RR[6]	impliedF
RSE	**rse-writers**[14]	**rse-readers**[14]	impliedF

5.3.3 WAW Dependency Table

General rules specific to the WAW table:

- All resources require at most an instruction group break to provide sequential behavior.
- Some resources require no instruction group break to provide sequential behavior.
- There are a few special cases that are described in greater detail elsewhere in the manual and are indicated with an SC (special case) result.
- Each sub-row of writers represents a group of instructions that when taken in pairs in any combination has the dependency result indicated. If the column is split in sub-columns, then the dependency semantics apply to any pair of instructions where one is chosen from left sub-column and one is chosen from the right sub-column.

Table 5-3. WAW Dependencies Organized by Resource

Resource Name	Writers		Semantics of Dependency
ALAT	**mem-readers-alat**, **mem-writers**, chk.a.clr, **invala-all**		none
AR[BSP]	br.call, brl.call, br.ret, cover, **mov-to-AR-BSPSTORE**, rfi		impliedF
AR[BSPSTORE]	alloc, loadrs, flushrs, **mov-to-AR-BSPSTORE**		impliedF
AR[CCV]	**mov-to-AR-CCV**		impliedF
AR[CFLG]	**mov-to-AR-CFLG**		impliedF
AR[CSD]	ld16, **mov-to-AR-CSD**		impliedF
AR[EC]	br.ret, **mod-sched-brs**, **mov-to-AR-EC**		impliedF
AR[EFLAG]	**mov-to-AR-EFLAG**		impliedF
AR[FCR]	**mov-to-AR-FCR**		impliedF
AR[FDR]	**mov-to-AR-FDR**		impliedF
AR[FIR]	**mov-to-AR-FIR**		impliedF
AR[FPSR].sf0.controls	**mov-to-AR-FPSR**, fsetc.s0		impliedF
AR[FPSR].sf1.controls	**mov-to-AR-FPSR**, fsetc.s1		impliedF
AR[FPSR].sf2.controls	**mov-to-AR-FPSR**, fsetc.s2		impliedF
AR[FPSR].sf3.controls	**mov-to-AR-FPSR**, fsetc.s3		impliedF
AR[FPSR].sf0.flags	**fp-arith-s0**, **fcmp-s0**, **fpcmp-s0**		none
	fclrf.s0, **fcmp-s0**, **fp-arith-s0**, **fpcmp-s0**, **mov-to-AR-FPSR**	fclrf.s0, **mov-to-AR-FPSR**	impliedF
AR[FPSR].sf1.flags	**fp-arith-s1**, **fcmp-s1**, **fpcmp-s1**		none
	fclrf.s1, **fcmp-s1**, **fp-arith-s1**, **fpcmp-s1**, **mov-to-AR-FPSR**	fclrf.s1, **mov-to-AR-FPSR**	impliedF
AR[FPSR].sf2.flags	**fp-arith-s2**, **fcmp-s2**, **fpcmp-s2**		none
	fclrf.s2, **fcmp-s2**, **fp-arith-s2**, **fpcmp-s2**, **mov-to-AR-FPSR**	fclrf.s2, **mov-to-AR-FPSR**	impliedF
AR[FPSR].sf3.flags	**fp-arith-s3**, **fcmp-s3**, **fpcmp-s3**		none
	fclrf.s3, **fcmp-s3**, **fp-arith-s3**, **fpcmp-s3**, **mov-to-AR-FPSR**	fclrf.s3, **mov-to-AR-FPSR**	impliedF
AR[FPSR].rv	**mov-to-AR-FPSR**		impliedF
AR[FPSR].traps	**mov-to-AR-FPSR**		impliedF
AR[FSR]	**mov-to-AR-FSR**		impliedF
AR[ITC]	**mov-to-AR-ITC**		impliedF

Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
AR[K%], % in 0 - 7	**mov-to-AR-K**[1]		impliedF
AR[LC]	**mod-sched-brs-counted**, **mov-to-AR-LC**		impliedF
AR[PFS]	br.call, brl.call		none
	br.call, brl.call	**mov-to-AR-PFS**	impliedF
AR[RNAT]	alloc, flushrs, loadrs, **mov-to-AR-RNAT**, **mov-to-AR-BSPSTORE**		impliedF
AR[RSC]	**mov-to-AR-RSC**		impliedF
AR[RUC]	**mov-to-AR-RUC**		impliedF
AR[SSD]	**mov-to-AR-SSD**		impliedF
AR[UNAT]{%}, % in 0 - 63	**mov-to-AR-UNAT**, st8.spill		impliedF
AR%, % in 8-15, 20, 22-23, 31, 33-35, 37-39, 41-43, 46-47, 67-111	**none**		none
AR%, % in 48 - 63, 112-127	**mov-to-AR-ig**[1]		impliedF
BR%, % in 0 - 7	br.call[1], brl.call[1]	**mov-to-BR**[1]	impliedF
	mov-to-BR[1]		impliedF
	br.call[1], brl.call[1]		none
CFM	**mod-sched-brs**, br.call, brl.call, br.ret, alloc, clrrrb, cover, rfi		impliedF
CPUID#	**none**		none
CR[CMCV]	**mov-to-CR-CMCV**		impliedF
CR[DCR]	**mov-to-CR-DCR**		impliedF
CR[EOI]	**mov-to-CR-EOI**		SC Section 5.8.3.4, “End of External Interrupt Register (EOI – CR67)” on page 2:124
CR[IFA]	**mov-to-CR-IFA**		impliedF
CR[IFS]	**mov-to-CR-IFS**, cover		impliedF
CR[IHA]	**mov-to-CR-IHA**		impliedF
CR[IIB%], % in 0 - 1	**mov-to-CR-IIB**		impliedF
CR[IIM]	**mov-to-CR-IIM**		impliedF
CR[IIP]	**mov-to-CR-IIP**		impliedF
CR[IIPA]	**mov-to-CR-IIPA**		impliedF
CR[IPSR]	**mov-to-CR-IPSR**		impliedF
CR[IRR%], % in 0 - 3	**mov-from-CR-IVR**		impliedF
CR[ISR]	**mov-to-CR-ISR**		impliedF
CR[ITIR]	**mov-to-CR-ITIR**		impliedF
CR[ITM]	**mov-to-CR-ITM**		impliedF
CR[ITO]	**mov-to-CR-ITO**		impliedF

Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
CR[ITV]	**mov-to-CR-ITV**		impliedF
CR[IVA]	**mov-to-CR-IVA**		impliedF
CR[IVR]	**none**		SC
CR[LID]	**mov-to-CR-LID**		SC
CR[LRR%], % in 0 - 1	**mov-to-CR-LRR**[1]		impliedF
CR[PMV]	**mov-to-CR-PMV**		impliedF
CR[PTA]	**mov-to-CR-PTA**		impliedF
CR[TPR]	**mov-to-CR-TPR**		impliedF
CR%, % in 3, 5-7, 10-15, 18, 28-63, 75-79, 82-127	**none**		none
DBR#	**mov-to-IND-DBR**[3]		impliedF
DTC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d		none
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	itc.i, itc.d, itr.i, itr.d	impliedF
DTC_LIMIT*	ptc.g, ptc.ga		impliedF
DTR	itr.d		impliedF
	itr.d	ptr.d	impliedF
	ptr.d		none
FR%, % in 0 - 1	**none**		none
FR%, % in 2 - 127	**fr-writers**[1], **ldf-c**[1], **ldfp-c**[1]		impliedF
GR0	**none**		none
GR%, % in 1 - 127	**ld-c**[1], **gr-writers**[1]		impliedF
IBR#	**mov-to-IND-IBR**[3]		impliedF
InService*	**mov-to-CR-EOI**, **mov-from-CR-IVR**		SC
IP	**all**		none
ITC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d		none
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	itc.i, itc.d, itr.i, itr.d	impliedF
ITR	itr.i	itr.i, ptr.i	impliedF
	ptr.i		none
memory	**mem-writers**		none
PKR#	**mov-to-IND-PKR**[3]	**mov-to-IND-PKR**[4]	none
	mov-to-IND-PKR[3]		impliedF
PMC#	**mov-to-IND-PMC**[3]		impliedF
PMD#	**mov-to-IND-PMD**[3]		impliedF
PR0	**pr-writers**[1]		none

Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
PR%, % in 1 - 15	**pr-and-writers**[1]		none
	pr-or-writers[1]		none
	pr-unc-writers-fp[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-and-writers**[1], **mov-to-PR-allreg**[7]	**pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-or-writers**[1], **mov-to-PR-allreg**[7]	impliedF
PR%, % in 16 - 62	**pr-and-writers**[1]		none
	pr-or-writers[1]		none
	pr-unc-writers-fp[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-and-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-or-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	impliedF
PR63	**pr-and-writers**[1]		none
	pr-or-writers[1]		none
	mod-sched-brs, **pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-and-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**mod-sched-brs**, **pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-or-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	impliedF
PSR.ac	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.be	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.bn	bsw, rfi		impliedF
PSR.cpl	epc, br.ret, rfi		impliedF
PSR.da	rfi		impliedF
PSR.db	**mov-to-PSR-l**, rfi		impliedF
PSR.dd	rfi		impliedF
PSR.dfh	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.dfl	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.di	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.dt	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ed	rfi		impliedF
PSR.i	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ia	rfi		impliedF
PSR.ic	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.id	rfi		impliedF
PSR.is	br.ia, rfi		impliedF
PSR.it	rfi		impliedF
PSR.lp	**mov-to-PSR-l**, rfi		impliedF
PSR.mc	rfi		impliedF

Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
PSR.mfh	**fr-writers**[9]		none
	user-mask-writers-partial[7], **mov-to-PSR-um**, **fr-writers**[9], **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	impliedF
PSR.mfl	**fr-writers**[9]		none
	user-mask-writers-partial[7], **mov-to-PSR-um**, **fr-writers**[9], **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	impliedF
PSR.pk	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.pp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ri	rfi		impliedF
PSR.rt	**mov-to-PSR-l**, rfi		impliedF
PSR.si	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.sp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ss	rfi		impliedF
PSR.tb	**mov-to-PSR-l**, rfi		impliedF
PSR.up	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.vm	rfi, vmsw		impliedF
RR#	**mov-to-IND-RR**[6]		impliedF
RSE	**rse-writers**[14]		impliedF

5.3.4 WAR Dependency Table

A general rule specific to the WAR table:

1. WAR dependencies are always allowed within instruction groups except for the entry in Table 5-4 below. The readers and subsequent writers specified must be separated by a stop in order to have defined behavior.

Table 5-4. WAR Dependencies Organized by Resource

Resource Name	Readers	Writers	Semantics of Dependency
PR63	**pr-readers-br**[1]	**mod-sched-brs**	stop

5.3.5 Listing of Rules Referenced in Dependency Tables

The following rules restrict the specific instances in which some of the instructions in the tables cause a dependency and must be applied where referenced to correctly interpret those entries. Rules only apply to the instance of the instruction class, or instruction mnemonic prefix where the rule is referenced as a superscript. If the rule is referenced in Table 5-5 where instruction classes are defined, then it applies to all instances of the instruction class.

Rule 1. These instructions only write a register when that register's number is explicitly encoded as a target of the instruction and is only read when it is encoded as a source of the instruction (or encoded as its PR[qp]).

Rule 2. These instructions only read CFM when they access a rotating GR, FR, or PR. **mov-to-PR** and **mov-from-PR** only access CFM when their qualifying predicate is in the rotating region.

Rule 3. These instructions use a general register value to determine the specific indirect register accessed. These instructions only access the register resource specified by the value in bits {7:0} of the dynamic value of the index register.

Rule 4. These instructions only read the given resource when bits {7:0} of the indirect index register value *does not* match the register number of the resource.

Rule 5. All rules are implementation specific.

Rule 6. There is a dependency only when both the index specified by the reader and the index specified by the writer have the same value in bits {63:61}.

Rule 7. These instructions access the specified resource only when the corresponding mask bit is set.

Rule 8. PSR.dfh is only read when these instructions reference FR32-127. PSR.dfl is only read when these instructions reference FR2-31.

Rule 9. PSR.mfl is only written when these instructions write FR2-31. PSR.mfh is only written when these instructions write FR32-127.

Rule 10. The PSR.bn bit is only accessed when one of GR16-31 is specified in the instruction.

Rule 11. The target predicates are written independently of PR[qp], but source registers are only read if PR[qp] is true.

Rule 12. This instruction only reads the specified predicate register when that register is the PR[qp].

Rule 13. This reference to 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-l** or ssm instructions that set PSR.i, PSR.pp or PSR.up.

5.4 Support Tables

Table 5-5. Instruction Classes

Class	Events/Instructions
all	**predicatable-instructions**, **unpredicatable-instructions**
branches	**indirect-brs**, **ip-rel-brs**
cfm-readers	**fr-readers**, **fr-writers**, **gr-readers**, **gr-writers**, **mod-sched-brs**, **predicatable-instructions**, **pr-writers**, alloc, br.call, brl.call, br.ret, cover, loadrs, rfi, **chk-a**, invala.e
chk-a	chk.a.clr, chk.a.nc
cmpxchg	cmpxchg1, cmpxchg2, cmpxchg4, cmpxchg8, cmp8xchg16
czx	czx1, czx2
fcmp-s0	fcmp[Field(sf)==s0]
fcmp-s1	fcmp[Field(sf)==s1]
fcmp-s2	fcmp[Field(sf)==s2]
fcmp-s3	fcmp[Field(sf)==s3]
fetchadd	fetchadd4, fetchadd8
fp-arith	fadd, famax, famin, fcvt.fx, fcvt.fxu, fcvt.xuf, fma, fmax, fmin, fmpy, fms, fnma, fnmpy, fnorm, fpamax, fpamin, fpcvt.fx, fpcvt.fxu, fpma, fpmax, fpmin, fpmpy, fpms, fpnma, fpnmpy, fprcpa, fprsqrta, frcpa, frsqrta, fsub
fp-arith-s0	**fp-arith**[Field(sf)==s0]
fp-arith-s1	**fp-arith**[Field(sf)==s1]
fp-arith-s2	**fp-arith**[Field(sf)==s2]
fp-arith-s3	**fp-arith**[Field(sf)==s3]
fp-non-arith	fabs, fand, fandcm, fclass, fcvt.xf, fmerge, fmix, fneg, fnegabs, for, fpabs, fpmerge, fpack, fpneg, fpnegabs, fselect, fswap, fsxt, fxor, xma, xmpy
fpcmp-s0	fpcmp[Field(sf)==s0]
fpcmp-s1	fpcmp[Field(sf)==s1]
fpcmp-s2	fpcmp[Field(sf)==s2]
fpcmp-s3	fpcmp[Field(sf)==s3]
fr-readers	**fp-arith**, **fp-non-arith**, **mem-writers-fp**, **pr-writers-fp**, chk.s[Format in {M21}], getf
fr-writers	**fp-arith**, **fp-non-arith**\fclass, **mem-readers-fp**, setf
gr-readers	**gr-readers-writers**, **mem-readers**, **mem-writers**, chk.s, cmp, cmp4, fc, itc.i, itc.d, itr.i, itr.d, **mov-to-AR-gr**, **mov-to-BR**, **mov-to-CR**, **mov-to-IND**, **mov-from-IND**, **mov-to-PR-allreg**, **mov-to-PSR-l**, **mov-to-PSR-um**, **probe-all**, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, setf, tbit, tnat
gr-readers-writers	**mov-from-IND**, add, addl, addp4, adds, and, andcm, clz, **czx**, dep\dep[Format in {I13}], extr, **mem-readers-int**, **ld-all-postinc**, **lfetch-postinc**, **mix**, **mux**, or, **pack**, **padd**, **pavg**, **pavgsub**, **pcmp**, **pmax**, **pmin**, **pmpy**, **pmpyshr**, popcnt, **probe-regular**, **psad**, **pshl**, **pshladd**, **pshr**, **pshradd**, **psub**, shl, shladd, shladdp4, shr, shrp, **st-postinc**, sub, **sxt**, tak, thash, tpa, ttag, **unpack**, xor, **zxt**
gr-writers	alloc, dep, getf, **gr-readers-writers**, **mem-readers-int**, **mov-from-AR**, **mov-from-BR**, **mov-from-CR**, **mov-from-PR**, **mov-from-PSR**, **mov-from-PSR-um**, **mov-ip**, movl
indirect-brp	brp[Format in {B7}]
indirect-brs	br.call[Format in {B5}], br.cond[Format in {B4}], br.ia, br.ret
invala-all	invala[Format in {M24}], invala.e
ip-rel-brs	**mod-sched-brs**, br.call[Format in {B3}], brl.call, brl.cond, br.cond[Format in {B1}], br.cloop
ld	ld1, ld2, ld4, ld8, ld8.fill, ld16
ld-a	ld1.a, ld2.a, ld4.a, ld8.a

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
ld-all-postinc	**ld**[Format in {M2 M3}], **ldfp**[Format in {M12}], **ldf**[Format in {M7 M8}]
ld-c	**ld-c-nc**, **ld-c-clr**
ld-c-clr	ld1.c.clr, ld2.c.clr, ld4.c.clr, ld8.c.clr, **ld-c-clr-acq**
ld-c-clr-acq	ld1.c.clr.acq, ld2.c.clr.acq, ld4.c.clr.acq, ld8.c.clr.acq
ld-c-nc	ld1.c.nc, ld2.c.nc, ld4.c.nc, ld8.c.nc
ld-s	ld1.s, ld2.s, ld4.s, ld8.s
ld-sa	ld1.sa, ld2.sa, ld4.sa, ld8.sa
ldf	ldfs, ldfd, ldfe, ldf8, ldf.fill
ldf-a	ldfs.a, ldfd.a, ldfe.a, ldf8.a
ldf-c	**ldf-c-nc**, **ldf-c-clr**
ldf-c-clr	ldfs.c.clr, ldfd.c.clr, ldfe.c.clr, ldf8.c.clr
ldf-c-nc	ldfs.c.nc, ldfd.c.nc, ldfe.c.nc, ldf8.c.nc
ldf-s	ldfs.s, ldfd.s, ldfe.s, ldf8.s
ldf-sa	ldfs.sa, ldfd.sa, ldfe.sa, ldf8.sa
ldfp	ldfps, ldfpd, ldfp8
ldfp-a	ldfps.a, ldfpd.a, ldfp8.a
ldfp-c	**ldfp-c-nc**, **ldfp-c-clr**
ldfp-c-clr	ldfps.c.clr, ldfpd.c.clr, ldfp8.c.clr
ldfp-c-nc	ldfps.c.nc, ldfpd.c.nc, ldfp8.c.nc
ldfp-s	ldfps.s, ldfpd.s, ldfp8.s
ldfp-sa	ldfps.sa, ldfpd.sa, ldfp8.sa
lfetch-all	lfetch
lfetch-fault	lfetch[Field(lftype)==fault]
lfetch-nofault	lfetch[Field(lftype)==]
lfetch-postinc	lfetch[Format in {M20 M22}]
mem-readers	**mem-readers-fp**, **mem-readers-int**
mem-readers-alat	**ld-a**, **ldf-a**, **ldfp-a**, **ld-sa**, **ldf-sa**, **ldfp-sa**, **ld-c**, **ldf-c**, **ldfp-c**
mem-readers-fp	**ldf**, **ldfp**
mem-readers-int	**cmpxchg**, **fetchadd**, **xchg**, **ld**
mem-readers-spec	**ld-s**, **ld-sa**, **ldf-s**, **ldf-sa**, **ldfp-s**, **ldfp-sa**
mem-writers	**mem-writers-fp**, **mem-writers-int**
mem-writers-fp	**stf**
mem-writers-int	**cmpxchg**, **fetchadd**, **xchg**, **st**
mix	mix1, mix2, mix4
mod-sched-brs	br.cexit, br.ctop, br.wexit, br.wtop
mod-sched-brs-counted	br.cexit, br.cloop, br.ctop
mov-from-AR	**mov-from-AR-M**, **mov-from-AR-I**, **mov-from-AR-IM**
mov-from-AR-BSP	**mov-from-AR-M**[Field(ar3) == BSP]
mov-from-AR-BSPSTORE	**mov-from-AR-M**[Field(ar3) == BSPSTORE]
mov-from-AR-CCV	**mov-from-AR-M**[Field(ar3) == CCV]
mov-from-AR-CFLG	**mov-from-AR-M**[Field(ar3) == CFLG]
mov-from-AR-CSD	**mov-from-AR-M**[Field(ar3) == CSD]
mov-from-AR-EC	**mov-from-AR-I**[Field(ar3) == EC]
mov-from-AR-EFLAG	**mov-from-AR-M**[Field(ar3) == EFLAG]
mov-from-AR-FCR	**mov-from-AR-M**[Field(ar3) == FCR]

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
mov-from-AR-FDR	**mov-from-AR-M**[Field(ar3) == FDR]
mov-from-AR-FIR	**mov-from-AR-M**[Field(ar3) == FIR]
mov-from-AR-FPSR	**mov-from-AR-M**[Field(ar3) == FPSR]
mov-from-AR-FSR	**mov-from-AR-M**[Field(ar3) == FSR]
mov-from-AR-I	mov_ar[Format in {I28}]
mov-from-AR-ig	**mov-from-AR-IM**[Field(ar3) in {48-63 112-127}]
mov-from-AR-IM	mov_ar[Format in {I28 M31}]
mov-from-AR-ITC	**mov-from-AR-M**[Field(ar3) == ITC]
mov-from-AR-K	**mov-from-AR-M**[Field(ar3) in {K0 K1 K2 K3 K4 K5 K6 K7}]
mov-from-AR-LC	**mov-from-AR-I**[Field(ar3) == LC]
mov-from-AR-M	mov_ar[Format in {M31}]
mov-from-AR-PFS	**mov-from-AR-I**[Field(ar3) == PFS]
mov-from-AR-RNAT	**mov-from-AR-M**[Field(ar3) == RNAT]
mov-from-AR-RSC	**mov-from-AR-M**[Field(ar3) == RSC]
mov-from-AR-RUC	**mov-from-AR-M**[Field(ar3) == RUC]
mov-from-AR-rv	**none**
mov-from-AR-SSD	**mov-from-AR-M**[Field(ar3) == SSD]
mov-from-AR-UNAT	**mov-from-AR-M**[Field(ar3) == UNAT]
mov-from-BR	mov_br[Format in {I22}]
mov-from-CR	mov_cr[Format in {M33}]
mov-from-CR-CMCV	**mov-from-CR**[Field(cr3) == CMCV]
mov-from-CR-DCR	**mov-from-CR**[Field(cr3) == DCR]
mov-from-CR-EOI	**mov-from-CR**[Field(cr3) == EOI]
mov-from-CR-IFA	**mov-from-CR**[Field(cr3) == IFA]
mov-from-CR-IFS	**mov-from-CR**[Field(cr3) == IFS]
mov-from-CR-IHA	**mov-from-CR**[Field(cr3) == IHA]
mov-from-CR-IIB	**mov-from-CR**[Field(cr3) in {IIB0 IIB1}]
mov-from-CR-IIM	**mov-from-CR**[Field(cr3) == IIM]
mov-from-CR-IIP	**mov-from-CR**[Field(cr3) == IIP]
mov-from-CR-IIPA	**mov-from-CR**[Field(cr3) == IIPA]
mov-from-CR-IPSR	**mov-from-CR**[Field(cr3) == IPSR]
mov-from-CR-IRR	**mov-from-CR**[Field(cr3) in {IRR0 IRR1 IRR2 IRR3}]
mov-from-CR-ISR	**mov-from-CR**[Field(cr3) == ISR]
mov-from-CR-ITIR	**mov-from-CR**[Field(cr3) == ITIR]
mov-from-CR-ITM	**mov-from-CR**[Field(cr3) == ITM]
mov-from-CR-ITO	**mov-from-CR**[Field(cr3) == ITO]
mov-from-CR-ITV	**mov-from-CR**[Field(cr3) == ITV]
mov-from-CR-IVA	**mov-from-CR**[Field(cr3) == IVA]
mov-from-CR-IVR	**mov-from-CR**[Field(cr3) == IVR]
mov-from-CR-LID	**mov-from-CR**[Field(cr3) == LID]
mov-from-CR-LRR	**mov-from-CR**[Field(cr3) in {LRR0 LRR1}]
mov-from-CR-PMV	**mov-from-CR**[Field(cr3) == PMV]
mov-from-CR-PTA	**mov-from-CR**[Field(cr3) == PTA]
mov-from-CR-rv	**none**
mov-from-CR-TPR	**mov-from-CR**[Field(cr3) == TPR]

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
mov-from-IND	mov_indirect[Format in {M43}]
mov-from-IND-CPUID	**mov-from-IND**[Field(ireg) == cpuid]
mov-from-IND-DBR	**mov-from-IND**[Field(ireg) == dbr]
mov-from-IND-IBR	**mov-from-IND**[Field(ireg) == ibr]
mov-from-IND-PKR	**mov-from-IND**[Field(ireg) == pkr]
mov-from-IND-PMC	**mov-from-IND**[Field(ireg) == pmc]
mov-from-IND-PMD	**mov-from-IND**[Field(ireg) == pmd]
mov-from-IND-priv	**mov-from-IND**[Field(ireg) in {dbr ibr pkr pmc rr}]
mov-from-IND-RR	**mov-from-IND**[Field(ireg) == rr]
mov-from-interruption-CR	**mov-from-CR-ITIR**, **mov-from-CR-IFS**, **mov-from-CR-IIB**, **mov-from-CR-IIM**, **mov-from-CR-IIP**, **mov-from-CR-IPSR**, **mov-from-CR-ISR**, **mov-from-CR-IFA**, **mov-from-CR-IHA**, **mov-from-CR-IIPA**
mov-from-PR	mov_pr[Format in {I25}]
mov-from-PSR	mov_psr[Format in {M36}]
mov-from-PSR-um	mov_um[Format in {M36}]
mov-ip	mov_ip[Format in {I25}]
mov-to-AR	**mov-to-AR-M**, **mov-to-AR-I**
mov-to-AR-BSP	**mov-to-AR-M**[Field(ar3) == BSP]
mov-to-AR-BSPSTORE	**mov-to-AR-M**[Field(ar3) == BSPSTORE]
mov-to-AR-CCV	**mov-to-AR-M**[Field(ar3) == CCV]
mov-to-AR-CFLG	**mov-to-AR-M**[Field(ar3) == CFLG]
mov-to-AR-CSD	**mov-to-AR-M**[Field(ar3) == CSD]
mov-to-AR-EC	**mov-to-AR-I**[Field(ar3) == EC]
mov-to-AR-EFLAG	**mov-to-AR-M**[Field(ar3) == EFLAG]
mov-to-AR-FCR	**mov-to-AR-M**[Field(ar3) == FCR]
mov-to-AR-FDR	**mov-to-AR-M**[Field(ar3) == FDR]
mov-to-AR-FIR	**mov-to-AR-M**[Field(ar3) == FIR]
mov-to-AR-FPSR	**mov-to-AR-M**[Field(ar3) == FPSR]
mov-to-AR-FSR	**mov-to-AR-M**[Field(ar3) == FSR]
mov-to-AR-gr	**mov-to-AR-M**[Format in {M29}], **mov-to-AR-I**[Format in {I26}]
mov-to-AR-I	mov_ar[Format in {I26 I27}]
mov-to-AR-ig	**mov-to-AR-IM**[Field(ar3) in {48-63 112-127}]
mov-to-AR-IM	mov_ar[Format in {I26 I27 M29 M30}]
mov-to-AR-ITC	**mov-to-AR-M**[Field(ar3) == ITC]
mov-to-AR-K	**mov-to-AR-M**[Field(ar3) in {K0 K1 K2 K3 K4 K5 K6 K7}]
mov-to-AR-LC	**mov-to-AR-I**[Field(ar3) == LC]
mov-to-AR-M	mov_ar[Format in {M29 M30}]
mov-to-AR-PFS	**mov-to-AR-I**[Field(ar3) == PFS]
mov-to-AR-RNAT	**mov-to-AR-M**[Field(ar3) == RNAT]
mov-to-AR-RSC	**mov-to-AR-M**[Field(ar3) == RSC]
mov-to-AR-RUC	**mov-to-AR-M**[Field(ar3) == RUC]
mov-to-AR-SSD	**mov-to-AR-M**[Field(ar3) == SSD]
mov-to-AR-UNAT	**mov-to-AR-M**[Field(ar3) == UNAT]
mov-to-BR	mov_br[Format in {I21}]
mov-to-CR	mov_cr[Format in {M32}]
mov-to-CR-CMCV	**mov-to-CR**[Field(cr3) == CMCV]

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
mov-to-CR-DCR	**mov-to-CR**[Field(cr3) == DCR]
mov-to-CR-EOI	**mov-to-CR**[Field(cr3) == EOI]
mov-to-CR-IFA	**mov-to-CR**[Field(cr3) == IFA]
mov-to-CR-IFS	**mov-to-CR**[Field(cr3) == IFS]
mov-to-CR-IHA	**mov-to-CR**[Field(cr3) == IHA]
mov-to-CR-IIB	**mov-to-CR**[Field(cr3) in {IIB0 IIB1}]
mov-to-CR-IIM	**mov-to-CR**[Field(cr3) == IIM]
mov-to-CR-IIP	**mov-to-CR**[Field(cr3) == IIP]
mov-to-CR-IIPA	**mov-to-CR**[Field(cr3) == IIPA]
mov-to-CR-IPSR	**mov-to-CR**[Field(cr3) == IPSR]
mov-to-CR-IRR	**mov-to-CR**[Field(cr3) in {IRR0 IRR1 IRR2 IRR3}]
mov-to-CR-ISR	**mov-to-CR**[Field(cr3) == ISR]
mov-to-CR-ITIR	**mov-to-CR**[Field(cr3) == ITIR]
mov-to-CR-ITM	**mov-to-CR**[Field(cr3) == ITM]
mov-to-CR-ITO	**mov-to-CR**[Field(cr3) == ITO]
mov-to-CR-ITV	**mov-to-CR**[Field(cr3) == ITV]
mov-to-CR-IVA	**mov-to-CR**[Field(cr3) == IVA]
mov-to-CR-IVR	**mov-to-CR**[Field(cr3) == IVR]
mov-to-CR-LID	**mov-to-CR**[Field(cr3) == LID]
mov-to-CR-LRR	**mov-to-CR**[Field(cr3) in {LRR0 LRR1}]
mov-to-CR-PMV	**mov-to-CR**[Field(cr3) == PMV]
mov-to-CR-PTA	**mov-to-CR**[Field(cr3) == PTA]
mov-to-CR-TPR	**mov-to-CR**[Field(cr3) == TPR]
mov-to-IND	mov_indirect[Format in {M42}]
mov-to-IND-CPUID	**mov-to-IND**[Field(ireg) == cpuid]
mov-to-IND-DBR	**mov-to-IND**[Field(ireg) == dbr]
mov-to-IND-IBR	**mov-to-IND**[Field(ireg) == ibr]
mov-to-IND-PKR	**mov-to-IND**[Field(ireg) == pkr]
mov-to-IND-PMC	**mov-to-IND**[Field(ireg) == pmc]
mov-to-IND-PMD	**mov-to-IND**[Field(ireg) == pmd]
mov-to-IND-priv	**mov-to-IND**
mov-to-IND-RR	**mov-to-IND**[Field(ireg) == rr]
mov-to-interruption-CR	**mov-to-CR-ITIR**, **mov-to-CR-IFS**, **mov-to-CR-IIB**, **mov-to-CR-IIM**, **mov-to-CR-IIP**, **mov-to-CR-IPSR**, **mov-to-CR-ISR**, **mov-to-CR-IFA**, **mov-to-CR-IHA**, **mov-to-CR-IIPA**
mov-to-PR	**mov-to-PR-allreg**, **mov-to-PR-rotreg**
mov-to-PR-allreg	mov_pr[Format in {I23}]
mov-to-PR-rotreg	mov_pr[Format in {I24}]
mov-to-PSR-l	mov_psr[Format in {M35}]
mov-to-PSR-um	mov_um[Format in {M35}]
mux	mux1, mux2
non-access	fc, lfetch, **probe-all**, tpa, tak
none	-
pack	pack2, pack4
padd	padd1, padd2, padd4
pavg	pavg1, pavg2

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
pavgsub	pavgsub1, pavgsub2
pcmp	pcmp1, pcmp2, pcmp4
pmax	pmax1, pmax2
pmin	pmin1, pmin2
pmpy	pmpy2
pmpyshr	pmpyshr2
pr-and-writers	**pr-gen-writers-int**[Field(ctype) in {and andcm}], **pr-gen-writers-int**[Field(ctype) in {or.andcm and.orcm}]
pr-gen-writers-fp	fclass, fcmp
pr-gen-writers-int	cmp, cmp4, tbit, tf, tnat
pr-norm-writers-fp	**pr-gen-writers-fp**[Field(ctype)==]
pr-norm-writers-int	**pr-gen-writers-int**[Field(ctype)==]
pr-or-writers	**pr-gen-writers-int**[Field(ctype) in {or orcm}], **pr-gen-writers-int**[Field(ctype) in {or.andcm and.orcm}]
pr-readers-br	br.call, br.cond, brl.call, brl.cond, br.ret, br.wexit, br.wtop, break.b, hint.b, nop.b, **ReservedBQP**
pr-readers-nobr-nomovpr	add, addl, addp4, adds, and, andcm, break.f, break.i, break.m, break.x, chk.s, **chk-a**, cmp, cmp4, **cmpxchg**, clz, **czx**, dep, extr, **fp-arith**, **fp-non-arith**, fc, fchkf, fclrf, fcmp, **fetchadd**, fpcmp, fsetc, fwb, getf, hint.f, hint.i, hint.m, hint.x, **invala-all**, itc.i, itc.d, itr.i, itr.d, **ld**, **ldf**, **ldfp**, **lfetch-all**, mf, **mix**, **mov-from-AR-M**, **mov-from-AR-IM**, **mov-from-AR-I**, **mov-to-AR-M**, **mov-to-AR-I**, **mov-to-AR-IM**, **mov-to-BR**, **mov-from-BR**, **mov-to-CR**, **mov-from-CR**, **mov-to-IND**, **mov-from-IND**, **mov-ip**, **mov-to-PSR-l**, **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**, **pshradd**, **psub**, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.d, ptr.i, **ReservedQP**, rsm, setf, shl, shladd, shladdp4, shr, shrp, srlz.i, srlz.d, ssm, **st**, **stf**, sub, sum, **sxt**, sync, tak, tbit, tf, thash, tnat, tpa, ttag, **unpack**, **xchg**, xma, xmpy, xor, **zxt**
pr-unc-writers-fp	**pr-gen-writers-fp**[Field(ctype)==unc][11], fprcpa[11], fprsqrta[11], frcpa[11], frsqrta[11]
pr-unc-writers-int	**pr-gen-writers-int**[Field(ctype)==unc][11]
pr-writers	**pr-writers-int**, **pr-writers-fp**
pr-writers-fp	**pr-norm-writers-fp**, **pr-unc-writers-fp**
pr-writers-int	**pr-norm-writers-int**, **pr-unc-writers-int**, **pr-and-writers**, **pr-or-writers**
predicatable-instructions	**mov-from-PR**, **mov-to-PR**, **pr-readers-br**, **pr-readers-nobr-nomovpr**
priv-ops	**mov-to-IND-priv**, bsw, itc.i, itc.d, itr.i, itr.d, **mov-to-CR**, **mov-from-CR**, **mov-to-PSR-l**, **mov-from-PSR**, **mov-from-IND-priv**, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, rfi, rsm, ssm, tak, tpa, vmsw
probe-all	**probe-fault**, **probe-regular**
probe-fault	probe[Format in {M40}]
probe-regular	probe[Format in {M38 M39}]
psad	psad1
pshl	pshl2, pshl4
pshladd	pshladd2
pshr	pshr2, pshr4
pshradd	pshradd2
psub	psub1, psub2, psub4
ReservedBQP	-[15]
ReservedQP	-[16]

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
rse-readers	alloc, br.call, br.ia, br.ret, brl.call, cover, flushrs, loadrs, **mov-from-AR-BSP**, **mov-from-AR-BSPSTORE**, **mov-to-AR-BSPSTORE**, **mov-from-AR-RNAT**, **mov-to-AR-RNAT**, rfi
rse-writers	alloc, br.call, br.ia, br.ret, brl.call, cover, flushrs, loadrs, **mov-to-AR-BSPSTORE**, rfi
st	st1, st2, st4, st8, st8.spill, st16
st-postinc	**stf**[Format in {M10}], **st**[Format in {M5}]
stf	stfs, stfd, stfe, stf8, stf.spill
sxt	sxt1, sxt2, sxt4
sys-mask-writers-partial	rsm, ssm
unpack	unpack1, unpack2, unpack4
unpredicatable-instructions	alloc, br.cloop, br.ctop, br.cexit, br.ia, brp, bsw, clrrrb, cover, epc, flushrs, loadrs, rfi, vmsw
user-mask-writers-partial	rum, sum
xchg	xchg1, xchg2, xchg4, xchg8
zxt	zxt1, zxt2, zxt4

§

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

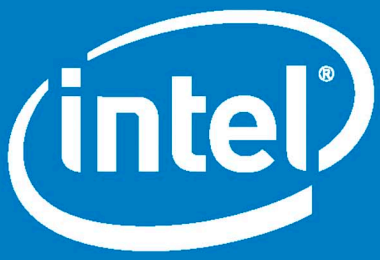
U

V

W

X

Z



0510/FL/DS/NOD/RRD/2K 323207-001US