IA-64 Application Developer’s Architecture Guide

May 1999

Order Number: 245188-001

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

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

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

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

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

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

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

Contents

Part I: IA-64 Application Architecture Guide

Part II: IA-64 Optimization Guide

Part III: Appendices

Figures

Tables

About the IA-64 Application Developer's Architecture Guide 1

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

The first part of this document (*Part I, "IA-64 Application Architecture Guide"*) provides a comprehensive description of the IA-64 architecture which is exposed to application software. This includes information on application level resources (registers, etc.), the application environment, detailed application (non-privileged) instruction descriptions, formats and encodings. The IA-64 architecture supports IA-32 instruction set compatibility which is covered in this document.

The second portion of this document (*Part II, "IA-64 Optimization Guide"*) provides a refresher on the IA-64 application architecture before describing certain IA-64 architectural features and elaborates on applying these features to generate highly optimized code. Each section describes how specific IA-64 features can be used to reduce or eliminate performance barriers.

Full details of the IA-64 programming environment including the system architecture and software conventions will be provided in IA-64 Programmer's Reference Manual to be available later.

1.1 Overview of the IA-64 Application Developer's Architecture Guide

Chapter 1, "About the IA-64 Application Developer's Architecture Guide". Gives an overview of this guide.

Chapter 2, "Introduction to the IA-64 Processor Architecture". Provides an overview of key features of IA-64 architecture.

Chapter 3, "IA-64 Execution Environment". Describes the IA-64 application architectural state (registers, memory, etc.).

Chapter 4, "IA-64 Application Programming Model". Describes the IA-64 architecture from the perspective of the application programmer. IA-64 instructions are grouped into related functions and an overview of their behavior is given.

Chapter 5, "IA-64 Floating-point Programming Model". This chapter provides a description of IA-64 floating-point registers, data types and formats and floating-point instructions.

Chapter 6, "IA-32 Application Execution Model in an IA-64 System Environment". This chapter describes execution of IA-32 applications running in IA-64 System Environment.

Chapter 7, "IA-64 Instruction Reference". Provides detailed description of IA-64 application instructions, operation, and instruction format.

Chapter 8, “About the IA-64 Optimization Guide”. Gives an overview of the IA-64 optimization guide.

Chapter 9, “Introduction to IA-64 Programming”. Provides an overview of the IA-64 application programming environment.

Chapter 10, “Memory Reference”. Discusses features and optimizations related to control and data speculation.

Chapter 11, “Predication, Control Flow, and Instruction Stream”. Describes optimization features related to predication, control flow, and branch hints.

Chapter 12, “Software Pipelining and Loop Support”. Provides a detailed discussion on optimizing loops through use of software pipelining.

Chapter 13, “Floating-point Applications”. Discusses current performance limitations in floating-point applications and IA-64 features that address these limitations.

Appendix A, “Instruction Sequencing Considerations”. Describes instruction sequencing in IA-64 architecture.

Appendix B, “IA-64 Pseudo-Code Functions”. Describes pseudo-code functions used in Chapter 7, “IA-64 Instruction Reference”.

Appendix C, “IA-64 Instruction Formats”. Describes the encoding and instruction format of instructions covered in Chapter 7.

1.2 Terminology

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

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

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

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

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

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

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

1.3 Related Documents

- ***Intel Architecture Software Developer's Manual* –** This reference set provides detailed information on Intel 32-bit architecture and is readily available from the Intel Literature Center.

Part I: IA-64 Application Architecture Guide

Introduction to the IA-64 Processor Architecture 2

The IA-64 architecture was designed to overcome the performance limitations of traditional architectures and provide maximum headroom for the future. To achieve this, IA-64 was designed with an array of innovative features to extract greater instruction level parallelism including: speculation, predication, large register files, a register stack, advanced branch architecture, and many others. 64-bit memory addressability was added to meet the increasing large memory footprint requirements of data warehousing, e-business, and other high performance server applications. The IA-64 architecture has an innovative floating-point architecture and other enhancements that support the high performance requirements of workstation applications such as digital content creation, design engineering, and scientific analysis.

The IA-64 architecture also provides binary compatibility with the IA-32 instruction set. IA-64 processors can run IA-32 applications on an IA-64 operating system that supports execution of IA-32 applications. IA-64 processors can run IA-32 application binaries on IA-32 legacy operating systems assuming the platform and firmware support exists in the system. The IA-64 architecture also provides the capability to support mixed IA-32 and IA-64 code execution.

2.1 IA-64 Operating Environments

The IA-64 architecture supports two operating system environments:

- IA-32 System Environment: supports IA-32 32-bit operating systems, and
- IA-64 System Environment: supports IA-64 operating systems.

The architectural model also supports a mixture of IA-32 and IA-64 applications within a single IA-64 operating system. Table 2-1 defines the major operating environments supported on IA-64 processors.

Table 2-1. IA-64 Processor Operating Environments

System Environment	Application Environment	Usage
IA-32	IA-32 Instruction Set	IA-32 Protected Mode, Real Mode and Virtual 8086 Mode application and operating system environment. Compatible with IA-32 Pentium®, Pentium Pro, Pentium II, and Pentium III processors.
IA-64	IA-32 Protected Mode	IA-32 Protected Mode applications in the IA-64 system environment, if supported by OS.
	IA-32 Real Mode	IA-32 Real Mode applications in the IA-64 system environment, if supported by OS.
	IA-32 Virtual Mode	IA-32 Virtual 86 Mode applications in the IA-64 system environment, if supported by OS.
	IA-64 Instruction Set	IA-64 Applications on IA-64 operating systems.

2.2 Instruction Set Transition Model Overview

Within the IA-64 System Environment, the processor can execute either IA-32 or IA-64 instructions at any time. Three special instructions and interruptions are defined to transition the processor between the IA-32 and the IA-64 instruction set.

- `jmpe` (IA-32 instruction) Jump to an IA-64 target instruction, and change the instruction set to IA-64.
- `br.ia` (IA-64 instruction) IA-64 branch to an IA-32 target instruction, and change the instruction set to IA-32.
- Interruptions transition the processor to the IA-64 instruction set for handling all interruption conditions.
- `rfi` (IA-64 instruction) "return from interruption", is defined to return to an IA-32 or IA-64 instruction.

The `jmpe` and `br.ia` instructions provide a low overhead mechanism to transfer control between the instruction sets. These instructions are typically incorporated into "thunks" or "stubs" that implement the required call linkage and calling conventions to call dynamic or statically linked libraries. Please refer to Chapter 6, "IA-32 Application Execution Model in an IA-64 System Environment" for additional details.

2.3 IA-64 Instruction Set Features

IA-64 incorporates architecture features which enable high sustained performance and remove barriers to further performance increases. The IA-64 architecture is based on the following principles:

- Explicit parallelism
  - — Mechanisms for synergy between the compiler and the processor
  - — Massive resources to take advantage of instruction level parallelism
  - — 128 Integer and Floating-point registers, 64 1-bit predicate registers, 8 branch registers
  - — Support for many execution units and memory ports
- Features that enhance instruction level parallelism
  - — Speculation (which minimizes memory latency impact).
  - — Predication (which removes branches).
  - — Software pipelining of loops with low overhead
  - — Branch prediction to minimize the cost of branches
- Focussed enhancements for improved software performance
  - — Special support for software modularity
  - — High performance floating-point architecture
  - — Specific multimedia instructions

The following sections highlight these important features of IA-64.

2.4 Instruction Level Parallelism

Instruction Level Parallelism (ILP) is the ability to execute multiple instructions at the same time. The IA-64 architecture allows issuing of independent instructions in bundles (three instructions per bundle) for parallel execution and can issue multiple bundles per clock. Supported by a large number of parallel resources such as large register files and multiple execution units, the IA-64 architecture enables the compiler to manage work in progress and schedule simultaneous threads of computation.

The IA-64 architecture incorporates mechanisms to take advantage of ILP. Compilers for traditional architectures are often limited in their ability to utilize speculative information because it cannot always be guaranteed to be correct. The IA-64 architecture enables the compiler to exploit speculative information without sacrificing the correct execution of an application (see Section 2.6). In traditional architectures, procedure calls limit performance since registers need be spilled and filled. IA-64 enables procedures to communicate register usage to the processor. This allows the processor to schedule procedure register operations even when there is a low degree of ILP. See Section 2.7, “Register Stack” on page 2-5.

2.5 Compiler to Processor Communication

The IA-64 architecture provides mechanisms, such as instruction templates, branch hints, and cache hints to enable the compiler to communicate compile-time information to the processor. In addition, IA-64 allows compiled code to manage the processor hardware using run-time information. These communication mechanisms are vital in minimizing the performance penalties associated with branches and cache misses.

Every memory load and store in IA-64 has a 2-bit cache hint field in which the compiler encodes its prediction of the spatial and/or temporal locality of the memory area being accessed. An IA-64 processor can use this information to determine the placement of cache lines in the cache hierarchy. This leads to better utilization of the hierarchy since the relative cost of cache misses continues to grow.

2.6 Speculation

There are two types of speculation: control and data. In both control and data speculation, the compiler exposes ILP by issuing an operation early and removing the latency of this operation from critical path. The compiler will issue an operation speculatively if it is reasonably sure that the speculation will be beneficial. To be beneficial two conditions should hold: it must be statistically frequent enough that the probability it will require recovery is small, and issuing the operation early should expose further ILP-enhancing optimization. Speculation is one of the primary mechanisms for the compiler to exploit statistical ILP by overlapping, and therefore tolerating, the latencies of operations.

2.6.1 Control Speculation

Control speculation is the execution of an operation before the branch which guards it. Consider the code sequence below:

```
if (a>b) load(ld_addr1,target1)
else load(ld_addr2, target2)
```

If the operation `load(ld_addr1,target1)`were to be performed prior to the determination of `(a>b)`, then the operation would be control speculative with respect to the controlling condition `(a>b)`. Under normal execution, the operation `load(ld_addr1,target1)` may or may not execute. If the new control speculative load causes an exception then the exception should only be serviced if `(a>b)` is true. When the compiler uses control speculation it leaves a check operation at the original location. The check verifies whether an exception has occurred and if so it branches to recovery code. The code sequence above now translates into:

```
/* off critical path */
sload(ld_addr1,target1)
sload(ld_addr2,target2)

/* other operations including uses of target1/target2 */
if (a>b) scheck(target1,recovery_addr1)
else scheck(target2, recovery_addr2)
```

2.6.2 Data Speculation

Data speculation is the execution of a memory load prior to a store that preceded it and that may potentially alias with it. Data speculative loads are also referred to as “advanced loads”. Consider the code sequence below:

```
store(st_addr,data)
load(ld_addr,target)
use(target)
```

The process of determining at compile time the relationship between memory addresses is called disambiguation. In the example above, if `ld_addr` and `st_addr` cannot be disambiguated, and if the load were to be performed prior to the store, then the load would be data speculative with respect to the store. If memory addresses overlap during execution, a data-speculative load issued before the store might return a different value than a regular load issued after the store. Therefore analogous to control speculation, when the compiler data speculates a load, it leaves a check instruction at the original location of the load. The check verifies whether an overlap has occurred and if so it branches to recovery code. The code sequence above now translates into:

```
/* off critical path */
aload(ld_addr,target)

/* other operations including uses of target */
store(st_addr,data)
acheck(target,recovery_addr)
use(target)
```

2.6.3 Predication

Predication is the conditional execution of instructions. Conditional execution is implemented through branches in traditional architectures. IA-64 implements this function through the use of predicated instructions. Predication removes branches used for conditional execution resulting in larger basic blocks and the elimination of associated mispredict penalties.

To illustrate, an unpredicated instruction

```
r1 = r2 + r3
```

when predicated, would be of the form

```
if (p5) r1 = r2 + r3
```

In this example p5 is the controlling predicate that decides whether or not the instruction executes and updates state. If the predicate value is true, then the instruction updates state. Otherwise it generally behaves like a nop. Predicates are assigned values by compare instructions.

Predicated execution avoids branches, and simplifies compiler optimizations by converting a control dependence to a data dependence. Consider the original code:

```
if (a>b) c = c + 1
else d = d * e + f
```

The branch at (a>b) can be avoided by converting the code above to the predicated code:

```
pT, pF = compare(a>b)
if (pT) c = c + 1
if (pF) d = d * e + f
```

The predicate pT is set to 1 if the condition evaluates to true, and to 0 if the condition evaluates to false. The predicate pF is the complement of pT. The control dependence of the instructions c = c + 1 and d = d * e + f on the branch with the condition (a>b) is now converted into a data dependence on compare(a>b) through predicates pT and pF (the branch is eliminated). An added benefit is that the compiler can schedule the instructions under pT and pF to execute in parallel. It is also worth noting that there are several different types of compare instructions that write predicates in different manners including unconditional compares and parallel compares.

2.7 Register Stack

IA-64 avoids the unnecessary spilling and filling of registers at procedure call and return interfaces through compiler-controlled renaming. At a call site, a new frame of registers is available to the called procedure without the need for register spill and fill (either by the caller or by the callee). Register access occurs by renaming the virtual register identifiers in the instructions through a base register into the physical registers. The callee can freely use available registers without having to spill and eventually restore the caller’s registers. The callee executes an alloc instruction specifying the number of registers it expects to use in order to ensure that enough registers are available. If sufficient registers are not available (stack overflow), the alloc stalls the processor and spills the caller’s registers until the requested number of registers are available.

At the return site, the base register is restored to the value that the caller was using to access registers prior to the call. Some of the caller’s registers may have been spilled by the hardware and not yet restored. In this case (stack underflow), the return stalls the processor until the processor has restored an appropriate number of the caller’s registers. The hardware can exploit the explicit register stack frame information to spill and fill registers from the register stack to memory at the best opportunity (independent of the calling and called procedures).

2.8 Branching

In addition to removing branches through the use of predication, several mechanisms are provided to decrease the branch misprediction rate and the cost of the remaining mispredicted branches. These mechanisms provide ways for the compiler to communicate information about branch conditions to the processor.

For indirect branches, a branch register is used to hold the target address.

Special loop-closing branches are provided to accelerate counted loops and modulo-scheduled loops. These branches provide information that allows for perfect prediction of loop termination, thereby eliminating costly mispredict penalties and a reduction of the loop overhead.

2.9 Register Rotation

Modulo scheduling of a loop is analogous to hardware pipelining of a functional unit since the next iteration of the loop starts before the previous iteration has finished. The iteration is split into stages similar to the stages of an execution pipeline. Modulo scheduling allows the compiler to execute loop iterations in parallel rather than sequentially. The concurrent execution of multiple iterations traditionally requires unrolling of the loop and software renaming of registers. IA-64 allows the renaming of registers which provide every iteration with its own set of registers, avoiding the need for unrolling. This kind of register renaming is called register rotation. The result is that software pipelining can be applied to a much wider variety of loops - both small as well as large with significantly reduced overhead.

2.10 Floating-point Architecture

IA-64 defines a floating-point architecture with full IEEE support for the single, double, and double-extended (80-bit) data types. Some extensions, such as a fused multiply and add operation, minimum and maximum functions, and a register file format with a larger range than the double-extended memory format, are also included. 128 floating-point registers are defined. Of these, 96 registers are rotating (not stacked) and can be used to modulo schedule loops compactly. Multiple floating-point status registers are provided for speculation.

IA-64 has parallel FP instructions which operate on two 32-bit single precision numbers, resident in a single floating-point register, in parallel and independently. These instructions significantly increase the single precision floating-point computation throughput and enhance the performance of 3D intensive applications and games.

2.11 Multimedia Support

IA-64 has multimedia instructions which treat the general registers as concatenations of eight 8-bit, four 16-bit, or two 32-bit elements. These instructions operate on each element in parallel, independent of the others. IA-64 multimedia instructions are semantically compatible with Intel’s MMX™ technology instructions and Streaming SIMD Extensions instruction technology.

IA-64 Execution Environment 3

The architectural state consists of registers and memory. The results of instruction execution become architecturally visible according to a set of execution sequencing rules. This chapter describes the IA-64 application architectural state and the rules for execution sequencing.

3.1 Application Register State

The following is a list of the registers available to application programs (see Figure 3-1):

- **General Registers (GRs)** – General purpose 64-bit register file, GR0 – GR127. IA-32 integer and segment registers are contained in GR8 - GR31 when executing IA-32 instructions.
- **Floating-point Registers (FRs)** – Floating-point register file, FR0 – FR127. IA-32 floating-point and multi-media registers are contained in FR8 - FR31 when executing IA-32 instructions.
- **Predicate Registers (PRs)** – Single-bit registers, used in IA-64 predication and branching, PR0 – PR63.
- **Branch Registers (BRs)** – Registers used in IA-64 branching, BR0 – BR7.
- **Instruction Pointer (IP)** – Register which holds the bundle address of the currently executing IA-64 instruction, or byte address of the currently executing IA-32 instruction.
- **Current Frame Marker (CFM)** – State that describes the current general register stack frame, and FR/PR rotation.
- **Application Registers (ARs)** – A collection of special-purpose IA-64 and IA-32 application registers.
- **Performance Monitor Data Registers (PMD)** – Data registers for performance monitor hardware.
- **User Mask (UM)** – A set of single-bit values used for alignment traps, performance monitors, and to monitor floating-point register usage.
- **Processor Identifiers (CPUID)** – Registers that describe processor implementation-dependent IA-64 features.

IA-32 application register state is entirely contained within the larger IA-64 application register set and is accessible by IA-64 instructions. IA-32 instructions cannot access the IA-64 specific register set.

3.1.1 Reserved and Ignored Registers

Registers which are not defined are either reserved or ignored. An access to a **reserved register** raises an Illegal Operation fault. A read of an **ignored register** returns zero. Software may write any value to an ignored register and the hardware will ignore the value written. In variable-sized register sets, registers which are unimplemented in a particular processor are also reserved registers. An access to one of these unimplemented registers causes a Reserved Register/Field fault.

Within defined registers, fields which are not defined are either reserved or ignored. For **reserved fields**, hardware will always return a zero on a read. Software must always write zeros to these fields. Any attempt to write a non-zero value into a reserved field will raise a Reserved register/ field fault. **Reserved** fields may have a possible future use.

For **ignored fields**, hardware will return a 0 on a read, unless noted otherwise. Software may write any value to these fields since the hardware will ignore any value written. Except where noted otherwise some IA-32 ignored fields may have a possible future use.

Table 3-1 summarizes how the processor treats reserved and ignored registers and fields.

Table 3-1. Reserved and Ignored Registers and Fields

Type	Read	Write
Reserved register	Illegal operation fault	Illegal operation fault
Ignored register	0	Value written is discarded
Reserved field	0	Write of non-zero causes Reserved Reg/Field fault
Ignored field	0 (unless noted otherwise)	Value written is discarded

For defined fields in registers, values which are not defined are reserved. Software must always write defined values to these fields. Any attempt to write a **reserved value** will raise a Reserved Register/Field fault. Certain registers are **read-only registers**. A write to a read-only register raises an Illegal Operation fault.

When fields are marked as **reserved**, it is essential for compatibility with future processors that software treat these fields as having a future, though unknown effect. Software should follow these guidelines when dealing with **reserved** fields:

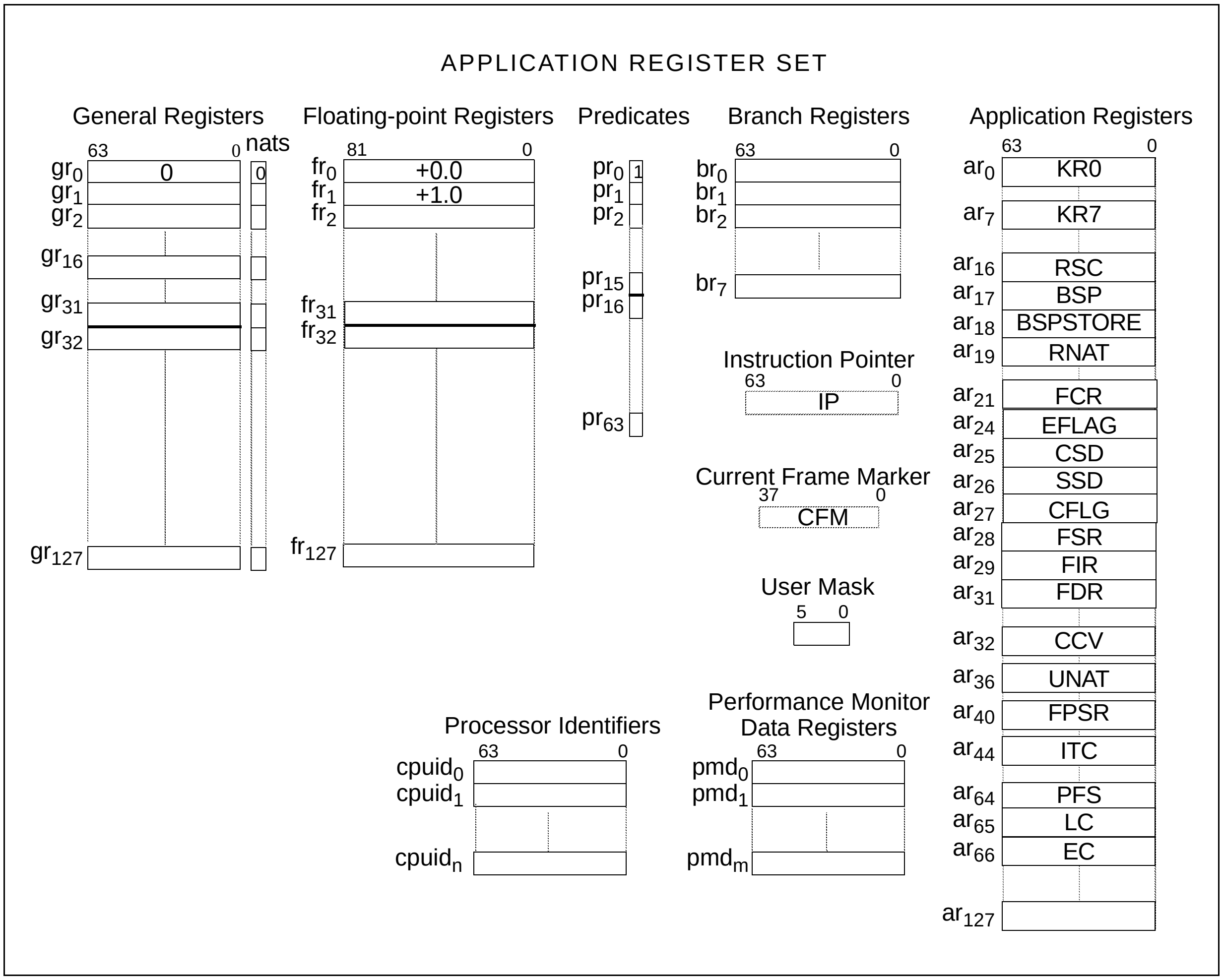
- Do not depend on the state of any reserved fields. Mask all reserved fields before testing.
- Do not depend on the states of any reserved fields when storing to memory or a register.
- Do not depend on the ability to retain information written into reserved or ignored fields.
- Where possible reload reserved or ignored fields with values previously returned from the same register, otherwise load zeros.

3.1.2 General Registers

A set of 128 (64-bit) **general registers** provide the central resource for all integer and integer multimedia computation. They are numbered GR0 through GR127, and are available to all programs at all privilege levels. Each general register has 64 bits of normal data storage plus an additional bit, the **NaT** bit (Not a Thing), which is used to track deferred speculative exceptions.

The general registers are partitioned into two subsets. General registers 0 through 31 are termed the **static general registers**. Of these, GR0 is special in that it always reads as zero when sourced as an operand and attempting to write to GR 0 causes an Illegal Operation fault. General registers 32 through 127 are termed the **stacked general registers**. The stacked registers are made available to a program by allocating a register stack frame consisting of a programmable number of local and output registers. See Chapter 4.1 for a description. A portion of the stacked registers can be programmatically renamed to accelerate loops. See "Modulo-Scheduled Loop Support" on page 4-26.

Figure 3-1. Application Register Model



General registers 8 through 31 contain the IA-32 integer, segment selector and segment descriptor registers when executing IA-32 instructions.

3.1.3 Floating-point Registers

A set of 128 (82-bit) **floating-point registers** are used for all floating-point computation. They are numbered FR0 through FR127, and are available to all programs at all privilege levels. The floating-point registers are partitioned into two subsets. Floating-point registers 0 through 31 are termed the **static floating-point registers**. Of these, FR0 and FR1 are special. FR0 always reads as +0.0 when sourced as an operand, and FR 1 always reads as +1.0. When either of these is used as a destination, a fault is raised. Deferred speculative exceptions are recorded with a special register value called **NaTVal (Not a Thing Value)**.

Floating-point registers 32 through 127 are termed the **rotating floating-point registers**. These registers can be programmatically renamed to accelerate loops. See "Modulo-Scheduled Loop Support" on page 4-26.

Floating-point registers 8 through 31 contain the IA-32 floating-point and multi-media registers when executing IA-32 instructions.

3.1.4 Predicate Registers

A set of 64 (1-bit) **predicate registers** are used to hold the results of IA-64 compare instructions. These registers are numbered PR0 through PR63, and are available to all programs at all privilege levels. These registers are used for conditional execution of instructions.

The predicate registers are partitioned into two subsets. Predicate registers 0 through 15 are termed the **static predicate registers**. Of these, PR0 always reads as '1' when sourced as an operand, and when used as a destination, the result is discarded. The static predicate registers are also used in conditional branching. See "Predication" on page 4-7.

Predicate registers 16 through 63 are termed the **rotating predicate registers**. These registers can be programmatically renamed to accelerate loops. See "Modulo-Scheduled Loop Support" on page 4-26.

3.1.5 Branch Registers

A set of 8 (64-bit) **branch registers** are used to hold IA-64 branching information. They are numbered BR 0 through BR 7, and are available to all programs at all privilege levels. The branch registers are used to specify the branch target addresses for indirect branches. For more information see "Branch Instructions" on page 4-24.

3.1.6 Instruction Pointer

The Instruction Pointer (IP) holds the address of the bundle which contains the current executing IA-64 instruction. The IP can be read directly with a mov ip instruction. The IP cannot be directly written, but is incremented as instructions are executed, and can be set to a new value with a branch. Because IA-64 instruction bundles are 16 bytes, and are 16-byte aligned, the least significant 4 bits of IP are always zero. See "Instruction Encoding Overview" on page 3-14. For IA-32 instruction set execution, IP holds the zero extended 32-bit virtual linear address of the currently executing IA-32 instruction. IA-32 instructions are byte-aligned, therefore the least significant 4 bits of IP are preserved for IA-32 instruction set execution.

3.1.7 Current Frame Marker

Each general register stack frame is associated with a frame marker. The frame marker describes the state of the IA-64 general register stack. The Current Frame Marker (CFM) holds the state of the current stack frame. The CFM cannot be directly read or written (see "Register Stack" on page 4-1).

The frame markers contain the sizes of the various portions of the stack frame, plus three Register Rename Base values (used in register rotation). The layout of the frame markers is shown in Figure 3-2 and the fields are described in Table 3-2.

On a call, the CFM is copied to the Previous Frame Marker field in the Previous Function State register (see Section 3.1.8.10). A new value is written to the CFM, creating a new stack frame with no locals or rotating registers, but with a set of output registers which are the caller's output registers. Additionally, all Register Rename Base registers (RRBs) are set to 0. See "Modulo-Scheduled Loop Support" on page 4-26.

Figure 3-2. Frame Marker Format

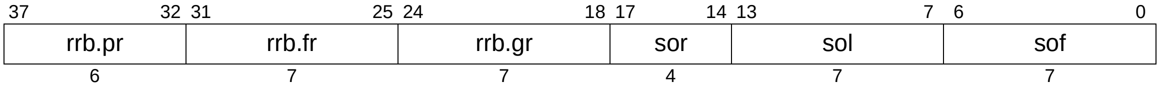


Table 3-2. Frame Marker Field Description

Field	Bit Range	Description
sof	6:0	Size of stack frame
sol	13:7	Size of locals portion of stack frame
sor	17:14	Size of rotating portion of stack frame (the number of rotating registers is 8 * sor)
rrb.gr	24:18	Register Rename Base for general registers
rrb.fr	31:25	Register Rename Base for floating-point registers
rrb.pr	37:32	Register Rename Base for predicate registers

3.1.8 Application Registers

The application register file includes special-purpose data registers and control registers for application-visible processor functions for both the IA-32 and IA-64 instruction sets. These registers can be accessed by IA-64 application software (except where noted). Table 3-3 contains a list of the application registers.

Application registers can only be accessed by either a M or I execution unit. This is specified in the last column of the table. The ignored registers are for future backward-compatible extensions.

Table 3-3. Application Registers

Register	Name	Description	Execution Unit Type
AR 0-7	KR 0-7[a]	Kernel Registers 0-7	M
AR 8-15		Reserved	
AR 16	RSC	Register Stack Configuration Register	
AR 17	BSP	Backing Store Pointer (read-only)	
AR 18	BSPSTORE	Backing Store Pointer for Memory Stores	
AR 19	RNAT	RSE NAT Collection Register	
AR 20		Reserved	
AR 21	FCR	IA-32 Floating-point Control Register	
AR 22 – AR 23		Reserved	
AR 24	EFLAG[b]	IA-32 EFLAG register	
AR 25	CSD	IA-32 Code Segment Descriptor	
AR 26	SSD	IA-32 Stack Segment Descriptor	
AR 27	CFLG[a]	IA-32 Combined CR0 and CR4 register	
AR 28	FSR	IA-32 Floating-point Status Register	
AR 29	FIR	IA-32 Floating-point Instruction Register	
AR 30	FDR	IA-32 Floating-point Data Register	
AR 31		Reserved	
AR 32	CCV	Compare and Exchange Compare Value Register	
AR 33 – AR 35		Reserved	
AR 36	UNAT	User NAT Collection Register	
AR 37 – AR 39		Reserved	
AR 40	FPSR	Floating-point Status Register	
AR 41 – AR 43		Reserved	
AR 44	ITC	Interval Time Counter	
AR 45 – AR 47		Reserved	
AR 48 – AR 63		Ignored	M or I
AR 64	PFS	Previous Function State	I
AR 65	LC	Loop Count Register	
AR 66	EC	Epilog Count Register	
AR 67 – AR 111		Reserved	
AR 112 – AR 127		Ignored	M or I

[a] Writes to these registers when the privilege level is not zero result in a Privileged Register fault. Reads are always allowed.
[b] Some IA-32 EFLAG field writes are silently ignored if the privilege level is not zero.

3.1.8.1 Kernel Registers (KR 0-7 – AR 0-7)

Eight user-visible IA-64 64-bit data kernel registers are provided to convey information from the operating system to the application. These registers can be read at any privilege level but are writable only at the most privileged level. KR0 - KR2 are also used to hold additional IA-32 register state when the IA-32 instruction set is executing.

3.1.8.2 Register Stack Configuration Register (RSC – AR 16)

The Register Stack Configuration (RSC) Register is a 64-bit register used to control the operation of the IA-64 Register Stack Engine (RSE). The RSC format is shown in Figure 3-3 and the field description is contained in Table 3-4. Instructions that modify the RSC can never set the privilege level field to a more privileged level than the currently executing process.

Figure 3-3. RSC Format

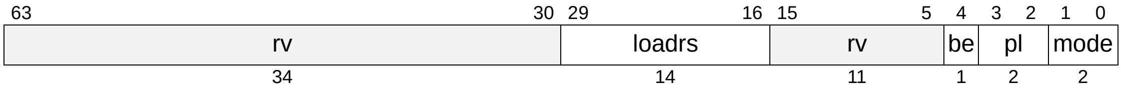


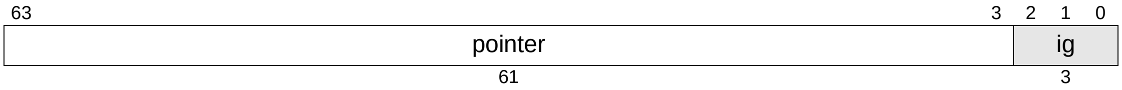
Table 3-4. RSC Field Description

<table>
<tr><th>Field</th><th>Bit Range</th><th colspan="4">Description</th></tr>
<tr><td rowspan="3">mode</td><td rowspan="3">1:0</td><td colspan="4">RSE mode – controls how aggressively the RSE saves and restores register frames. Eager and intensive settings are hints and can be implemented as lazy.</td></tr>
<tr><td>Bit Pattern</td><td>RSE Mode</td><td>Bit 1: eager loads</td><td>Bit 0: eager stores</td></tr>
<tr><td>00
10
01
11</td><td>enforced lazy
load intensive
store intensive
eager</td><td>disabled
enabled
disabled
enabled</td><td>disabled
disabled
enabled
enabled</td></tr>
<tr><td>pl</td><td>3:2</td><td colspan="4">RSE privilege level – loads and stores issued by the RSE are at this privilege level</td></tr>
<tr><td>be</td><td>4</td><td colspan="4">RSE endian mode – loads and stores issued by the RSE use this byte ordering (0: little endian; 1: big endian)</td></tr>
<tr><td>loadrs</td><td>29:16</td><td colspan="4">RSE load distance to tear point – value used in the <code>loadrs</code> instruction for synchronizing the RSE to a tear point</td></tr>
<tr><td>rv</td><td>15:5, 63:30</td><td colspan="4">Reserved</td></tr>
</table>

3.1.8.3 RSE Backing Store Pointer (BSP – AR 17)

The RSE Backing Store Pointer is a 64-bit read-only register (Figure 3-4). It holds the address of the location in memory which is the save location for GR 32 in the current stack frame.

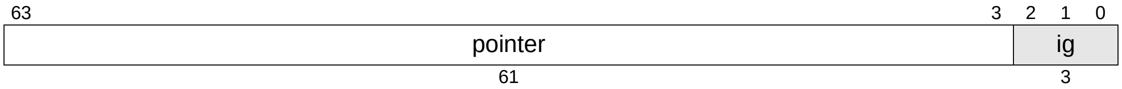
Figure 3-4. BSP Register Format



3.1.8.4 RSE Backing Store Pointer for Memory Stores (BSPSTORE – AR 18)

The RSE Backing Store Pointer for memory stores is a 64-bit register (Figure 3-5). It holds the address of the location in memory to which the RSE will spill the next value.

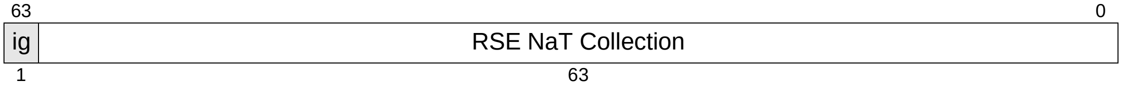
Figure 3-5. BSPSTORE Register Format



3.1.8.5 RSE NAT Collection Register (RNAT – AR 19)

The RSE NaT Collection Register is a 64-bit register (Figure 3-6) used by the RSE to temporarily hold NaT bits when it is spilling general registers. Bit 63 always reads as zero and ignores all writes.

Figure 3-6. RNAT Register Format



3.1.8.6 Compare and Exchange Value Register (CCV – AR 32)

The Compare and Exchange Value Register is a 64-bit register that contains the compare value used as the third source operand in the IA-64 `cmpxchg` instruction.

3.1.8.7 User NAT Collection Register (UNAT – AR 36)

The User NaT Collection Register is a 64-bit register used to temporarily hold NaT bits when saving and restoring general registers with the IA-64 `ld8.fill` and `st8.spill` instructions.

3.1.8.8 Floating-point Status Register (FPSR – AR 40)

The floating-point status register (FPSR) controls traps, rounding mode, precision control, flags, and other control bits for IA-64 floating-point instructions. FPSR does not control or reflect the status of IA-32 floating-point instructions. For more details on the FPSR, see Section 5.2.

3.1.8.9 Interval Time Counter (ITC – AR 44)

The Interval Time Counter (ITC) is a 64-bit register which counts up at a fixed relationship to the processor clock frequency. Applications can directly sample the ITC for time-based calculations and performance measurements. System software can secure the interval time counter from non-privileged IA-64 access. When secured, a read of the ITC at any privilege level other than the most privileged causes a Privileged Register fault. The ITC can be written only at the most privileged level. The IA-32 Time Stamp Counter (TSC) is equivalent to ITC. ITC can directly be read by the IA-32 `rdtsc` (read time stamp counter) instruction. System software can secure the ITC from non-privileged IA-32 access. When secured, an IA-32 read of the ITC at any privilege level other than the most privileged raises an IA-32_Exception(GPfault).

3.1.8.10 Previous Function State (PFS – AR 64)

The IA-64 Previous Function State register (PFS) contains multiple fields: Previous Frame Marker (pfm), Previous Epilog Count (pec), and Previous Privilege Level (ppl). Figure 3-7 diagrams the PFS format and Table 3-5 describes the PFS fields. These values are copied automatically on a call from the CFM register, Epilog Count Register (EC) and PSR.cpl (Current Privilege Level in the Processor Status Register) to accelerate procedure calling.

When an IA-64 `br.call` is executed, the CFM, EC, and PSR.cpl are copied to the PFS and the old contents of the PFS are discarded. When an IA-64 `br.ret` is executed, the PFS is copied to the CFM and EC. PFS.ppl is copied to PSR.cpl, unless this action would increase the privilege level.

The PFS.pfm has the same layout as the CFM (see Section 3.1.7), and the PFS.pec has the same layout as the EC (see Section 3.1.8.12).

Figure 3-7. PFS Format

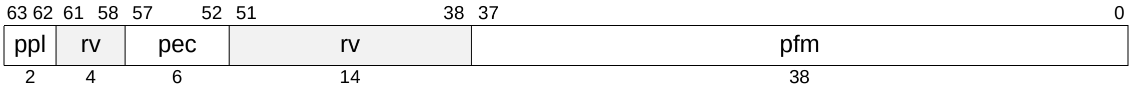


Table 3-5. PFS Field Description

Field	Bit Range	Description
pfm	37:0	Previous Frame Marker
pec	57:52	Previous Epilog Count
ppl	63:62	Previous Privilege Level
rv	51:38, 61:58	Reserved

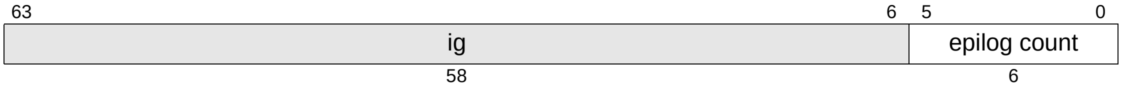
3.1.8.11 Loop Count Register (LC – AR 65)

The Loop Count register (LC) is a 64-bit register used in IA-64 counted loops. LC is decremented by counted-loop-type branches.

3.1.8.12 Epilog Count Register (EC – AR 66)

The Epilog Count register (EC) is a 6-bit register used for counting the final (epilog) stages in IA-64 modulo-scheduled loops. See "Modulo-Scheduled Loop Support" on page 4-26. A diagram of the EC register is shown in Figure 3-8.

Figure 3-8. Epilog Count Register Format



3.1.9 Performance Monitor Data Registers (PMD)

A set of performance monitoring registers can be configured by privileged software to be accessible at all privilege levels. Performance monitor data can be directly sampled from within the application. The operating system is allowed to secure user-configured performance monitors.

Secured performance counters return zeros when read, regardless of the current privilege level. The performance monitors can only be written at the most privileged level. Performance monitors can be used to gather performance information for both IA-32 and IA-64 instruction set execution.

3.1.10 User Mask (UM)

The user mask is a subset of the Processor Status Register and is accessible to IA-64 application programs. The user mask controls memory access alignment, byte-ordering and user-configured performance monitors. It also records the modification state of IA-64 floating-point registers. Figure 3-9 show the user mask format and Table 3-6 describes the user mask fields.

Figure 3-9. User Mask Format

5	4	3	2	1	0
mfh	mfl	ac	up	be	rv
1	1	1	1	1	1

Table 3-6. User Mask Field Descriptions

Field	Bit Range	Description
rv	0	Reserved
be	1	IA-64 Big-endian memory access enable (controls loads and stores but not RSE memory accesses) 0: accesses are done little-endian 1: accesses are done big-endian This bit is ignored for IA-32 data memory accesses. IA-32 data references are always performed little-endian.
up	2	User performance monitor enable for IA-32 and IA-64 instruction set execution 0: user performance monitors are disabled 1: user performance monitors are enabled
ac	3	Alignment check for IA-32 and IA-64 data memory references 0: unaligned data memory references may cause an Unaligned Data Reference fault. 1: all unaligned data memory references cause an Unaligned Data Reference fault.
mfl	4	Lower (f2 .. f31) floating-point registers written – This bit is set to one when an IA-64 instruction that uses register f2..f31 as a target register, completes. This bit is sticky and is only cleared by an explicit write of the user mask.
mfh	5	Upper (f32 .. f127) floating-point registers written – This bit is set to one when an IA-64 instruction that uses register f32..f127 as a target register, completes. This bit is sticky and only cleared by an explicit write of the user mask.

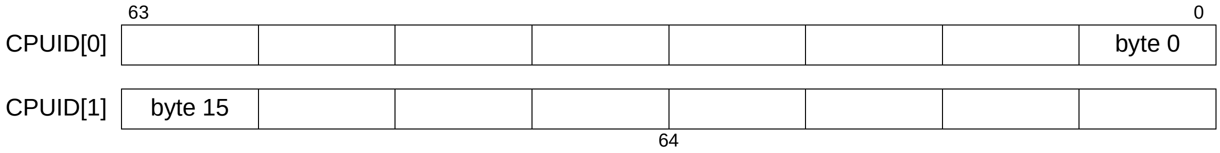
3.1.11 Processor Identification Registers

Application level processor identification information is available in an IA-64 register file termed: CPUID. This register file is divided into a fixed region, registers 0 to 4, and a variable region, register 5 and above. The CPUID[3].number field indicates the maximum number of 8-byte registers containing processor specific information.

The CPUID registers are unprivileged and accessed using the indirect `mov` (from) instruction. All registers beyond register CPUID[3].number are reserved and raise a Reserved Register/Field fault if they are accessed. Writes are not permitted and no instruction exists for such an operation.

Vendor information is located in CPUID registers 0 and 1 and specify a vendor name, in ASCII, for the processor implementation (Figure 3-10). All bytes after the end of the string up to the 16th byte are zero. Earlier ASCII characters are placed in lower number register and lower numbered byte positions.

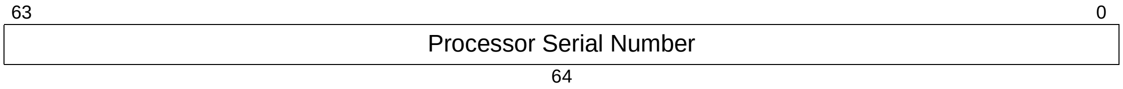
Figure 3-10. CPUID Registers 0 and 1 – Vendor Information



A Processor Serial Number is located in CPUID register 2. If Processor Serial Numbers are supported by the processor model and are not disabled, this register returns a 64-bit number Processor Serial Number (Figure 3-11), otherwise zero is returned. The Processor Serial Number (64-bits) must be combined with the 32-bit version information (CPUID register 3; processor archrev, family, model, and revision numbers) to form a 96-bit Processor Identifier.

The 96-bit Processor Identifier is designed to be unique.

Figure 3-11. CPUID Register 2 – Processor Serial Number



CPUID register 3 contains several fields indicating version information related to the processor implementation. Figure 3-12 and Table 3-7 specify the definitions of each field.

Figure 3-12. CPUID Register 3 – Version Information

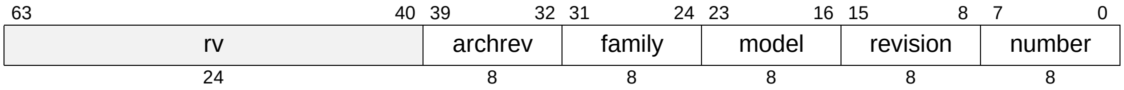
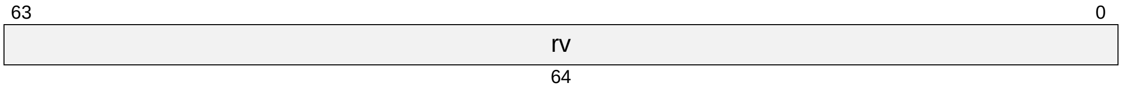


Table 3-7. CPUID Register 3 Fields

Field	Bits	Description
number	7:0	The index of the largest implemented CPUID register (one less than the number of implemented CPUID registers). This value will be at least 4.
revision	15:8	Processor revision number. An 8-bit value that represents the revision or stepping of this processor implementation within the processor model.
model	23:16	Processor model number. A unique 8-bit value representing the processor model within the processor family.
family	31:24	Processor family number. A unique 8-bit value representing the processor family.
archrev	39:32	Architecture revision. An 8-bit value that represents the architecture revision number that the processor implements.
rv	63:40	Reserved.

CPUID register 4 provides general application level information about IA-64 features. As shown in Figure 3-13, it is a set of flag bits used to indicate if a given IA-64 feature is supported in the processor model. When a bit is one the feature is supported; when 0 the feature is not supported. This register does not contain IA-32 instruction set features. IA-32 instruction set features can be acquired by the IA-32 `cpuid` instruction. There are no defined feature bits in the current architecture. As new features are added (or removed) from future processor models the presence (or removal) of new features will be indicated by new feature bits. A value of zero in this register indicates all features defined in the first IA-64 architectural revision are implemented.

Figure 3-13. CPUID Register 4 – General Features/Capability Bits



3.2 Memory

This section describes an IA-64 application program’s view of memory. This includes a description of how memory is accessed, for both 32-bit and 64-bit applications. The size and alignment of addressable units in memory is also given, along with a description of how byte ordering is handled.

3.2.1 Application Memory Addressing Model

Memory is byte addressable and is accessed with 64-bit pointers. A 32-bit pointer model without a hardware mode is supported architecturally. Pointers which are 32 bits in memory are loaded and manipulated in 64-bit registers. Software must explicitly convert 32-bit pointers into 64-bit pointers before use.

3.2.2 Addressable Units and Alignment

Memory can be addressed in units of 1, 2, 4, 8, 10 and 16 bytes.

It is recommended that all addressable units be stored on their naturally aligned boundaries. Hardware and/or operating system software may have support for unaligned accesses, possibly with some performance cost. 10-byte floating-point values should be stored on 16-byte aligned boundaries.

Bits within larger units are always numbered from 0 starting with the least-significant bit. Quantities loaded from memory to general registers are always placed in the least-significant portion of the register (loaded values are placed right justified in the target general register).

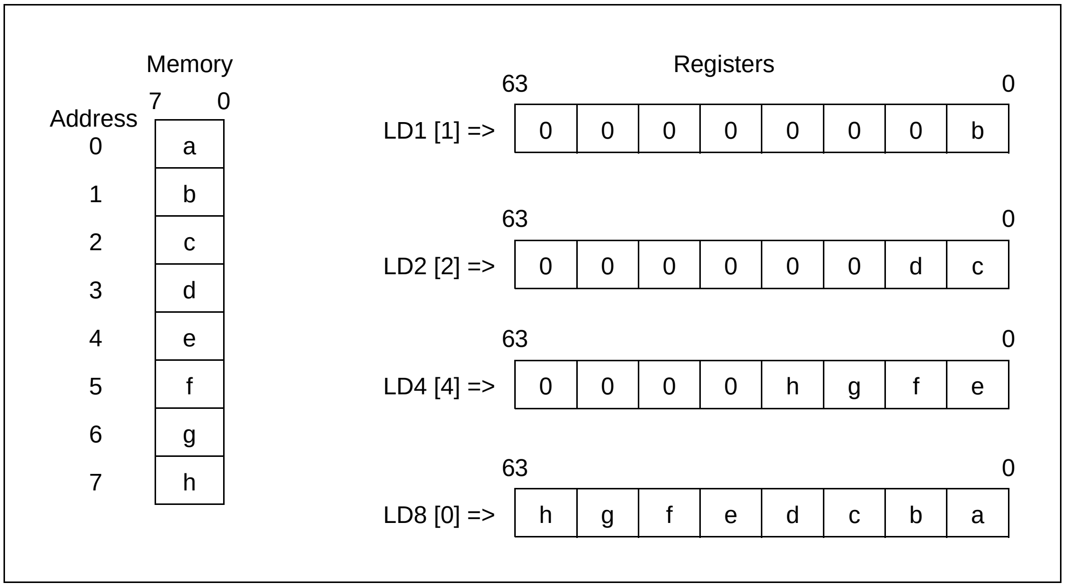
Instruction bundles (3 IA-64 instructions per bundle) are 16-byte units that are always aligned on 16-byte boundaries.

3.2.3 Byte Ordering

The UM.be bit in the User Mask controls whether loads and stores use little-endian or big-endian byte ordering for IA-64 references. When the UM.be bit is 0, larger-than-byte loads and stores are little endian (lower-addressed bytes in memory correspond to the lower-order bytes in the register). When the UM.be bit is 1, larger-than-byte loads and stores are big endian (lower-addressed bytes in memory correspond to the higher-order bytes in the register). Load byte and store byte are not affected by the UM.be bit. The UM.be bit does not affect instruction fetch, IA-32 references, or the RSE. IA-64 instructions are always accessed by the processor as little-endian units. When instructions are referenced as big-endian data, the instruction will appear reversed in a register.

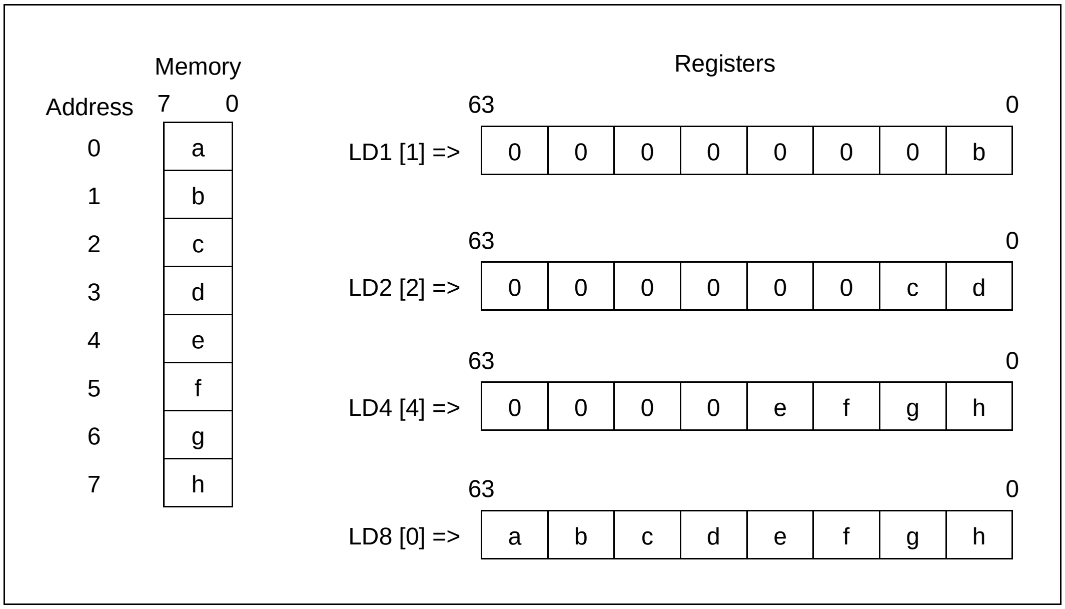
Figure 3-14 shows various loads in little-endian format. Figure 3-15 shows various loads in big endian format. Stores are not shown but behave similarly.

Figure 3-14. Little-endian Loads



Memory
Address
7 0
0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
Registers
63 0
LD1 [1] => 0 0 0 0 0 0 0 b
63 0
LD2 [2] => 0 0 0 0 0 0 d c
63 0
LD4 [4] => 0 0 0 0 h g f e
63 0
LD8 [0] => h g f e d c b a

Figure 3-15. Big-endian Loads



Memory
Address
7 0
0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
Registers
63 0
LD1 [1] => 0 0 0 0 0 0 0 b
63 0
LD2 [2] => 0 0 0 0 0 0 c d
63 0
LD4 [4] => 0 0 0 0 e f g h
63 0
LD8 [0] => a b c d e f g h

3.3 Instruction Encoding Overview

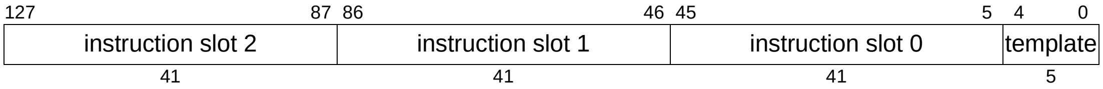
Each IA-64 instruction is categorized into one of six types; each instruction type may be executed on one or more execution unit types. Table 3-8 lists the instruction types and the execution unit type on which they are executed:

Table 3-8. 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

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 3-16.

Figure 3-16. Bundle Format



During execution, architectural **stops** in the program indicate to the hardware that one or more instructions before the stop may have certain kinds of resource dependencies with one or more instructions after the stop. A stop is present after each slot having a double line to the right of it in Table 3-9. For example, template 00 has no stops, while template 03 has a stop after slot 1 and another after slot 2.

In addition to the location of stops, the template field specifies the mapping of instruction slots to execution unit types. Not all possible mappings of instructions to units are available. Table 3-9 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.

Table 3-9. Template Field Encoding and Instruction Slot Mapping[a]

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
05	M-unit	L-unit	X-unit
06			
07			

Table 3-9. Template Field Encoding and Instruction Slot Mapping[a] (Cont'd)

Template	Slot 0	Slot 1	Slot 2
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] Extended instructions, used for long immediate integer, occupy two instruction slots.

3.4 Instruction Sequencing

An IA-64 program consists of a sequence of instructions and stops packed in bundles. Instruction execution is ordered as follows:

- Bundles are ordered from lowest to highest memory address. Instructions in bundles with lower memory addresses are considered to precede instructions in bundles with higher memory addresses. The byte order of each bundle in memory is little-endian (the template field is contained in byte 0 of a bundle).
- Within a bundle, instructions are ordered from instruction slot 0 to instruction slot 2 as specified in Figure 3-16.

For additional details on Instruction sequencing, refer to Appendix A, "Instruction Sequencing Considerations".

IA-64 Application Programming Model 4

This section describes the IA-64 architectural functionality from the perspective of the application programmer. IA-64 instructions are grouped into related functions and an overview of their behavior is given. Unless otherwise noted, all immediates are sign extended to 64 bits before use. The floating-point programming model is described separately in Chapter 5, "IA-64 Floating-point Programming Model".

The main features of the IA-64 programming model covered here are:

- General Register Stack
- Integer Computation Instructions
- Compare Instructions and Predication
- Memory Access Instructions and Speculation
- Branch Instructions and Branch Prediction
- Multimedia Instructions
- Register File Transfer Instructions
- Character Strings and Population Count

4.1 Register Stack

As described in "General Registers" on page 3-2, the general register file is divided into static and stacked subsets. The static subset is visible to all procedures and consists of the 32 registers from GR 0 through GR 31. The stacked subset is local to each procedure and may vary in size from zero to 96 registers beginning at GR 32. The register stack mechanism is implemented by renaming register addresses as a side-effect of procedure calls and returns. The implementation of this rename mechanism is not otherwise visible to application programs. The register stack is disabled during IA-32 instruction set execution.

The static subset must be saved and restored at procedure boundaries according to software convention. The stacked subset is automatically saved and restored by the Register Stack Engine (RSE) without explicit software intervention. All other register files are visible to all procedures and must be saved/restored by software according to software convention.

4.1.1 Register Stack Operation

The registers in the stacked subset visible to a given procedure are called a register stack frame. The frame is further partitioned into two variable-size areas: the local area and the output area. Immediately after a call, the size of the local area of the newly activated frame is zero and the size of the output area is equal to the size of the caller's output area and overlays the caller's output area.

The local and output areas of a frame can be re-sized using the `alloc` instruction which specifies immediates that determine the size of frame (sof) and size of locals (sol).

Note: In the assembly language, `alloc` specifies three operands: the size of inputs immediate; the size of locals immediate; and the size of outputs immediate. The value of sol is determined by adding the size of inputs immediate and the size of locals immediate; the value of sof is determined by adding all three immediates.

The value of sof specifies the size of the entire stacked subset visible to the current procedure; the value of sol specifies the size of the local area. The size of the output area is determined by the difference between sof and sol. The values of these parameters for the currently active procedure are maintained in the Current Frame Marker (CFM).

Reading a stacked register outside the current frame will return an undefined result. Writing a stacked register outside the current frame will cause an Illegal Operation fault.

When a call-type branch is executed, the CFM is copied to the Previous Frame Marker (PFM) field in the Previous Function State application register (PFS), and the callee's frame is created as follows:

- The stacked registers are renamed such that the first register in the caller's output area becomes GR 32 for the callee.
- The size of the local area is set to zero.
- The size of the callee's frame (sof_{b1}) is set to the size of the caller's output area (sof_a – sol_a).

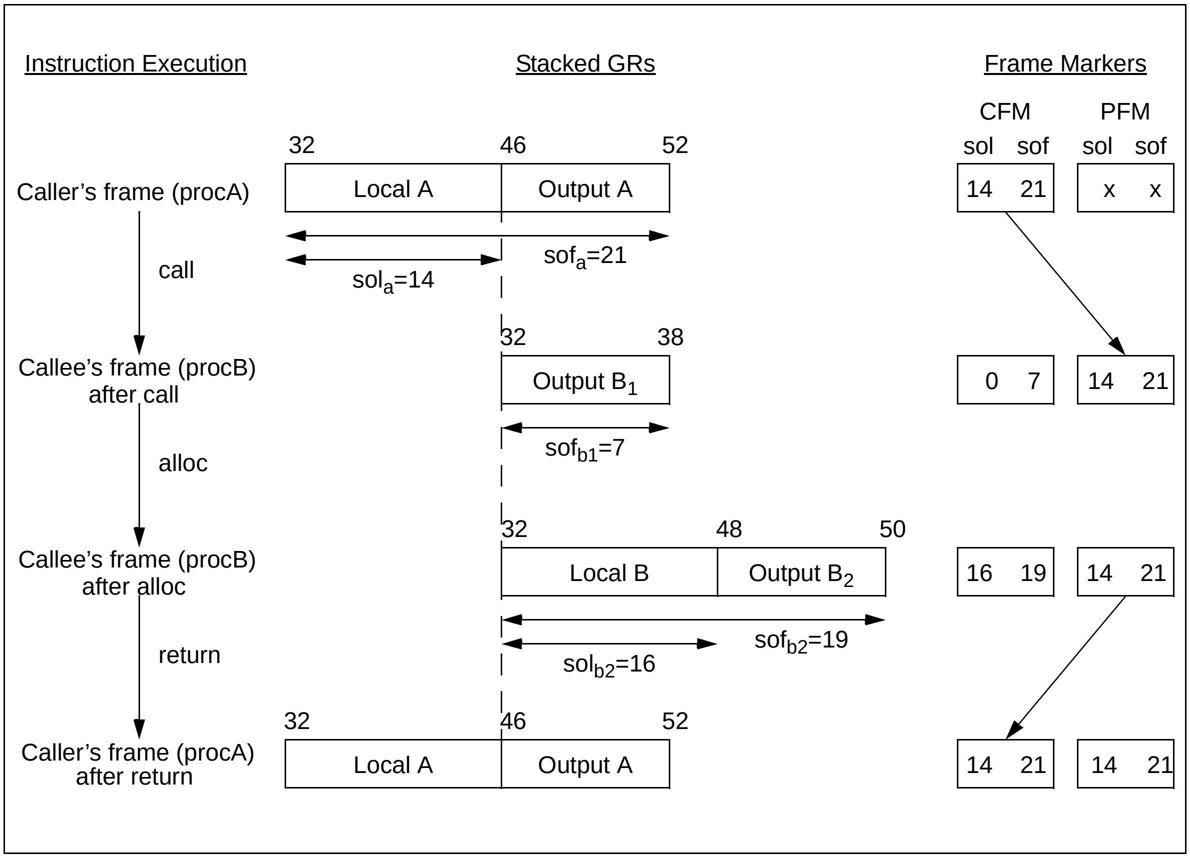
Values in the output area of the caller's register stack frame are visible to the callee. This overlap permits parameter and return value passing between procedures to take place entirely in registers.

Procedure frames may be dynamically re-sized by issuing an `alloc` instruction. An `alloc` instruction causes no renaming, but only changes the size of the register stack frame and the partitioning between local and output areas. Typically, when a procedure is called, it will allocate some number of local registers for its use (which will include the parameters passed to it in the caller's output registers), plus an output area (for passing parameters to procedures it will call). Newly allocated registers (including their NaT bits) have undefined values.

When a return-type branch is executed, CFM is restored from PFM and the register renaming is restored to the caller's configuration. The PFM is procedure local state and must be saved and restored by non-leaf procedures. The CFM is not directly accessible in application programs and is updated only through the execution of calls, returns, `alloc`, and `clrrrb`.

Figure 4-1 depicts the behavior of the register stack on a procedure call from procA (caller) to procB (callee). The state of the register stack is shown at four points: prior to the call, immediately following the call, after procB has executed an `alloc`, and after procB returns to procA.

Figure 4-1. Register Stack Behavior on Procedure Call and Return



The majority of application programs need only issue `alloc` instructions and save/restore PFM in order to effectively utilize the register stack. A detailed knowledge of the RSE (Register Stack Engine) is required only by certain specialized application software such as user-level thread packages, debuggers, etc.

4.1.2 Register Stack Instructions

The `alloc` instruction is used to change the size of the current register stack frame. An `alloc` instruction must be the first instruction in an instruction group otherwise the results are undefined. An `alloc` instruction affects the register stack frame seen by all instructions in an instruction group, including the `alloc` itself. An `alloc` cannot be predicated. An `alloc` does not affect the values or NaT bits of the allocated registers. When a register stack frame is expanded, newly allocated registers may have their NaT bit set.

In addition, there are three instructions which provide explicit control over the state of the register stack. These instructions are used in thread and context switching which necessitate a corresponding switch of the backing store for the register stack.

The `flushrs` instruction is used to force all previous stack frames out to backing store memory. It stalls instruction execution until all active frames in the physical register stack up to, but not including the current frame are spilled to the backing store by the RSE. A `flushrs` instruction must be the first instruction in an instruction group; otherwise, the results are undefined. A `flushrs` cannot be predicated.

Table 4-1 lists the architectural visible state relating to the register stack. Table 4-2 summarizes the register stack management instructions. Call- and return-type branches, which affect the stack, are described in "Branch Instructions" on page 4-24.

Table 4-1. Architectural Visible State Related to the Register Stack

Register	Description
AR[PFS].pfm	Previous Frame Marker field
AR[RSC]	Register Stack Configuration application register
AR[BSP]	Backing store pointer application register
AR[BSPSTORE]	Backing store pointer application register for memory stores
AR[RNAT]	RSE NaT collection application register

Table 4-2. Register Stack Management Instructions

Mnemonic	Operation
`alloc`	Allocate register stack frame
`flushrs`	Flush register stack to backing store

4.2 Integer Computation Instructions

The integer execution units provide a set of arithmetic, logical, shift and bit-field-manipulation instructions. Additionally, they provide a set of instructions to accelerate operations on 32-bit data and pointers.

Arithmetic, logical and 32-bit acceleration instructions can be executed on both I- and M-units

4.2.1 Arithmetic Instructions

Addition and subtraction (`add`, `sub`) are supported with regular two input forms and special three input forms. The three input addition form adds one to the sum of two input registers. The three input subtraction form subtracts one from the difference of two input registers. The three input forms share the same mnemonics as the two input forms and are specified by appending a "1" as a third source operand.

Immediate forms of addition and subtraction use a register and a 15-bit immediate. The immediate form is obtained simply by specifying an immediate rather than a register as the first operand. Also, addition can be performed between a register and a 22-bit immediate; however, the source register must be GR 0, 1, 2 or 3.

A shift left and add instruction (`shladd`) shifts one register operand to the left by 1 to 4 bits and adds the result to a second register operand. Table 4-3 summarizes the integer arithmetic instructions.

Table 4-3. Integer Arithmetic Instructions

Mnemonic	Operation
add	Addition
add ...,1	Three input addition
sub	Subtraction
sub ...,1	Three input subtraction
shladd	Shift left and add

Note that an integer multiply instruction is defined which uses the floating-point registers. See "Integer Multiply and Add Instructions" on page 5-17 for details. Integer divide is performed in software similarly to floating-point divide.

4.2.2 Logical Instructions

Instructions to perform logical AND (and), OR (or), and exclusive OR (xor) between two registers or between a register and an immediate are defined. The andcm instruction performs a logical AND of a register or an immediate with the complement of another register. Table 4-4 summarizes the integer logical instructions.

Table 4-4. Integer Logical Instructions

Mnemonic	Operation
and	Logical and
or	Logical or
andcm	Logical and complement
xor	Logical exclusive or

4.2.3 32-bit Addresses and Integers

Support for IA-64 32-bit addresses is provided in the form of add instructions that perform region bit copying. This supports the virtual address translation model. The add 32-bit pointer instruction (addp) adds two registers or a register and an immediate, zeroes the most significant 32-bits of the result, and copies bits 31:30 of the second source to bits 62:61 of the result. The shladdp instruction operates similarly but shifts the first source to the left by 1 to 4 bits before performing the add, and is provided only in the two-register form.

In addition, support for 32-bit integers is provided through 32-bit compare instructions and instructions to perform sign and zero extension. Compare instructions are described in "Compare Instructions and Predication" on page 4-7. The sign and zero extend (sxt, zxt) instructions take an 8-bit, 16-bit, or 32-bit value in a register, and produce a properly extended 64-bit result.

Table 4-5 summarizes 32-bit pointer and 32-bit integer instructions.

Table 4-5. 32-bit Pointer and 32-bit Integer Instructions

Mnemonic	Operation
`addp`	32-bit pointer addition
`shladdp`	Shift left and add 32-bit pointer
`sxt`	Sign extend
`zxt`	Zero extend

4.2.4 Bit Field and Shift Instructions

Four classes of instructions are defined for shifting and operating on bit fields within a general register: variable shifts, fixed shift-and-mask instructions, a 128-bit-input funnel shift, and special compare operations to test an individual bit within a general register. The compare instructions for testing a single bit (`tbit`), or for testing the NaT bit (`tnat`) are described in "Compare Instructions and Predication" on page 4-7.

The variable shift instructions shift the contents of a general register by an amount specified by another general register. The shift right signed (`shr`) and shift right unsigned (`shr.u`) instructions shift the contents of a register to the right with the vacated bit positions filled with the sign bit or zeroes respectively. The shift left (`shl`) instruction shifts the contents of a register to the left.

The fixed shift-and-mask instructions (`extr`, `dep`) are generalized forms of fixed shifts. The extract instruction (`extr`) copies an arbitrary bit field from a general register to the least-significant bits of the target register. The remaining bits of the target are written with either the sign of the bit field (`extr`) or with zero (`extr.u`). The length and starting position of the field are specified by two immediates. This is essentially a shift-right-and-mask operation. A simple right shift by a fixed amount can be specified by using `shr` with an immediate value for the shift amount. This is just an assembly pseudo-op for an extract instruction where the field to be extracted extends all the way to the left-most register bit.

The deposit instruction (`dep`) takes a field from either the least-significant bits of a general register, or from an immediate value of all zeroes or all ones, places it at an arbitrary position, and fills the result to the left and right of the field with either bits from a second general register (`dep`) or with zeroes (`dep.z`). The length and starting position of the field are specified by two immediates. This is essentially a shift-left-mask-merge operation. A simple left shift by a fixed amount can be specified by using `shl` with an immediate value for the shift amount. This is just an assembly pseudo-op for `dep.z` where the deposited field extends all the way to the left-most register bit.

The shift right pair (`shrp`) instruction performs a 128-bit-input funnel shift. It extracts an arbitrary 64-bit field from a 128-bit field formed by concatenating two source general registers. The starting position is specified by an immediate. This can be used to accelerate the adjustment of unaligned data. A bit rotate operation can be performed by using `shrp` and specifying the same register for both operands.

Table 4-6 summarizes the bit field and shift instructions.

Table 4-6. Bit Field and Shift Instructions

Mnemonic	Operation
shr	Shift right signed
shr.u	Shift right unsigned
shl	Shift left
extr	Extract signed (shift right and mask)
extr.u	Extract unsigned (shift right and mask)
dep	Deposit (shift left, mask and merge)
dep.z	Deposit in zeroes (shift left and mask)
shrp	Shift right pair

4.2.5 Large Constants

A special instruction is defined for generating large constants (see Table 4-7). For constants up to 22 bits in size, the add instruction can be used, or the mov pseudo-op (pseudo-op of add with GR0, which always reads 0). For larger constants, the move long immediate instruction (movl) is defined to write a 64-bit immediate into a general register. This instruction occupies two instruction slots within the same bundle, and is the only such instruction.

Table 4-7. Instructions to Generate Large Constants

Mnemonic	Operation
mov	Move 22-bit immediate
movl	Move 64-bit immediate

4.3 Compare Instructions and Predication

A set of compare instructions provides the ability to test for various conditions and affect the dynamic execution of instructions. A compare instruction tests for a single specified condition and generates a boolean result. These results are written to predicate registers. The predicate registers can then be used to affect dynamic execution in two ways: as conditions for conditional branches, or as qualifying predicates for predication.

4.3.1 Predication

Predication is the conditional execution of instructions. The execution of most IA-64 instructions is gated by a qualifying predicate. If the predicate is true, the instruction executes normally; if the predicate is false, the instruction does not modify architectural state (except for the unconditional type of compare instructions, floating-point approximation instructions and while-loop branches). Predicates are one-bit values and are stored in the predicate register file. A zero predicate is interpreted as false and a one predicate is interpreted as true (predicate register PR0 is hardwired to one).

A few IA-64 instructions cannot be predicated. These instructions are: allocate stack frame (alloc), clear rrb (clrrrb), flush register stack (flushrs), and counted branches (cloop, ctop, cexit).

4.3.2 Compare Instructions

Predicate registers are written by the following instructions: general register compare (cmp, cmp4), floating-point register compare (fcmp), test bit and test NaT (tbit, tnat), floating-point class (fclass), and floating-point reciprocal approximation and reciprocal square root approximation (frcpa, frsqrta). Most of these compare instructions (all but frcpa and frsqrta) set two predicate registers based on the outcome of the comparison. The setting of the two target registers is described below in "Compare Types" on page 4-8. Compare instructions are summarized in Table 4-8.

Table 4-8. Compare Instructions

Mnemonic	Operation
cmp, cmp4	GR compare
tbit	Test bit in a GR
tnat	Test GR NaT bit
fcmp	FR compare
fclass	FR class
frcpa, fprcpa	Floating-point reciprocal approximation
frsqrta, fprsqrta	Floating-point reciprocal square root approximation

The 64-bit (cmp) and 32-bit (cmp4) compare instructions compare two registers, or a register and an immediate, for one of ten relations (e.g., >, <=). The compare instructions set two predicate targets according to the result. The cmp4 instruction compares the least-significant 32-bits of both sources (the most significant 32-bits are ignored).

The test bit (tbit) instruction sets two predicate registers according to the state of a single bit in a general register (the position of the bit is specified by an immediate). The test NaT (tnat) instruction sets two predicate registers according to the state of the NaT bit corresponding to a general register.

The fcmp instruction compares two floating-point registers and sets two predicate targets according to one of eight relations. The fclass instruction sets two predicate targets according to the classification of the number contained in the floating-point register source.

The frcpa and frsqrta instructions set a single predicate target if their floating-point register sources are such that a valid approximation can be produced, otherwise the predicate target is cleared.

4.3.3 Compare Types

Compare instructions can have as many as five compare types: Normal, Unconditional, AND, OR, or DeMorgan. The type defines how the instruction writes its target predicate registers based on the outcome of the comparison and on the qualifying predicate. The description of these types is contained in Table 4-9. In the table, "qp" refers to the value of the qualifying predicate of the compare and "result" refers to the outcome of the compare relation (one if the compare relation is true and zero if the compare relation is false).

Table 4-9. Compare Type Function

Compare Type	Completer	Operation	
		First Predicate Target	Second Predicate Target
Normal	*none*	if (qp) {target = result}	if (qp) {target = !result}

Unconditional	unc	if (qp) {target = result}
		else {target = 0}
			if (qp) {target = !result}
			else {target = 0}

AND	and	if (qp && !result) {target = 0}	if (qp && !result) {target = 0}
	andcm	if (qp && result) {target = 0}	if (qp && result) {target = 0}
OR	or	if (qp && result) {target = 1}	if (qp && result) {target = 1}
	orcm	if (qp && !result) {target = 1}	if (qp && !result) {target = 1}
DeMorgan	or.andcm	if (qp && result) {target = 1}	if (qp && result) {target = 0}
	and.orcm	if (qp && !result) {target = 0}	if (qp && !result) {target = 1}

The Normal compare type simply writes the compare result to the first predicate target and the complement of the result to the second predicate target.

The Unconditional compare type behaves the same as the Normal type, except that if the qualifying predicate is 0, both predicate targets are written with 0. This can be thought of as an initialization of the predicate targets, combined with a Normal compare. Note that compare instructions with the Unconditional type modify architectural state when their qualifying predicate is false.

The AND, OR and DeMorgan types are termed "parallel" compare types because they allow multiple simultaneous compares (of the same type) to target a single predicate register. This provides the ability to compute a logical equation such as `p5 = (r4 == 0) || (r5 == r6)` in a single cycle (assuming p5 was initialized to 0 in an earlier cycle). The DeMorgan compare type is just a combination of an OR type to one predicate target and an AND type to the other predicate target. Multiple OR-type compares (including the OR part of the DeMorgan type) may specify the same predicate target in the same instruction group. Multiple AND-type compares (including the AND part of the DeMorgan type) may also specify the same predicate target in the same instruction group.

For all compare instructions (except for `tnat` and `fclass`), if one or both of the source registers contains a deferred exception token (NaT or NaTVal – see "Control Speculation" on page 4-13), the result of the compare is different. Both predicate targets are treated the same, and are either written to 0 or left unchanged. In combination with speculation, this allows predicated code to be turned off in the presence of a deferred exception. (`fclass` behaves this way as well if NaTVal is not one of the classes being tested for.) Table 4-10 describes the behavior.

Table 4-10. Compare Outcome with NaT Source Input

Compare Type	Operation
Normal	if (qp) {target = 0}
Unconditional	target = 0
AND	if (qp) {target = 0}
OR	(not written)
DeMorgan	(not written)

Only a subset of the compare types are provided for some of the compare instructions. Table 4-11 lists the compare types which are available for each of the instructions.

Table 4-11. Instructions and Compare Types Provided

Instruction	Relation	Types Provided
`cmp`, `cmp4`	a == b, a != b, a > 0, a >= 0, a < 0, a <= 0, 0 > a, 0 >= a, 0 < a, 0 <= a	Normal, Unconditional, AND, OR, DeMorgan
	All other relations	Normal, Unconditional
`tbit`, `tnat`	All	Normal, Unconditional, AND, OR, DeMorgan
`fcmp`, `fclass`	All	Normal, Unconditional
`frcpa`, `frsqrta`, `fprcpa`, `fprsqrta`	Not Applicable	Unconditional

4.3.4 Predicate Register Transfers

Instructions are provided to transfer between the predicate register file and a general register. These instructions operate in a "broadside" manner whereby multiple predicate registers are transferred in parallel, such that predicate register N is transferred to/from bit N of a general register.

The move to predicates instruction (`mov pr=`) loads multiple predicate registers from a general register according to a mask specified by an immediate. The mask contains one bit for each of PR 1 through PR 15 (PR 0 is hardwired to 1) and one bit for all of PR 16 through PR63 (the rotating predicates). A predicate register is written from the corresponding bit in a general register if the corresponding mask bit is 1; if the mask bit is 0 the predicate register is not modified.

The move to rotating predicates instruction (`mov pr.rot=`) copies 48 bits from an immediate value into the 48 rotating predicates (PR 16 through PR 63). The immediate value includes 28 bits, and is sign-extended. Thus PR 16 through PR 42 can be independently set to new values, and PR 43 through PR 63 are all set to either 0 or 1.

The move from predicates instruction (`mov =pr`) transfers the entire predicate register file into a general register target.

For all of these predicate register transfers, the predicate registers are accessed as though the register rename base (CFM.rrb.pr) were 0. Typically, therefore, software should clear CFM.rrb.pr before initializing rotating predicates.

4.4 Memory Access Instructions

Memory is accessed by simple load, store and semaphore instructions, which transfer data to and from general registers or floating-point registers. The memory address is specified by the contents of a general register.

Most load and store instructions can also specify base-address-register update. Base update adds either an immediate value or the contents of a general register to the address register, and places the result back in the address register. The update is done after the load or store operation, i.e., it is performed as an address post-increment.

For highest performance, data should be aligned on natural boundaries. Within a 4K-byte boundary, accesses misaligned with respect to their natural boundaries will always fault if UM.ac (alignment check bit in the User Mask register) is 1. If UM.ac is 0, then an unaligned access will succeed if it is supported by the implementation; otherwise it will cause an Unaligned Data Reference fault. All memory accesses that cross a 4K-byte boundary will cause an Unaligned Data Reference fault independent of UM.ac. Additionally, all semaphore instructions will cause an Unaligned Data Reference fault if the access is not aligned to its natural boundary, independent of UM.ac.

Accesses to memory quantities larger than a byte may be done in a big-endian or little-endian fashion. The byte ordering for all memory access instructions is determined by UM.be in the User Mask register for IA-64 memory references. All IA-32 memory references are performed little-endian.

Load, store and semaphore instructions are summarized in Table 4-12.

Table 4-12. Memory Access Instructions

Mnemonic			Operation
General	Floating-point		
	Normal	Load Pair	
`ld`	`ldf`	`ldfp`	Load
`ld.s`	`ldf.s`	`ldfp.s`	Speculative load
`ld.a`	`ldf.a`	`ldfp.a`	Advanced load
`ld.sa`	`ldf.sa`	`ldfp.sa`	Speculative advanced load

`ld.c.nc,`
`ld.c.clr`
	`ldf.c.nc,`
	`ldf.c.clr`
		`ldfp.c.nc,`
		`ldfp.c.clr`
			Check load

`ld.c.clr.acq`			Ordered check load
`ld.acq`			Ordered load
`ld.bias`			Biased load
`ld8.fill`	`ldf.fill`		Fill
`st`	`stf`		Store
`st.rel`			Ordered store
`st.spill`	`stf.spill`		Spill
`cmpxchg`			Compare and exchange
`xchg`			Exchange memory and GR
`fetchadd`			Fetch and add

4.4.1 Load Instructions

Load instructions transfer data from memory to a general register, a floating-point register or a pair of floating-point registers.

For general register loads, access sizes of 1, 2, 4, and 8 bytes are defined. For sizes less than eight bytes, the loaded value is zero extended to 64-bits.

For floating-point loads, five access sizes are defined: single precision (4 bytes), double precision (8 bytes), double-extended precision (10 bytes), single precision pair (8 bytes), and double precision pair (16 bytes). The value(s) loaded from memory are converted into floating-point register format (see “Memory Access Instructions” on page 5-7 for details). The floating-point load pair instructions load two adjacent single or double precision numbers into two independent floating-point registers (see the `ldfp[s/d]` instruction description for restrictions on target register specifiers). The floating-point load pair instructions cannot specify base register update.

Variants of both general and floating-point register loads are defined for supporting compiler-directed control and data speculation. These use the general register NaT bits and the ALAT. See "Control Speculation" on page 4-13 and "Data Speculation" on page 4-16.

Variants are also provided for controlling the memory/cache subsystem. An ordered load can be used to force ordering in memory accesses. See "Memory Access Ordering" on page 4-23. A biased load provides a hint to acquire exclusive ownership of the accessed line. See "Memory Hierarchy Control and Consistency" on page 4-20.

Special-purpose loads are defined for restoring register values that were spilled to memory. The `ld8.fill` instruction loads a general register and the corresponding NaT bit (defined for an 8-byte access only). The `ldf.fill` instruction loads a value in floating-point register format from memory without conversion (defined for 16-byte access only). See"Register Spill and Fill" on page 4-15.

4.4.2 Store Instructions

Store instructions transfer data from a general or floating-point register to memory. Store instructions are always non-speculative. Store instructions can specify base-address-register update, but only by an immediate value. A variant is also provided for controlling the memory/cache subsystem. An ordered store can be used to force ordering in memory accesses.

Both general and floating-point register stores are defined with the same access sizes as their load counterparts. The only exception is that there are no floating-point store pair instructions.

Special purpose stores are defined for spilling register values to memory. The `st8.spill` instruction stores a general register and the corresponding NaT bit (defined for 8-byte access only). This allows the result of a speculative calculation to be spilled to memory and restored. The `stf.spill` instruction stores a floating-point register in memory in the floating-point register format without conversion. This allows register spill and restore code to be written to be compatible with possible future extensions to the floating-point register format. The `stf.spill` instruction also does not fault if the register contains a NaTVal, and is defined for 16-byte access only. See"Register Spill and Fill" on page 4-15.

4.4.3 Semaphore Instructions

Semaphore instructions atomically load a general register from memory, perform an operation and then store a result to the same memory location. Semaphore instructions are always non-speculative. No base register update is provided.

Three types of atomic semaphore operations are defined: exchange (`xchg`); compare and exchange (`cmpxchg`); and fetch and add (`fetchadd`).

The `xchg` target is loaded with the zero-extended contents of the memory location addressed by the first source and then the second source is stored into the same memory location.

The `cmpxchg` target is loaded with the zero-extended contents of the memory location addressed by the first source; if the zero-extended value is equal to the contents of the Compare and Exchange Compare Value application register (CCV), then the second source is stored into the same memory location.

The `fetchadd` instruction specifies one general register source, one general register target, and an immediate. The `fetchadd` target is loaded with the zero-extended contents of the memory location addressed by the source and then the immediate is added to the loaded value and the result is stored into the same memory location.

Table 4-13. State Relating to Memory Access

Register	Function
UM.be	User mask byte ordering
UM.ac	User mask Unaligned Data Reference fault enable
UNAT	GR NaT collection
CCV	Compare and Exchange Compare Value application register

4.4.4 Control Speculation

Special mechanisms are provided to allow for compiler-directed speculation. This speculation takes two forms, control speculation and data speculation, with a separate mechanism to support each. See also "Data Speculation" on page 4-16.

4.4.4.1 Control Speculation Concepts

Control speculation describes the compiler optimization where an instruction or a sequence of instructions is executed before it is known that the dynamic control flow of the program will actually reach the point in the program where the sequence of instructions is needed. This is done with instruction sequences that have long execution latencies. Starting the execution early allows the compiler to overlap the execution with other work, increasing the parallelism and decreasing overall execution time. The compiler performs this optimization when it determines that it is very likely that the dynamic control flow of the program will eventually require this calculation. In cases where the control flow is such that the calculation turns out not to be needed, its results are simply discarded (the results in processor registers are simply not used).

Since the speculative instruction sequence may not be required by the program, no exceptions encountered that would be visible to the program can be signalled until it is determined that the program's control flow does require the execution of this instruction sequence. For this reason, a mechanism is provided for recording the occurrence of an exception so that it can be signalled later if and when it is necessary. In such a situation, the exception is said to be deferred. When an exception is deferred by an instruction, a special token is written into the target register to indicate the existence of a deferred exception in the program.

Deferred exception tokens are represented differently in the general and floating-point register files. In general registers, an additional bit is defined for each register called the NaT bit (Not a Thing). Thus general registers are 65 bits wide. A NaT bit equal to 1 indicates that the register contains a deferred exception token, and that its 64-bit data portion contains an implementation specific value that software cannot rely upon. In floating-point registers, a deferred exception is indicated by a specific pseudo-zero encoding called the NaTVal (see "Representation of Values in Floating-point Registers" on page 5-2 for details).

4.4.4.2 Control Speculation and Instructions

Instructions are divided into two categories: speculative (instructions which can be used speculatively) and non-speculative (instructions which cannot). Non-speculative instructions will raise exceptions if they occur and are therefore unsafe to schedule before they are known to be executed. Speculative instructions defer exceptions (they do not raise them) and are therefore safe to schedule before they are know to be executed.

Loads to general and floating-point registers have both non-speculative (`ld`, `ldf`, `ldfp`) and speculative (`ld.s`, `ldf.s`, `ldfp.s`) variants. Generally, all computation instructions which write their results to general or floating-point registers are speculative. Any instruction that modifies state other than a general or floating-point register is non-speculative, since there would be no way to represent the deferred exception (there are a few exceptions).

Deferred exception tokens propagate through the program in a dataflow manner. A speculative instruction that reads a register containing a deferred exception token will propagate a deferred exception token into its target. Thus a chain of instructions can be executed speculatively, and only the result register need be checked for a deferred exception token to determine whether any exceptions occurred.

At the point in the program when it is known that the result of a speculative calculation is needed, a speculation check (`chk.s`) instruction is used. This instruction tests for a deferred exception token. If none is found, then the speculative calculation was successful, and execution continues normally. If a deferred exception token is found, then the speculative calculation was unsuccessful and must be re-done. In this case, the `chk.s` instruction branches to a new address (specified by an immediate offset in the `chk.s` instruction). Software can use this mechanism to invoke code that contains a copy of the speculative calculation (but with non-speculative loads). Since it is now known that the calculation is required, any exceptions which now occur can be signalled and handled normally.

Since computational instructions do not generally cause exceptions, the only instructions which generate deferred exception tokens are speculative loads. (IEEE floating-point exceptions are handled specially through a set of alternate status fields. See "Floating-point Status Register" on page 5-5.) Other speculative instructions propagate deferred exception tokens, but do not generate them.

4.4.4.3 Control Speculation and Compares

As stated earlier, most instructions that write a register file other than the general registers or the floating-point registers are non-speculative. The compare (`cmp`, `cmp4`, `fcmp`), test bit (`tbit`), floating-point class (`fclass`), and floating-point approximation (`frcpa`, `frsqrta`) instructions are special cases. These instructions read general or floating-point registers and write one or two predicate registers.

For these instructions, if any source contains a deferred exception token, all predicate targets are either cleared or left unchanged, depending on the compare type (see Table 4-10 on page 4-9). Software can use this behavior to ensure that any dependent conditional branches are not taken and any dependent predicated instructions are nullified. See "Predication" on page 4-7.

Deferred exception tokens can also be tested for with certain compare instructions. The test NaT (`tnat`) instruction tests the NaT bit corresponding to the specified general register and writes two predicate results. The floating-point class (`fclass`) instruction can be used to test for a NaTVal in a floating-point register and write the result to two predicate registers. (`fclass` does not clear both predicate targets in the presence of a NaTVal input if NaTVal is one of the classes being tested for.)

4.4.4.4 Control Speculation without Recovery

A non-speculative instruction that reads a register containing a deferred exception token will raise a Register NaT Consumption fault. Such instructions can be thought of as performing a non-recoverable speculation check operation. In some compilation environments, it may be true that the only exceptions that are deferred are fatal errors. In such a program, if the result of a speculative calculation is checked and a deferred exception token is found, execution of the program is terminated. For such a program, the results of speculative calculations can be checked simply by using non-speculative instructions.

4.4.4.5 Register Spill and Fill

Special store and load instructions are provided for spilling a register to memory and preserving any deferred exception token, and for restoring a spilled register.

The spill and fill general register instructions (`st8.spill`, `ld8.fill`) are defined to save/restore a general register along with the corresponding NaT bit.

The `st8.spill` instruction writes a general register's NaT bit into the User NaT Collection application register (UNAT), and, if the NaT bit was 0, writes the register's 64-bit data portion to memory. If the register's NaT bit was 1, the UNAT is updated, but the memory update is implementation specific, and must consistently follow one of three spill behaviors:

- The `st8.spill` may not update memory with the register's 64-bit data portion, or
- The `st8.spill` may write a zero to the specified memory location, or
- The `st8.spill` may write the register's 64-bit data portion to memory, only if that implementation returns a zero into the target register of all NaTed speculative loads, and that implementation also guarantees that all NaT propagating instructions perform all computations as specified by the instruction pages.

Bits 8:3 of the memory address determine which bit in the UNAT register is written.

The `ld8.fill` instruction loads a general register from memory taking the corresponding NaT bit from the bit in the UNAT register addressed by bits 8:3 of the memory address. The UNAT register must be saved and restored by software. It is the responsibility of software to ensure that the contents of the UNAT register are correct while executing `st8.spill` and `ld8.fill` instructions.

The floating-point spill and fill instructions (`stf.spill`, `ldf.fill`) are defined to save/restore a floating-point register (saved as 16 bytes) without surfacing an exception if the FR contains a NaTVal (these instructions do not affect the UNAT register).

The general and floating-point spill/fill instructions allow spilling/filling of registers that are targets of a speculative instruction and may therefore contain a deferred exception token. Note also that transfers between the general and floating-point register files cause a conversion between the two deferred exception token formats.

Table 4-14 lists the state relating to control speculation. Table 4-15 summarizes the instructions related to control speculation.

Table 4-14. State Related to Control Speculation

Register	Description
NaT bits	65th bit associated with each GR indicating a deferred exception
NaTVal	Pseudo-Zero encoding for FR indicating a deferred exception
UNAT	User NaT collection application register

Table 4-15. Instructions Related to Control Speculation

Mnemonic	Operation
`ld.s, ldf.s, ldfp.s`	GR and FR speculative loads
`ld8.fill, ldf.fill`	Fill GR with NaT collection, fill FR
`st8.spill, stf.spill`	Spill GR with NaT collection, spill FR
`chk.s`	Test GR or FR for deferred exception token
`tnat`	Test GR NaT bit and set predicate

4.4.5 Data Speculation

Just as control speculative loads and checks allow the compiler to schedule instructions across control dependences, data speculative loads and checks allow the compiler to schedule instructions across some types of ambiguous data dependences. This section details the usage model and semantics of data speculation and related instructions.

4.4.5.1 Data Speculation Concepts

An ambiguous memory dependence is said to exist between a store (or any operation that may update memory state) and a load when it cannot be statically determined whether the load and store might access overlapping regions of memory. For convenience, a store that cannot be statically disambiguated relative to a particular load is said to be ambiguous relative to that load. In such cases, the compiler cannot change the order in which the load and store instructions were originally specified in the program. To overcome this scheduling limitation, a special kind of load instruction called an advanced load can be scheduled to execute earlier than one or more stores that are ambiguous relative to that load.

As with control speculation, the compiler can also speculate operations that are dependent upon the advanced load and later insert a check instruction that will determine whether the speculation was successful or not. For data speculation, the check can be placed anywhere the original non-data speculative load could have been scheduled.

Thus, a data-speculative sequence of instructions consists of an advanced load, zero or more instructions dependent on the value of that load, and a check instruction. This means that any sequence of stores followed by a load can be transformed into an advanced load followed by a sequence of stores followed by a check. The decision to perform such a transformation is highly dependent upon the likelihood and cost of recovering from an unsuccessful data speculation.

4.4.5.2 Data Speculation and Instructions

Advanced loads are available in integer (`ld.a`), floating-point (`ldf.a`), and floating-point pair (`ldfp.a`) forms. When an advanced load is executed, it allocates an entry in a structure called the Advanced Load Address Table (ALAT). Later, when a corresponding check instruction is executed, the presence of an entry indicates that the data speculation succeeded; otherwise, the speculation failed and one of two kinds of compiler-generated recovery is performed:

1. The check load instruction (`ld.c`, `ldf.c`, or `ldfp.c`) is used for recovery when the only instruction scheduled before a store that is ambiguous relative to the advanced load is the advanced load itself. The check load searches the ALAT for a matching entry. If found, the speculation was successful; if a matching entry was not found, the speculation was unsuccessful and the check load reloads the correct value from memory. Figure 4-2 shows this transformation.

Figure 4-2. Data Speculation Recovery Using ld.c

Before Data Speculation	After Data Speculation
`// other instructions`	`ld8.a  r6 = [r8];;   // advanced load`
`st8    [r4] = r12`	`// other instructions`
`ld8    r6 = [r8];;`	`st8    [r4] = r12`
`add    r5 = r6, r7;;`	`ld8.c.clr r6 = [r8] // check load`
`st8    [r18] = r5`	`add    r5 = r6, r7;;`
	`st8    [r18] = r5`

2. The advanced load check (`chk.a`) is used when an advanced load and several instructions that depend on the loaded value are scheduled before a store that is ambiguous relative to the advanced load. The advanced load check works like the speculation check (`chk.s`) in that, if the speculation was successful, execution continues inline and no recovery is necessary; if speculation was unsuccessful, the `chk.a` branches to compiler-generated recovery code. The recovery code contains instructions that will re-execute all the work that was dependent on the failed data speculative load up to the point of the check instruction. As with the check load, the success of a data speculation using an advanced load check is determined by searching the ALAT for a matching entry. This transformation is shown in Figure 4-3.

Figure 4-3. Data Speculation Recovery Using chk.a

Before Data Speculation	After Data Speculation
`// other instructions`	`    ld8.a  r6 = [r8];;`
`st8    [r4] = r12`	`    // other instructions`
`ld8    r6 = [r8];;`	`    add    r5 = r6, r7;;`
`add    r5 = r6, r7;;`	`    // other instructions`
`st8    [r18] = r5`	`    st8    [r4] = r12`
	`    chk.a.clr r6, recover`
	`back:`
	`    st8    [r18] = r5`
	`    // somewhere else in program`
	`recover:`
	`    ld8    r6 = [r8];;`
	`    add    r5 = r6, r7`
	`    br     back`

Recovery code may use either a normal or advanced load to obtain the correct value for the failed advanced load. An advanced load is used only when it is advantageous to have an ALAT entry reallocated after a failed speculation. The last instruction in the recovery code should branch to the instruction following the `chk.a`.

4.4.5.3 Detailed Functionality of the ALAT and Related Instructions

The ALAT is the structure that holds the state necessary for advanced loads and checks to operate correctly. The ALAT is searched in two different ways: by physical addresses and by ALAT register tags. An ALAT register tag is a unique number derived from the physical target register number and type in conjunction with other implementation-specific state. Implementation-specific state might include register stack wrap-around information to distinguish one instance of a physical register that may have been spilled by the RSE from the current instance of that register, thus avoiding the need to purge the ALAT on all register stack wrap-arounds.

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. On entry to IA-32 instruction set, existing entries in the ALAT are ignored.

4.4.5.3.1 Allocating and Checking ALAT Entries

Advanced loads perform the following actions:

1. The ALAT register tag for the advanced load is computed. (For `ldfp.a`, a tag is computed only for the first target register.)
2. If an entry with a matching ALAT register tag exists, it is removed.
3. A new entry is allocated in the ALAT which contains the new ALAT register tag, the load access size, and a tag derived from the physical memory address.
4. The value at the address specified in the advanced load is loaded into the target register and, if specified, the base register is updated and an implicit prefetch is performed.

Since the success of a check is determined by finding a matching register tag in the ALAT, both the `chk.a` and the target register of a `ld.c` must specify the same register as their corresponding advanced load. Additionally, the check load must use the same address and operand size as the corresponding advanced load; otherwise, the value written into the target register by the check load is undefined.

An advanced load check performs the following actions:

1. It looks for a matching ALAT entry and if found, falls through to the next instruction.
2. If no matching entry is found, the `chk.a` branches to the specified address.

An implementation may choose to implement a failing advanced load check directly as a branch or as a fault where the fault-handler emulates the branch. Although the expected mode of operation is for an implementation to detect matching entries in the ALAT during checks, an implementation may fail a check instruction even when an entry with a matching ALAT register tag exists. This will be a rare occurrence but software must not assume that the ALAT does not contain the entry.

A check load checks for a matching entry in the ALAT. If no matching entry is found, it reloads the value from memory and any faults that occur during the memory reference are raised. When a matching entry is found, the target register is left unchanged

If the check load was an ordered check load (`ld.c.clr.acq`), then it is performed with the semantics of an ordered load (`ld.acq`). ALAT register tag lookups by advanced load checks and check loads are subject to memory ordering constraints as outlined in "Memory Access Ordering" on page 4-23.

In addition to the flexibility described above, the size, organization, matching algorithm, and replacement algorithm of the ALAT are implementation dependent. Thus, the success or failure of specific advanced loads and checks in a program may change: when the program is run on different processor implementations, within the execution of a single program on the same implementation, or between different runs on the same implementation.

4.4.5.3.2 Invalidating ALAT Entries

In addition to entries removed by advanced loads, ALAT entry invalidations can occur implicitly by events that alter memory state or explicitly by any of the following instructions: `ld.c.clr`, `ld.c.clr.acq`, `chk.a.clr`, `invala`, `invala.e`. Events that may implicitly invalidate ALAT entries include those that change memory state or memory translation state such as:

1. The execution of stores or semaphores on other processors in the coherence domain.
2. The execution of store or semaphore instructions issued on the local processor.

When one of these events occurs, hardware checks each memory region represented by an entry in the ALAT to see if it overlaps with the locations affected by the invalidation event. ALAT entries whose memory regions overlap with the invalidation event locations are removed.

4.4.5.4 Combining Control and Data Speculation

Control speculation and data speculation are not mutually exclusive; a given load may be both control and data speculative. Both control speculative (`ld.sa`, `ldf.sa`, `ldfp.sa`) and non-control speculative (`ld.a`, `ldf.a`, `ldfp.a`) variants of advanced loads are defined for general and floating-point registers. If a speculative advanced load generates a deferred exception token then:

1. Any existing ALAT entry with the same ALAT register tag is invalidated.
2. No new ALAT entry is allocated.
3. If the target of the load was a general-purpose register, its NaT bit is set.
4. If the target of the load was a floating-point register, then NaTVal is written to the target register.

If a speculative advanced load does not generate a deferred exception, then its behavior is the same as the corresponding non-control speculative advanced load.

Since there can be no matching entry in the ALAT after a deferred fault, a single advanced load check or check load is sufficient to check both for data speculation failures and to detect deferred exceptions.

4.4.5.5 Instruction Completers for ALAT Management

To help the compiler manage the allocation and deallocation of ALAT entries, two variants of advanced load checks and check loads are provided: variants with clear (`chk.a.clr`, `ld.c.clr`, `ld.c.clr.acq`, `ldf.c.clr`, `ldfp.c.clr`) and variants with no clear (`chk.a.nc`, `ld.c.nc`, `ldf.c.nc`, `ldfp.c.nc`).

The clear variants are used when the compiler knows that the ALAT entry will not be used again and wants the entry explicitly removed. This allows software to indicate when entries are unneeded, making it less likely that a useful entry will be unnecessarily forced out because all entries are currently allocated.

For the clear variants of check load, any ALAT entry with the same ALAT register tag is invalidated independently of whether the address or size fields of the check load and the corresponding advanced load match. For `chk.a.clr`, the entry is guaranteed to be invalidated only when the instruction falls through (the recovery code is not executed). Thus, a failing `chk.a.clr` may or may not clear any matching ALAT entries. In such cases, the recovery code must explicitly invalidate the entry in question if program correctness depends on the entry being absent after a failed `chk.a.clr`.

Non-clear variants of both kinds of data speculation checks act as a hint to the processor that an existing entry should be maintained in the ALAT or that a new entry should be allocated when a matching ALAT entry doesn't exist. Such variants can be used within loops to check advanced loads which were presumed loop-invariant and moved out of the loop by the compiler. This behavior ensures that if the check load fails on one iteration, then the check load will not necessarily fail on all subsequent iterations. Whenever a new entry is inserted into the ALAT or when the contents of an entry are updated, the information written into the ALAT only uses information from the check load and does not use any residual information from a prior entry. The non-clear variant of `chk.a`, `chk.a.nc`, does not allocate entries and the '`nc`' completer acts as a hint to the processor that the entry should not be cleared.

Table 4-16 and Table 4-17 summarize state and instructions relating to data speculation.

Table 4-16. State Relating to Data Speculation

Structure	Function
ALAT	Advanced load address table

Table 4-17. Instructions Relating to Data Speculation

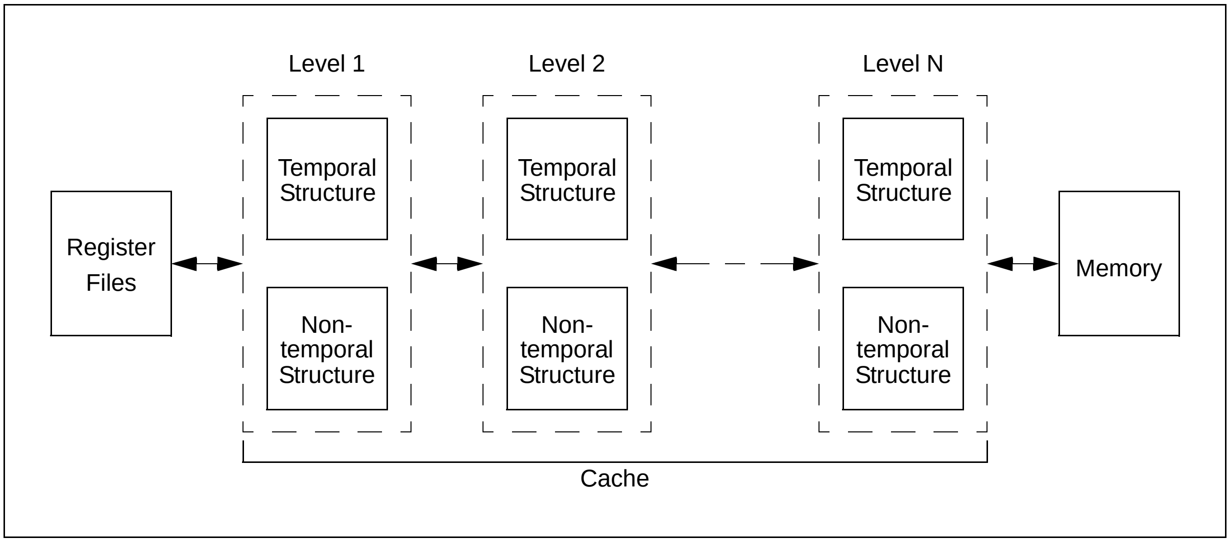
Mnemonic	Operation
`ld.a, ldf.a, ldfp.a`	GR and FR advanced load
`st, st.rel, st8.spill, stf, stf.spill`	GR and FR store
`cmpxchg, fetchadd, xchg`	GR semaphore
`ld.c.clr, ld.c.clr.acq, ldf.c.clr, ldfp.c.clr`	GR and FR check load, clear on ALAT hit
`ld.c.nc, ldf.c.nc, ldfp.c.nc`	GR and FR check load, re-allocate on ALAT miss
`ld.sa, ldf.sa, ldfp.sa`	GR and FR speculative advanced load
`chk.a.clr, chk.a.nc`	GR and FR advanced load check
`invala`	Invalidate all ALAT entries
`invala.e`	Invalidate individual ALAT entry for GR or FR

4.4.6 Memory Hierarchy Control and Consistency

4.4.6.1 Hierarchy Control and Hints

IA-64 memory access instructions are defined to specify whether the data being accessed possesses temporal locality. In addition, memory access instructions can specify which levels of the memory hierarchy are affected by the access. This leads to an architectural view of the memory hierarchy depicted in Figure 4-4 composed of zero or more levels of cache between the register files and memory where each level may consist of two parallel structures: a temporal structure and a non-temporal structure. Note that this view applies to data accesses and not instruction accesses.

Figure 4-4. Memory Hierarchy



The temporal structures cache memory accessed with temporal locality; the non-temporal structures cache memory accessed without temporal locality. Both structures assume that memory accesses possess spatial locality. The existence of separate temporal and non-temporal structures, as well as the number of levels of cache, is implementation dependent.

Three mechanisms are defined for allocation control: locality hints; explicit prefetch; and implicit prefetch. Locality hints are specified by load, store, and explicit prefetch (`lfetch`) instructions. A locality hint specifies a hierarchy level (e.g., 1, 2, all). An access that is temporal with respect to a given hierarchy level is treated as temporal with respect to all lower (higher numbered) levels. An access that is non-temporal with respect to a given hierarchy level is treated as temporal with respect to all lower levels. Finding a cache line closer in the hierarchy than specified in the hint does not demote the line. This enables the precise management of lines using `lfetch` and then subsequent uses by `.nta` loads and stores to retain that level in the hierarchy. For example, specifying the `.nt2` hint by a prefetch indicates that the data should be cached at level 3. Subsequent loads and stores can specify `.nta` and have the data remain at level 3.

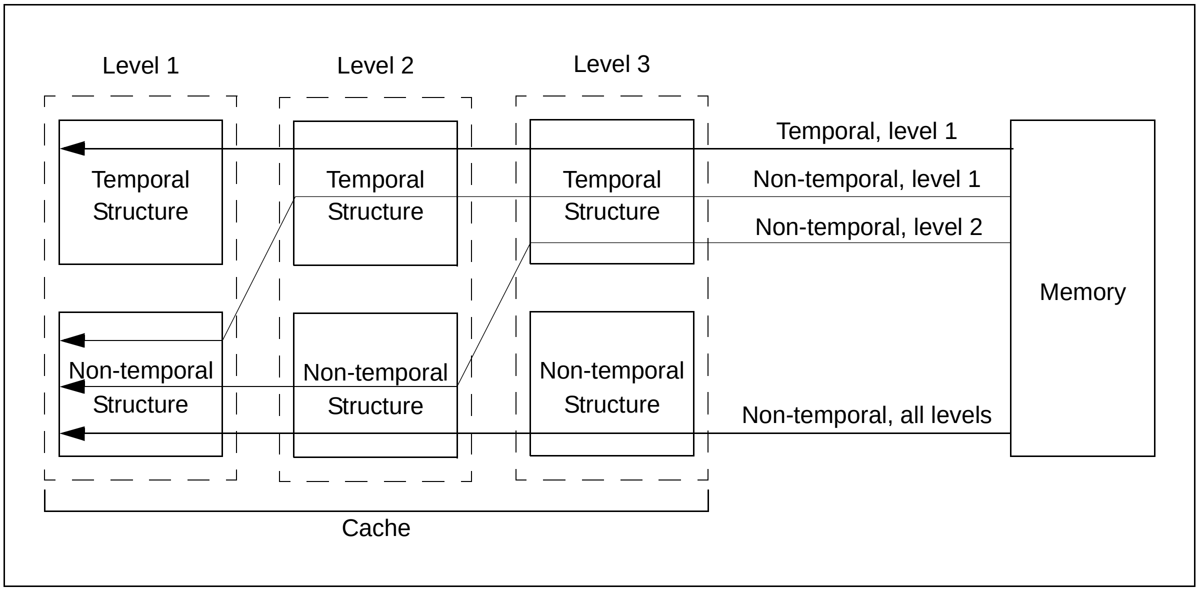
Locality hints do not affect the functional behavior of the program and may be ignored by the implementation. The locality hints available to loads, stores, and explicit prefetch instructions are given in Table 4-18. Instruction accesses are considered to possess both temporal and spatial locality with respect to level 1.

Table 4-18. Locality Hints Specified by Each Instruction Class

Mnemonic	Locality Hint	Instruction Type		
		Load	Store	lfetch, lfetch.fault
`none`	Temporal, level 1	x	x	x
nt1	Non-temporal, level 1	x		x
`nt2`	Non-temporal, level 2			x
`nta`	Non-temporal, all levels	x	x	x

Each locality hint implies a particular allocation path in the memory hierarchy. The allocation paths corresponding to the locality hints are depicted in Figure 4-5. The allocation path specifies the structures in which the line containing the data being referenced would best be allocated. If the line is already at the same or higher level in the hierarchy no movement occurs. Hinting that a datum should be cached in a temporal structure indicates that it is likely to be read in the near future.

Figure 4-5. Allocation Paths Supported in the Memory Hierarchy



Explicit prefetch is defined in the form of the line prefetch instruction (`lfetch`, `lfetch.fault`). The lfetch instructions moves the line containing the addressed byte to a location in the memory hierarchy specified by the locality hint. If the line is already at the same or higher level in the hierarchy, no movement occurs. Both immediate and register post-increment are defined for `lfetch` and `lfetch.fault`. The `lfetch` instruction does not cause any exceptions, does not affect program behavior, and may be ignored by the implementation. The `lfetch.fault` instruction affects the memory hierarchy in exactly the same way as `lfetch` but takes exceptions as if it were a 1-byte load instruction.

Implicit prefetch is based on the address post-increment of loads, stores, `lfetch` and `lfetch.fault`. The line containing the post-incremented address is moved in the memory hierarchy based on the locality hint of the originating load, store, `lfetch` or `lfetch.fault`. If the line is already at the same or higher level in the hierarchy then no movement occurs. Implicit prefetch does not cause any exceptions, does not affect program behavior, and may be ignored by the implementation.

Another form of hint that can be provided on loads is the `ld.bias` load type. This is a hint to the implementation to acquire exclusive ownership of the line containing the addressed data. The bias hint does not affect program functionality and may be ignored by the implementation.

The `fc` instruction invalidates the cache line in all levels of the memory hierarchy above memory. If the cache line is not consistent with memory, then it is copied into memory before invalidation.

Table 4-19 summarizes the memory hierarchy control instructions and hint mechanisms.

Table 4-19. Memory Hierarchy Control Instructions and Hint Mechanisms

Mnemonic	Operation
`.nt1` and `.nta` completer on loads	Load usage hints
`.nta` completer on stores	Store usage hints
prefetch line at post-increment address on loads and stores	Prefetch hint
`lfetch`, `lfetch.fault` with `.nt1`, `.nt2`, and `.nta` hints	Prefetch line
`fc`	Flush cache

4.4.6.2 Memory Consistency

IA-64 instruction accesses made by a processor are not coherent with respect to instruction and/or data accesses made by any other processor, nor are instruction accesses made by a processor coherent with respect to data accesses made by that same processor. Therefore, hardware is not required to keep a processor's instruction caches consistent with respect to any processor's data caches, including that processor's own data caches; nor is hardware required to keep a processor's instruction caches consistent with respect to any other processor's instruction caches. Data accesses from different processors in the same coherence domain are coherent with respect to each other; this consistency is provided by the hardware. Data accesses from the same processor are subject to data dependency rules; see "Memory Access Ordering" below.

The mechanism(s) by which coherence is maintained is implementation dependent. Separate or unified structures for caching data and instructions are not architecturally visible. Within this context there are two categories of data memory hierarchy control: allocation and flush. Allocation refers to movement towards the processor in the hierarchy (lower numbered levels) and flush refers to movement away from the processor in the hierarchy (higher numbered levels). Allocation and flush occur in line-sized units; the minimum architecturally visible line size is 32-bytes (aligned on a 32-byte boundary). The line size in an implementation may be smaller in which case the implementation will need to move multiple lines for each allocation and flush event. An implementation may allocate and flush in units larger than 32-bytes.

In order to guarantee that a write from a given processor becomes visible to the instruction stream of that same, and other, processors, the affected line(s) must be flushed to memory. Software may use the `fc` instruction for this purpose. Memory updates by DMA devices are coherent with respect to instruction and data accesses of processors. The consistency between instruction and data caches of processors with respect to memory updates by DMA devices is provided by the hardware. In case a program modifies its own instructions, the `sync.i` and `srlz.i` instructions are used to ensure that prior coherency actions are observed by a given point in the program. Refer to the description `sync.i` on page 7-177 for an example of self-modifying code.

4.4.7 Memory Access Ordering

Memory data access ordering must satisfy read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) data dependencies to the same memory location. In addition, memory writes and flushes must observe control dependencies. Except for these restrictions, reads, writes, and flushes may occur in an order different from the specified program order. Note that no ordering exists between instruction accesses and data accesses or between any two instruction accesses. The mechanisms described below are defined to enforce a particular memory access order. In the following discussion, the terms "previous" and "subsequent" are used to refer to the program specified order. The term "visible" is used to refer to all architecturally visible effects of performing a memory access (at a minimum this involves reading or writing memory).

Memory accesses follow one of four memory ordering semantics: unordered, release, acquire or fence. Unordered data accesses may become visible in any order. Release data accesses guarantee that all previous data accesses are made visible prior to being made visible themselves. Acquire data accesses guarantee that they are made visible prior to all subsequent data accesses. Fence operations combine the release and acquire semantics into a bi-directional fence, i.e., they guarantee that all previous data accesses are made visible prior to any subsequent data accesses being made visible.

Explicit memory ordering takes the form of a set of instructions: ordered load and ordered check load (`ld.acq`, `ld.c.clr.acq`), ordered store (`st.rel`), semaphores (`cmpxchg`, `xchg`, `fetchadd`), and memory fence (`mf`). The `ld.acq` and `ld.c.clr.acq` instructions follow acquire semantics. The `st.rel` follows release semantics. The `mf` instruction is a fence operation. The `xchg`, `fetchadd.acq`, and `cmpxchg.acq` instructions have acquire semantics. The `cmpxchg.rel`, and `fetchadd.rel` instructions have release semantics. The semaphore instructions also have implicit ordering. If there is a write, it will always follow the read. In addition, the read and write will be performed atomically with no intervening accesses to the same memory region.

Table 4-20 illustrates the ordering interactions between memory accesses with different ordering semantics. “O” indicates that the first and second reference are performed in order with respect to each other. A “-” indicates that no ordering is implied other than data dependencies (and control dependencies for writes and flushes).

Table 4-20. Memory Ordering Rules

First Reference	Second Reference			
	Fence	Acquire	Release	Unordered
fence	O	O	O	O
acquire	O	O	O	O
release	O	–	O	–
unordered	O	–	O	–

Table 4-21 summarizes memory ordering instructions related to cacheable memory.

Table 4-21. Memory Ordering Instructions

Mnemonic	Operation
`ld.acq, ld.c.clr.acq`	Ordered load and ordered check load
`st.rel`	Ordered store
`xchg`	Exchange memory and general register
`cmpxchg.acq, cmpxchg.rel`	Conditional exchange of memory and general register
`fetchadd.acq,fetchadd.rel`	Add immediate to memory
`mf`	Memory ordering fence

4.5 Branch Instructions

Branch instructions effect a transfer of control flow to a new address. Branch targets are bundle-aligned, which means control is always passed to the first instruction slot of the target bundle (slot 0). Branch instructions are not required to be the last instruction in an instruction group. In fact, an instruction group can contain arbitrarily many branches (provided that the normal RAW and WAW dependency requirements are met). If a branch is taken, only instructions up to the taken branch will be executed. After a taken branch, the next instruction executed will be at the target of the branch.

There are two categories of branches: IP-relative branches, and indirect branches. IP-relative branches specify their target with a signed 21-bit displacement, which is added to the IP of the bundle containing the branch to give the address of the target bundle. The displacement allows a branch reach of ±16MBytes and is bundle-aligned. Indirect branches use the branch registers to specify the target address.

There are several branch types, as shown in Table 4-22. The conditional branch `br` is a branch which is taken if the specified predicate is 1, and not-taken otherwise. The conditional call branch `br.call` does the same thing, and in addition, writes a link address to a specified branch register and adjusts the general register stack (see "Register Stack" on page 4-1). The conditional return `br.ret` does the same thing as an indirect conditional branch, plus it adjusts the general register stack. Unconditional branches, calls and returns are executed by specifying PR 0 (which is always 1) as the predicate for the branch instruction.

Table 4-22. Branch Types

Mnemonic	Function	Branch Condition	Target Address
`br.cond` or `br`	Conditional branch	Qualifying predicate	IP-rel or Indirect
`br.call`	Conditional procedure call	Qualifying predicate	IP-rel or Indirect
`br.ret`	Conditional procedure return	Qualifying predicate	Indirect
`br.ia`	Invoke the IA-32 instruction set	Unconditional	Indirect
`br.cloop`	Counted loop branch	Loop count	IP-rel
`br.ctop`, `br.cexit`	Modulo-scheduled counted loop	Loop count and Epilog count	IP-rel
`br.wtop`, `br.wexit`	Modulo-scheduled while loop	Qualifying predicate and Epilog count	IP-rel

The counted loop type (CLOOP) uses the Loop Count (LC) application register. If LC is non-zero then it is decremented and the branch is taken. If LC is zero, the branch falls through. The modulo-scheduled loop type branches (CTOP, CEXIT, WTOP, WEXIT) are described in "Modulo-Scheduled Loop Support" on page 4-26. The loop type branches (CLOOP, CTOP, CEXIT, WTOP, WEXIT) are allowed only in slot 2 of a bundle. A loop type branch executed in slot 0 or 1 will cause an Illegal Operation fault.

Instructions are provided to move data between branch registers and general registers (`mov =br`, `mov br=`). Table 4-23 summarizes state and instructions relating to branching.

Table 4-23. State Relating to Branching

Register	Function
BRs	Branch registers
PRs	Predicate registers
CFM	Current Frame Marker
PFS	Previous Function State application register
LC	Loop Count application register
EC	Epilog Count application register

Table 4-24. Instructions Relating to Branching

Mnemonic	Operation
`br`	Branch
`mov =br`	Move from BR to GR
`mov br=`	Move from GR to BR

4.5.1 Modulo-Scheduled Loop Support

Support for software-pipelined loops is provided through rotating registers and loop branch types. Software pipelining of a loop is analogous to hardware pipelining of a functional unit. The loop body is partitioned into multiple "stages" with zero or more instructions in each stage. Modulo-scheduled loops have 3 phases: prolog, kernel, and epilog. During the prolog phase, new loop iterations are started each time around (filling the software pipeline). During the kernel phase, the pipeline is full. A new loop iteration is started, and another is finished each time around. During the epilog phase, no new iterations are started, but previous iterations are completed (draining the software pipeline).

A predicate is assigned to each stage to control the activation of the instructions in that stage (this predicate is called the "stage predicate"). To support the pipelining effect of stage predicates and registers in a software-pipelined loop, a fixed sized area of the predicate and floating-point register files (PR16-PR63 and FR32-FR127), and a programmable sized area of the general register file, are defined to "rotate." The size of the rotating area in the general register file is determined by an immediate in the `alloc` instruction. This immediate must be either zero or a multiple of 8. The general register rotating area is defined to start at GR32 and overlay the local and output areas, depending on their relative sizes. The stage predicates are allocated in the rotating area of the predicate register file. For counted loops, PR16 is architecturally defined to be the first stage predicate with subsequent stage predicates extending to higher predicate register numbers. For while loops, the first stage predicate may be any rotating predicate with subsequent stage predicates extending to higher predicate register numbers. Software is required to initialize the stage (rotating) predicates prior to entering the loop. An alloc instruction may not change the size of the rotating portion of the register stack frame unless all rotating register bases (rrb's) in the CFM are zero. All rrb's can be set to zero with the `clrrrb` instruction. The `clrrrb.pr` form can be used to clear just the rrb for the predicate registers. The `clrrrb` instruction must be the last instruction in an instruction group.

Rotation by one register position occurs when a software-pipelined loop type branch is executed. Registers are rotated towards larger register numbers in a wrap-around fashion. For example, the value in register X will be located in register X+1 after one rotation. If X is the highest addressed rotating register its value will wrap to the lowest addressed rotating register. Rotation is implemented by renaming register numbers based upon the value of a rotating register base (rrb) contained in CFM. A rrb is defined for each of the three rotating register files: CFM.rrb.gr for the general registers; CFM.rrb.fr for the floating-point registers; CFM.rrb.pr for the predicate registers. General registers only rotate when the size of the rotating region is not equal to zero. Floating-point and predicate registers always rotate. When rotation occurs, two or all three rrb's are decremented in unison. Each rrb is decremented modulo the size of their respective rotating regions (e.g., 96 for rrb.fr). The operation of the rotating register rename mechanism is not otherwise visible to software. The instructions that modify the rrb's are listed in Table 4-25.

Table 4-25. Instructions that Modify RRBs

Mnemonic	Operation
`clrrrb`	Clears all rrb's
`clrrrb.pr`	Clears rrb.pr
`br.call`	Clears all rrb's
`br.ret`	Restores CFM.rrb's from PFM.rrb's
`br.ctop, br.cexit, br.wtop, and br.wexit`	Decrements all rrb's

There are two categories of software-pipelined loop branch types: counted and while. Both categories have two forms: top and exit. The "top" variant is used when the loop decision is located at the bottom of the loop body. A taken branch will continue the loop while a not-taken branch will exit the loop. The "exit" variant is used when the loop decision is located somewhere other than the bottom of the loop. A not-taken branch will continue the loop and a taken branch will exit the loop. The "exit" variant is also used at intermediate points in an unrolled pipelined loop.

The branch condition of a counted loop branch is determined by the specific counted loop type (ctop or cexit), the value of the loop count application register (LC), and the value of the epilog count application register (EC). Note that the counted loop branches do not use a qualifying predicate. LC is initialized to one less than the number of iterations for the counted loop and EC is initialized to the number of stages into which the loop body has been partitioned. While LC is greater than zero, the branch direction will continue the loop, LC will be decremented, registers will be rotated (rrb's are decremented), and PR 16 will be set to 1 after rotation. (For each of the loop-type branches, PR 63 is written by the branch, and after rotation this value will be in PR 16.)

Execution of a counted loop branch with LC equal to zero signals the start of the epilog. While in the epilog and while EC is greater than one, the branch direction will continue the loop, EC will be decremented, registers will be rotated, and PR 16 will be set to 0 after rotation. Execution of a counted loop branch with LC equal to zero and EC equal to one signals the end of the loop; the branch direction will exit the loop, EC will be decremented, registers will be rotated, and PR 16 will be set to 0 after rotation. A counted loop type branch executed with both LC and EC equal to zero will have a branch direction to exit the loop. LC, EC, and the rrb's will not be modified (no rotation) and PR 63 will be set to 0. LC and EC equal to zero can occur in some types of optimized, unrolled software-pipelined loops if the target of a cexit branch is set to the next sequential bundle and the loop trip count is not evenly divisible by the unroll amount.

The direction of a while loop branch is determined by the specific while loop type (wtop or wexit), the value of the qualifying predicate, and the value of EC. The while loop branches do not use LC. While the qualifying predicate is one, the branch direction will continue the loop, registers will be rotated, and PR 16 will be set to 0 after rotation. While the qualifying predicate is zero and EC is greater than one, the branch direction will continue the loop, EC will be decremented, registers will be rotated, and PR 16 will be set to 0 after rotation. The qualifying predicate is one during the kernel and zero during the epilog. During the prolog, the qualifying predicate may be zero or one depending upon the scheme used to program the pipelined while loop. Execution of a while loop branch with qualifying predicate equal to zero and EC equal to one signals the end of the loop; the branch direction will exit the loop, EC will be decremented, registers will be rotated, and PR 16 will be set to 0 after rotation. A while loop branch executed with a zero qualifying predicate and with EC equal to zero has a branch direction to exit the loop. EC and the rrb's will not be modified (no rotation) and PR 63 will be set to 0.

For while loops, the initialization of EC depends upon the scheme used to program the pipelined while loop. Often, the first valid condition for the while loop branch is not computed until several stages into the prolog. Therefore, software pipelines for while loops often have several speculative prolog stages. During these stages, the qualifying predicate can be set to zero or one depending upon the scheme used to program the loop. If the qualifying predicate is one throughout the prolog, EC will be decremented only during the epilog phase and is initialized to one more than the number of epilog stages. If the qualifying predicate is zero during the speculative stages of the prolog, EC will be decremented during this part of the prolog, and the initialization value for EC is increased accordingly.

4.5.2 Branch Prediction Hints

Information about branch behavior can be provided to the processor to improve branch prediction. This information can be encoded with branch hints as part of a branch instruction (referred to as hints). Hints do not affect the functional behavior of the program and may be ignored by the processor.

Branch instructions can provide three types of hints:

- **Whether prediction strategy:** This describes (for COND, CALL and RET type branches) how the processor should predict the branch condition. (For the loop type branches, prediction is based on LC and EC.) The suggested strategies that can be hinted are shown in Table 4-26.

Table 4-26. Whether Prediction Hint on Branches

Completer	Strategy	Operation
`spnt`	Static Not-Taken	Ignore this branch, do not allocate prediction resources for this branch.
`sptk`	Static Taken	Always predict taken, do not allocate prediction resources for this branch.
`dpnt`	Dynamic Not-Taken	Use dynamic prediction hardware. If no dynamic history information exists for this branch, predict not-taken.
`dptk`	Dynamic Taken	Use dynamic prediction hardware. If no dynamic history information exists for this branch, predict taken.

- **Sequential prefetch:** This indicates how much code the processor should prefetch at the branch target (shown in Table 4-27).

Table 4-27. Sequential Prefetch Hint on Branches

Completer	Sequential Prefetch Hint	Operation
`few`	Prefetch few lines	When prefetching code at the branch target, stop prefetching after a few (implementation-dependent number of) lines.
`many`	Prefetch many lines	When prefetching code at the branch target, prefetch more lines (also an implementation-dependent number).

- **Predictor deallocation:** This provides re-use information to allow the hardware to better manage branch prediction resources. Normally, prediction resources keep track of the most-recently executed branches. However, sometimes the most-recently executed branch is not useful to remember, either because it will not be re-visited any time soon or because a hint instruction will re-supply the information prior to re-visiting the branch. In such cases, this hint can be used to free up the prediction resources.

Table 4-28. Predictor Deallocation Hint

Completer	Operation
`none`	Don't deallocate
`clr`	Deallocate branch information

4.6 Multimedia Instructions

Multimedia instructions (see Table 4-29) treat the general registers as concatenations of eight 8-bit, four 16-bit, or two 32-bit elements. They operate on each element independently and in parallel. The elements are always aligned on their natural boundaries within a general register. Most multimedia instructions are defined to operate on multiple element sizes. Three classes of multimedia instructions are defined: arithmetic, shift and data arrangement.

4.6.1 Parallel Arithmetic

There are three forms of parallel addition and subtraction: modulo (`padd`, `psub`), signed saturation (`padd.sss`, `psub.sss`), and unsigned saturation (`padd.uuu`, `padd.uus`, `psub.uuu`, `psub.uus`). The modulo forms have the result wrap around the largest or smallest representable value in the range of the result element. In the saturating forms, results larger than the largest representable value of the range of the result element, or smaller than the smallest representable value of the range, are clamped to the largest or smallest value in the range of the result element respectively. The signed saturation form treats both sources as signed and clamps the result to the limits of a signed range. The unsigned saturation form treats one source as unsigned and clamps the result to the limits of an unsigned range. Two variants are defined that treat the second source as either signed (`.uus`) or unsigned (`.uuu`).

The parallel average instruction (`pavg, pavg.raz`) adds corresponding elements from each source and right shifts each result by one bit. In the simple form of the instruction, the carry out of the most-significant bit of each sum is written into the most significant bit of the result element. In the round-away-from-zero form, a 1 is added to each sum before shifting. The parallel average subtract instruction (`pavgsub`) performs a similar operation on the difference of the sources.

The parallel shift left and add instruction (`pshladd`) performs a left shift on the elements of the first source and then adds them to the corresponding elements from the second source. Signed saturation is performed on both the shift and the add operations. The parallel shift right and add instruction (`pshradd`) is similar to `pshladd`. Both of these instructions are defined for 2-byte elements only.

The parallel compare instruction (`pcmp`) compares the corresponding elements of both sources and writes all ones (if true) or all zeroes (if false) into the corresponding elements of the target according to one of two relations (== or >).

The parallel multiply right instruction (`pmpy.r`) multiplies the corresponding two even-numbered signed 2-byte elements of both sources and writes the results into two 4-byte elements in the target. The `pmpy.l` instruction performs a similar operation on odd-numbered 2-byte elements. The parallel multiply and shift right instruction (`pmpyshr`, `pmpyshr.u`) multiplies the corresponding 2-byte elements of both sources producing four 4-byte results. The 4-byte results are shifted right by 0, 7, 15, or 16 bits as specified by the instruction. The least-significant 2 bytes of the 4-byte shifted results are then stored in the target register.

The parallel sum of absolute difference instruction (psad) accumulates the absolute difference of corresponding 1-byte elements and writes the result in the target.

The parallel minimum (pmin.u, pmin) and the parallel maximum (pmax.u, pmax) instructions deliver the minimum or maximum, respectively, of the corresponding 1-byte or 2-byte elements in the target. The 1-byte elements are treated as unsigned values and the 2-byte elements are treated as signed values.

Table 4-29. Parallel Arithmetic Instructions

Mnemonic	Operation	1-byte	2-byte	4-byte
padd	Parallel modulo addition	x	x	x
padd.sss	Parallel addition with signed saturation	x	x	
padd.uuu, padd.uus	Parallel addition with unsigned saturation	x	x	
psub	Parallel modulo subtraction	x	x	x
psub.sss	Parallel subtraction with signed saturation	x	x	
psub.uuu, psub.uus	Parallel subtraction with unsigned saturation	x	x	
pavg	Parallel arithmetic average	x	x	
pavg.raz	Parallel arithmetic average with round away from zero	x	x	
pavgsub	Parallel average of a difference	x	x	
pshladd	Parallel shift left and add with signed saturation		x	
pshradd	Parallel shift right and add with signed saturation		x	
pcmp	Parallel compare	x	x	x
pmpy.l	Parallel signed multiply of odd elements			x
pmpy.r	Parallel signed multiply of even elements			x
pmpyshr	Parallel signed multiply and shift right		x	
pmpyshr.u	Parallel unsigned multiply and shift right		x	
psad	Parallel sum of absolute difference	x		
pmin	Parallel minimum	x	x	
pmax	Parallel maximum	x	x	

4.6.2 Parallel Shifts

The parallel shift left instruction (pshl) individually shifts each element of the first source by a count contained in either a general register or an immediate. The parallel shift right instruction (pshr) performs an individual arithmetic right shift of each element of one source by a count contained in either a general register or an immediate. The pshr.u instruction performs an unsigned right shift. Table 4-30 summarizes the types of parallel shift instructions.

Table 4-30. Parallel Shift Instructions

Mnemonic	Operation	1-byte	2-byte	4-byte
pshl	Parallel shift left		x	x
pshr	Parallel signed shift right		x	x
pshr.u	Parallel unsigned shift right		x	x

4.6.3 Data Arrangement

The mix right instruction (mix.r) interleaves the even-numbered elements from both sources into the target. The mix left instruction (mix.l) interleaves the odd-numbered elements. The unpack low instruction (unpack.l) interleaves the elements in the least-significant 4 bytes of each source into the target register. The unpack high instruction (unpack.h) interleaves elements from the most significant 4 bytes. The pack instructions (pack.sss, pack.uss) convert from 32-bit or 16-bit elements to 16-bit or 8-bit elements respectively. The least-significant half of larger elements in both sources are extracted and written into smaller elements in the target register. The pack.sss instruction treats the extracted elements as signed values and performs signed saturation on them. The pack.uss instruction performs unsigned saturation. The mux instruction (mux) copies individual 2-byte or 1-byte elements in the source to arbitrary positions in the target according to a specified function. For 2-byte elements, an 8-bit immediate allows all possible permutations to be specified. For 1-byte elements the copy function is selected from one of five possibilities (reverse, mix, shuffle, alternate, broadcast). Table 4-31 describes the various types of parallel data arrangement instructions.

Table 4-31. Parallel Data Arrangement Instructions

Mnemonic	Operation	1-byte	2-byte	4-byte
mix.l	Interleave odd elements from both sources	x	x	x
mix.r	Interleave even elements from both sources	x	x	x
mux	Arbitrary copy of individual source elements	x	x	
pack.sss	Convert from larger to smaller elements with signed saturation		x	x
pack.uss	Convert from larger to smaller elements with unsigned saturation		x	
unpack.l	Interleave least-significant elements from both sources	x	x	x
unpack.h	Interleave most significant elements from both sources	x	x	x

4.7 Register File Transfers

Table 4-32 shows the instructions defined to move values between the general register file and the floating-point, branch, predicate, performance monitor, processor identification, and application register files. Several of the transfer instructions share the same mnemonic (mov). The value of the operand identifies which register file is accessed.

Table 4-32. Register File Transfer Instructions

Mnemonic	Operation
`getf.exp, getf.sig`	Move FR exponent or significand to GR
`getf.s, getf.d`	Move single/double precision memory format from FR to GR
`setf.s, setf.d`	Move single/double precision memory format from GR to FR
`setf.exp, setf.sig`	Move from GR to FR exponent or significand
`mov =br`	Move from BR to GR
`mov br=`	Move from GR to BR
`mov =pr`	Move from predicates to GR
`mov pr=, mov pr.rot=`	Move from GR to predicates
`mov ar=`	Move from GR to AR
`mov =ar`	Move from AR to GR
`sum, rum`	Set and reset user mask
`mov =pmd[...]`	Move from performance monitor data register to GR
`mov =cpuid[...]`	Move from processor identification register to GR
`mov =ip`	Move from Instruction Pointer

Memory access instructions only target or source the general and floating-point register files. It is necessary to use the general register file as an intermediary for transfers between memory and all other register files except the floating-point register file.

Two classes of move are defined between the general registers and the floating-point registers. The first type moves the significand or the sign/exponent (`getf.sig`, `setf.sig`, `getf.exp`, `setf.exp`). The second type moves entire single or double precision numbers (`getf.s`, `setf.s`, `getf.d`, `setf.d`). These instructions also perform a conversion between the deferred exception token formats.

Instructions are provided to transfer between the branch registers and the general registers.

Instructions are defined to transfer between the predicate register file and a general register. These instructions operate in a “broadside” manner whereby multiple predicate registers are transferred in parallel (predicate register N is transferred to and from bit N of a general register). The move to predicate instruction (`mov pr=`) transfers a general register to multiple predicate registers according to a mask specified by an immediate. The mask contains one bit for each of the static predicate registers (PR 1 through PR 15 – PR 0 is hardwired to 1) and one bit for all of the rotating predicates (PR 16 through PR63). A predicate register is written from the corresponding bit in a general register if the corresponding mask bit is set. If the mask bit is clear then the predicate register is not modified. The rotating predicates are transferred as if CFM.rrb.pr were zero. The actual value in CFM.rrb.pr is ignored and remains unchanged. The move from predicate instruction (`mov =pr`) transfers the entire predicate register file into a general register target.

The `mov =pmd[]` instruction is defined to move from a performance monitor data (PMD) register to a general register. If the operating system has not enabled reading of performance monitor data registers in user level then all zeroes are returned. The `mov =cpuid[]` instruction is defined to move from a processor identification register to a general register.

The `mov =ip` instruction is provided for copying the current value of the instruction pointer (IP) into a general register.

4.8 Character Strings and Population Count

A small set of special instructions accelerate operations on character and bit-field data.

4.8.1 Character Strings

The compute zero index instructions (`czx.l`, `czx.r`) treat the general register source as either eight 1-byte or four 2-byte elements and write the general register target with the index of the first zero element found. If there are no zero elements in the source, the target is written with a constant one higher than the largest possible index (8 for the 1-byte form, 4 for the 2-byte form). The `czx.l` instruction scans the source from left to right with the left-most element having an index of zero. The `czx.r` instruction scans from right to left with the right-most element having an index of zero. Table 4-33 summarizes the compute zero index instructions.

Table 4-33. String Support Instructions

Mnemonic	Operation	1-byte	2-byte
`czx.l`	Locate first zero element, left to right	x	x
`czx.r`	Locate first zero element, right to left	x	x

4.8.2 Population Count

The population count instruction (`popcnt`) writes the number of bits which have a value of 1 in the source register into the target register.

IA-64 Floating-point Programming Model 5

The IA-64 floating-point architecture is fully compliant with the ANSI/IEEE Standard for Binary Floating-point Arithmetic (Std. 754-1985). There is full IEEE support for single, double, and double-extended real formats. The two IEEE methods for controlling rounding precision are supported. The first method converts results to the double-extended exponent range. The second method converts results to the destination precision. Some IEEE extensions such as fused multiply and add, minimum and maximum operations, and a register file format with a larger range than the minimum double-extended format are also included.

5.1 Data Types and Formats

Six data types are supported directly: single, double, double-extended real (IEEE real types); 64-bit signed integer, 64-bit unsigned integer, and the 82-bit floating-point register format. A "Parallel FP" format where a pair of IEEE single precision values occupy a floating-point register's significand is also supported. A seventh data type, IEEE-style quad-precision, is supported by software routines. A future architecture extension may include additional support for the quad-precision real type.

5.1.1 Real Types

The parameters for the supported IEEE real types are summarized in Table 5-1.

Table 5-1. IEEE Real-Type Properties

	Single	Double	Double-Extended	Quad-Precision
IEEE Real-Type Parameters				
Sign	+ or –	+ or –	+ or –	+ or –
E_{max}	+127	+1023	+16383	+16383
E_{min}	–126	–1022	–16382	–16382
Exponent bias	+127	+1023	+16383	+16383
Precision (bits)	24	53	64	113
IEEE Memory Formats				
Total memory format width (bits)	32	64	80	128
Sign field width (bits)	1	1	1	1
Exponent field width (bits)	8	11	15	15
Significand field width (bits)	23	52	64	112

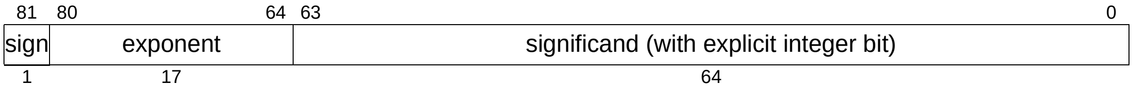
5.1.2 Floating-point Register Format

Data contained in the floating-point registers can be either integer or real type. The format of data in the floating-point registers is designed to accommodate both of these types with no loss of information.

Real numbers reside in 82-bit floating-point registers in a three-field binary format (see Figure 5-1). The three fields are:

- The 64-bit **significand** field, b_{63}**.** $b_{62}b_{61}$ .. b_1b_0, contains the number's significant digits. This field is composed of an explicit integer bit (significand{63}), and 63 bits of fraction (significand{62:0}). For Parallel FP data, the significand field holds a pair of 32-bit IEEE single real numbers.
- The 17-bit **exponent** field locates the binary point within or beyond the significant digits (i.e., it determines the number's magnitude). The exponent field is biased by 65535 (0xFFFF). An exponent field of all ones is used to encode the special values for IEEE signed infinity and NaNs. An exponent field of all zeros and a significand field of all zeros is used to encode the special values for IEEE signed zeros. An exponent field of all zeros and a non-zero significand field encodes the double-extended real denormals and double-extended real pseudo-denormals.
- The 1-bit **sign** field indicates whether the number is positive (sign=0) or negative (sign=1). For Parallel FP data, this bit is always 0.

Figure 5-1. Floating-point Register Format



The value of a finite floating-point number, encoded with non-zero exponent field, can be calculated using the expression:

$$(-1)^{(\text{sign})} * 2^{(\text{exponent} - 65535)} * (\text{significand}\{63\}.\text{significand}\{62{:}0\}_2)$$

The value of a finite floating-point number, encoded with zero exponent field, can be calculated using the expression:

$$(-1)^{(\text{sign})} * 2^{(-16382)} * (\text{significand}\{63\}.\text{significand}\{62{:}0\}_2)$$

Integers (64-bit signed/unsigned) and Parallel FP numbers reside in the 64-bit significand field. In their canonical form, the exponent field is set to 0x1003E (biased 63) and the sign field is set to 0.

5.1.3 Representation of Values in Floating-point Registers

The floating-point register encodings are grouped into classes and subclasses and listed below in Table 5-2 (shaded encodings are unsupported). The last two table entries contain the values of the constant floating-point registers, FR 0 and FR 1. The constant value in FR 1 does not change for the parallel single precision instructions or for the integer multiply accumulate instruction and would not generally be useful.

Table 5-2. Floating-point Register Encodings

Class or Subclass	Sign (1 bit)	Biased Exponent (17-bits)	Significand i.bb...bb (explicit integer bit is shown) (64-bits)
NaNs	0/1	0x1FFFF	1.000...01 through 1.111...11
Quiet NaNs	0/1	0x1FFFF	1.100...00 through 1.111...11
Quiet NaN Indefinite[a]	1	0x1FFFF	1.100...00
Signaling NaNs	0/1	0x1FFFF	1.000...01 through 1.011...11
Infinity	0/1	0x1FFFF	1.000...00
Pseudo-NaNs	0/1	0x1FFFF	0.000...01 through 0.111...11
Pseudo-Infinity	0/1	0x1FFFF	0.000...00
Normalized Numbers (Floating-point Register Format Normals)	0/1	0x00001 through 0x1FFFE	1.000...00 through 1.111...11
Integers or Parallel FP (large unsigned or negative signed integers)	0	0x1003E	1.000...00 through 1.111...11
Integer Indefinite[b]	0	0x1003E	1.000...00
IEEE Single Real Normals	0/1	0x0FF81 through 0x1007E	1.000...00...(40)0s through 1.111...11...(40)0s
IEEE Double Real Normals	0/1	0x0FC01 through 0x103FE	1.000...00...(11)0s through 1.111...11...(11)0s
IEEE Double-Extended Real Normals	0/1	0x0C001 through 0x13FFE	1.000...00 through 1.111...11
Normal numbers with the same value as Double-Extended Real Pseudo-Denormals	0/1	0x0C001	1.000...00 through 1.111...11
IA-32 Stack Single Real Normals (produced when the computation model is IA-32 Stack Single)	0/1	0x0C001 through 0x13FFE	1.000...00...(40)0s through 1.111...11...(40)0s
IA-32 Stack Double Real Normals (produced when the computation model is IA-32 Stack Double)	0/1	0x0C001 through 0x13FFE	1.000...00...(11)0s through 1.111...11...(11)0s
Unnormalized Numbers (Floating-point Register Format unnormalized numbers)	0/1	0x00000	0.000...01 through 1.111...11
		0x00001 through 0x1FFFE	0.000...01 through 0.111...11
		0x00001 through 0x1FFFD	0.000...00
	1	0x1FFFE	0.000...00
Integers or Parallel FP (positive signed/unsigned integers)	0	0x1003E	0.000...00 through 0.111...11
Single Real Denormals	0/1	0x0FF81	0.000...01...(40)0s through 0.111...11...(40)0s

Table 5-2. Floating-point Register Encodings (Cont’d)

Class or Subclass	Sign (1 bit)	Biased Exponent (17-bits)	Significand i.bb...bb (explicit integer bit is shown) (64-bits)
Double Real Denormals	0/1	0x0FC01	0.000...01...(11)0s through 0.111...11...(11)0s
Register Format Denormals	0/1	0x00001	0.000...01 through 0.111...11
Double-Extended Real Denormals	0/1	0x00000	0.000...01 through 0.111...11
Unnormal numbers with the same value as Double-Extended Real Denormals	0/1	0x0C001	0.000...01 through 0.111...11
Double-Extended Real Pseudo-Denormals (IA-32 stack and memory format)	0/1	0x00000	1.000...00 through 1.111...11
IA-32 Stack Single Real Denormals (produced when computation model is IA-32 Stack Single)	0/1	0x00000	0.000...01...(40)0s through 0.111...11...(40)0s
IA-32 Stack Double Real Denormals (produced when computation model is IA-32 Stack Double)	0/1	0x00000	0.000...01...(11)0s through 0.111...11...(11)0s
Pseudo-Zeros	0/1	0x00001 through 0x1FFFD	0.000...00
	1	0x1FFFE	0.000...00
NaTVal[c]	0	0x1FFFE	0.000...00
Zero	0/1	0x00000	0.000...00
FR 0 (positive zero)	0	0x00000	0.000...00
FR 1 (positive one)	0	0x0FFFF	1.000...00

a. Default response on a masked real invalid operation.
b. Default response on a masked integer invalid operation.
c. Created by unsuccessful speculative memory operation.

All register file encodings are allowed as inputs to arithmetic operations. The result of an arithmetic operation is always the most normalized register file representation of the computed value, with the exponent range limited from Emin to Emax of the destination type, and the significand precision limited to the number of precision bits of the destination type. Computed values, such as zeros, infinities, and NaNs that are outside these bounds are represented by the corresponding unique register file encoding. Double-extended real denormal results are mapped to the register file exponent of 0x00000 (instead of 0x0C001). Unsupported encodings (Pseudo-NaNs and Pseudo-Infinities), Pseudo-zeros and Double-extended Real Pseudo-denormals are never produced as a result of an arithmetic operation.

Arithmetic on pseudo-zeros operates exactly as an equivalently signed zero, with one exception. Pseudo-zero multiplied by infinity returns the correctly signed infinity instead of an Invalid Operation Floating-point Exception fault (and QNaN). Also, pseudo-zeros are classified as unnormalized numbers, not zeros.

5.2 Floating-point Status Register

The Floating-point Status Register (FPSR) contains the dynamic control and status information for floating-point operations. There is one main set of control and status information (FPSR.sf0), and three alternate sets (FPSR.sf1, FPSR.sf2, FPSR.sf3). The FPSR layout is shown in Figure 5-2 and its fields are defined in Table 5-3. Table 5-4 gives the FPSR's status field description and Figure 5-3 shows their layout.

Figure 5-2. Floating-point Status Register Format

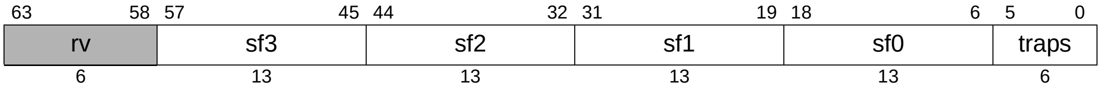


Table 5-3. Floating-point Status Register Field Description

Field	Bits	Description
traps.vd	0	Invalid Operation Floating-point Exception fault (IEEE Trap) disabled when this bit is set
traps.dd	1	Denormal/Unnormal Operand Floating-point Exception fault disabled when this bit is set
traps.zd	2	Zero Divide Floating-point Exception fault (IEEE Trap) disabled when this bit is set
traps.od	3	Overflow Floating-point Exception trap (IEEE Trap) disabled when this bit is set
traps.ud	4	Underflow Floating-point Exception trap (IEEE Trap) disabled when this bit is set
traps.id	5	Inexact Floating-point Exception trap (IEEE Trap) disabled when this bit is set
sf0	18:6	Main status field
sf1	31:19	Alternate status field 1
sf2	44:32	Alternate status field 2
sf3	57:45	Alternate status field 3
rv	63:58	Reserved

Figure 5-3. Floating-point Status Field Format

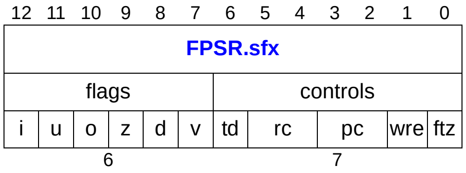


Table 5-4. Floating-point Status Register's Status Field Description

Field	Bits	Description
ftz	0	Flush-to-Zero mode
wre	1	Widest range exponent (see Table 5-6)
pc	3:2	Precision control (see Table 5-6)
rc	5:4	Rounding control (see Table 5-5)

Table 5-4. Floating-point Status Register's Status Field Description (Cont'd)

Field	Bits	Description
td	6	Traps disabled[a]
v	7	Invalid Operation (IEEE Flag)
d	8	Denormal/Unnormal Operand
z	9	Zero Divide (IEEE Flag)
o	10	Overflow (IEEE Flag)
u	11	Underflow (IEEE Flag)
i	12	Inexact (IEEE Flag)

[a] td is a reserved bit in the main status field, FPSR.sf0.

The Denormal/Unnormal Operand status flag is an IEEE-style sticky flag that is set if the value is used in an arithmetic instruction and in an arithmetic calculation; e.g. unorm*NaN doesn't set the d flag. Canonical single/double/double-extended denormal/double-extended pseudo-denormal/register format denormal encodings are a subset of the floating-point register format unnormalized numbers.

Note: The Floating-point Exception fault/trap occurs only if an enabled floating-point exception occurs during the processing of the instruction. Hence, setting a flag bit of a status field to 1 in software will not cause an interruption. The status fields flags are merely indications of the occurrence of floating-point exceptions.

Flush-to-Zero (FTZ) mode causes results which encounter "tininess" to be truncated to the correctly signed zero. Flush-to-Zero mode can be enabled only if Underflow is disabled. This can be accomplished by disabling all traps (FPSR.sf*x*.td being set to 1), or by disabling it individually (FPSR.traps.ud set to 1). If Underflow is enabled then it takes priority and Flush-to-Zero mode is ignored. Note that the software exception handler could examine the Flush-to Zero mode bit and choose to emulate the Flush-to-Zero operation when an enabled Underflow exception arises.

The FPSR.sf*x*.u and FPSR.sf*x*.i bits will be set to 1 when a result is flushed to the correctly signed zero because of Flush-to-Zero mode. If enabled, an inexact result exception is signaled.

A floating-point result is rounded based on the instruction's *.pc* completer and the status field's *wre*, *pc*, and *rc* control fields. The result's significand precision and exponent range are determined as described in Table 5-6. If the result isn't exact, FPSR.sf*x*.*rc* specifies the rounding direction (see Table 5-5).

Table 5-5. Floating-point Rounding Control Definitions

	Nearest (or even)	– Infinity (down)	+ Infinity (up)	Zero (truncate/chop)
FPSR.sf*x*.rc	00	01	10	11

Table 5-6. Floating-point Computation Model Control Definitions

Computation Model Control Fields			Computation Model Selected		
Instruction's *.pc* Completer	FPSR.sfx's Dynamic *pc* Field	FPSR.sfx's Dynamic *wre* Field	Significand Precision	Exponent Range	Computational Style
.s	ignored	0	24 bits	8 bits	IEEE real single
.d	ignored	0	53 bits	11 bits	IEEE real double
.s	ignored	1	24 bits	17 bits	Register file range, single precision
.d	ignored	1	53 bits	17 bits	Register file range, double precision
none	00	0	24 bits	15 bits	IA-32 stack single
none	01	0	N.A.	N.A.	Reserved
none	10	0	53 bits	15 bits	IA-32 stack double
none	11	0	64 bits	15 bits	IA-32 double-extended
none	00	1	24 bits	17 bits	Register file range, single precision
none	01	1	N.A.	N.A.	Reserved
none	10	1	53 bits	17 bits	Register file range, double precision
none	11	1	64 bits	17 bits	Register file range, double-extended precision
not applicable[a]	ignored	ignored	24 bits	8 bits	A pair of IEEE real singles
not applicable[b]	ignored	ignored	64 bits	17 bits	Register file range, double-extended precision

a. For parallel FP instructions which have no *.pc* completer (e.g., fpma).
b. For non-parallel FP instructions which have no *.pc* completer (e.g., fmerge).

The trap disable (sf*x*.td) control bit allows one to easily set up a local IEEE exception trap default environment. If FPSR.sf*x*.td is clear (enabled), the FPSR.traps bits are used. If FPSR.sf*x*.td is set, the FPSR.traps bits are treated as if they are all set (disabled). Note that FPSR.sf0.td is a reserved field which returns 0 when read.

5.3 Floating-point Instructions

This section describes the IA-64 floating-point instructions.

5.3.1 Memory Access Instructions

There are floating-point load and store instructions for the single, double, double-extended floating-point real data types, and the Parallel FP or signed/unsigned integer data type. The addressing modes for floating-point load and store instructions are the same as for integer load and store instructions, except for floating-point load pair instructions which can have an implicit base-register post increment. The memory hint options for floating-point load and store instructions are the same as those for integer load and store instructions. (See "Memory Hierarchy Control and Consistency" on page 4-20.) Table 5-7 lists the types of floating-point load and store instructions. The floating-point load pair instructions require the two target registers to be odd/even or even/odd. The floating-point store instructions (stfs, stfd, stfe) require the value in the floating-point register to have the same type as the store for the format conversion to be correct.

Table 5-7. Floating-point Memory Access Instructions

Operations	Load to FR	Load Pair to FR	Store from FR
Single	ldfs	ldfps	stfs
Integer/Parallel FP	ldf8	ldfp8	stf8
Double	ldfd	ldfpd	stfd
Double-extended	ldfe		stfe
Spill/fill	ldf.fill		stf.spill

Unsuccessful speculative loads write a NaTVal into the destination register or registers (see Section 4.4.4). Storing a NaTVal to memory will cause a Register NaT Consumption fault, except for the spill instruction (stf.spill).

Saving and restoring floating-point registers is accomplished by the spill and fill instructions (stf.spill, ldf.fill) using a 16-byte memory container. These are the only instructions that can be used for saving and restoring the actual register contents since they do not fault on NaTVal. They save and restore all types (single, double, double-extended, register format and integer or Parallel FP) and will ensure compatibility with possible future architecture extensions.

Figure 5-4, Figure 5-5, Figure 5-6 and Figure 5-7 describe how single precision, double precision, double-extended precision, and spill/fill data is translated during transfers between floating-point registers and memory.

Figure 5-4. Memory to Floating-point Register Data Translation – Single Precision

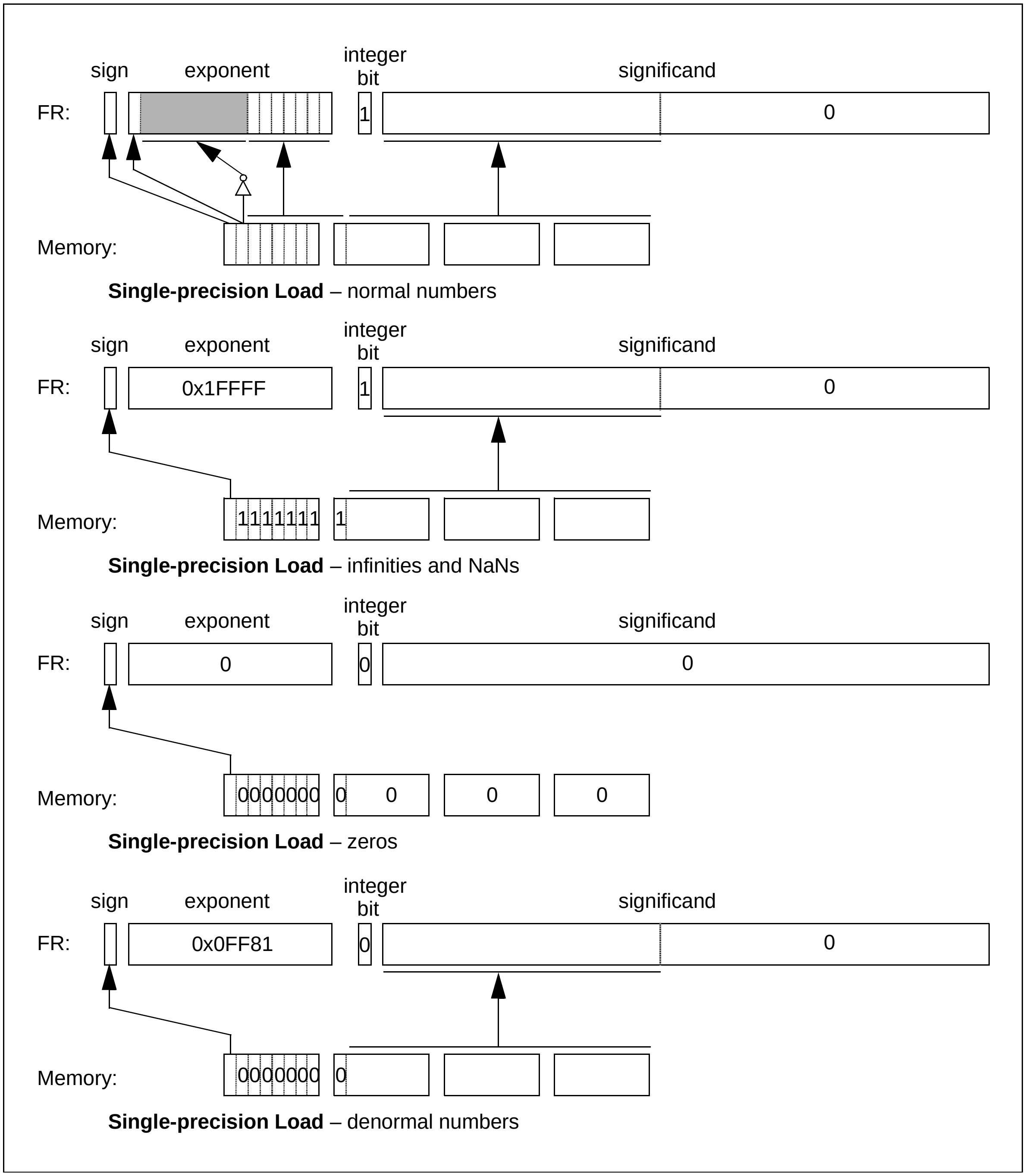


Figure 5-5. Memory to Floating-point Register Data Translation – Double Precision

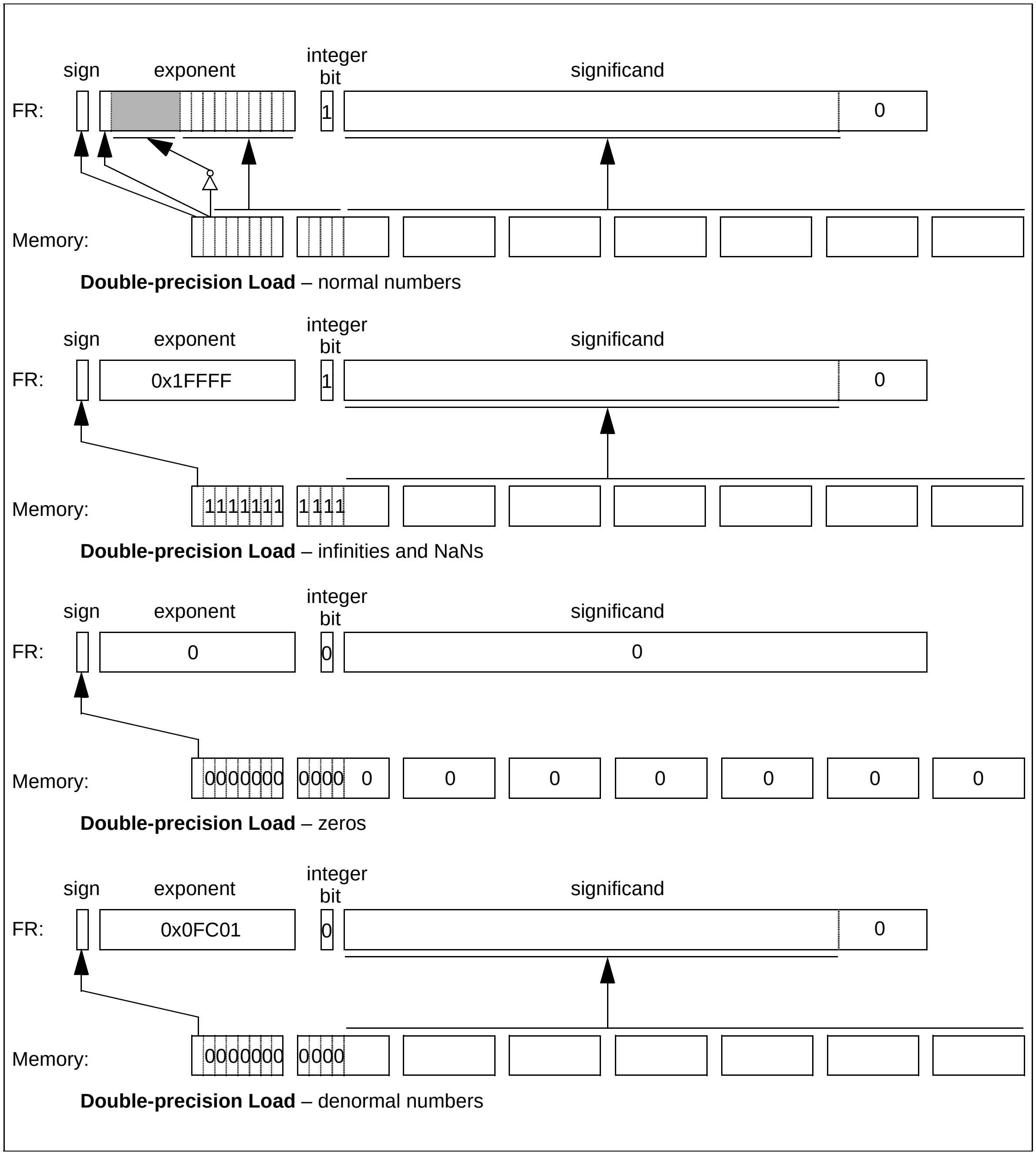


Figure 5-6. Memory to Floating-point Register Data Translation – Double Extended, Integer and Fill

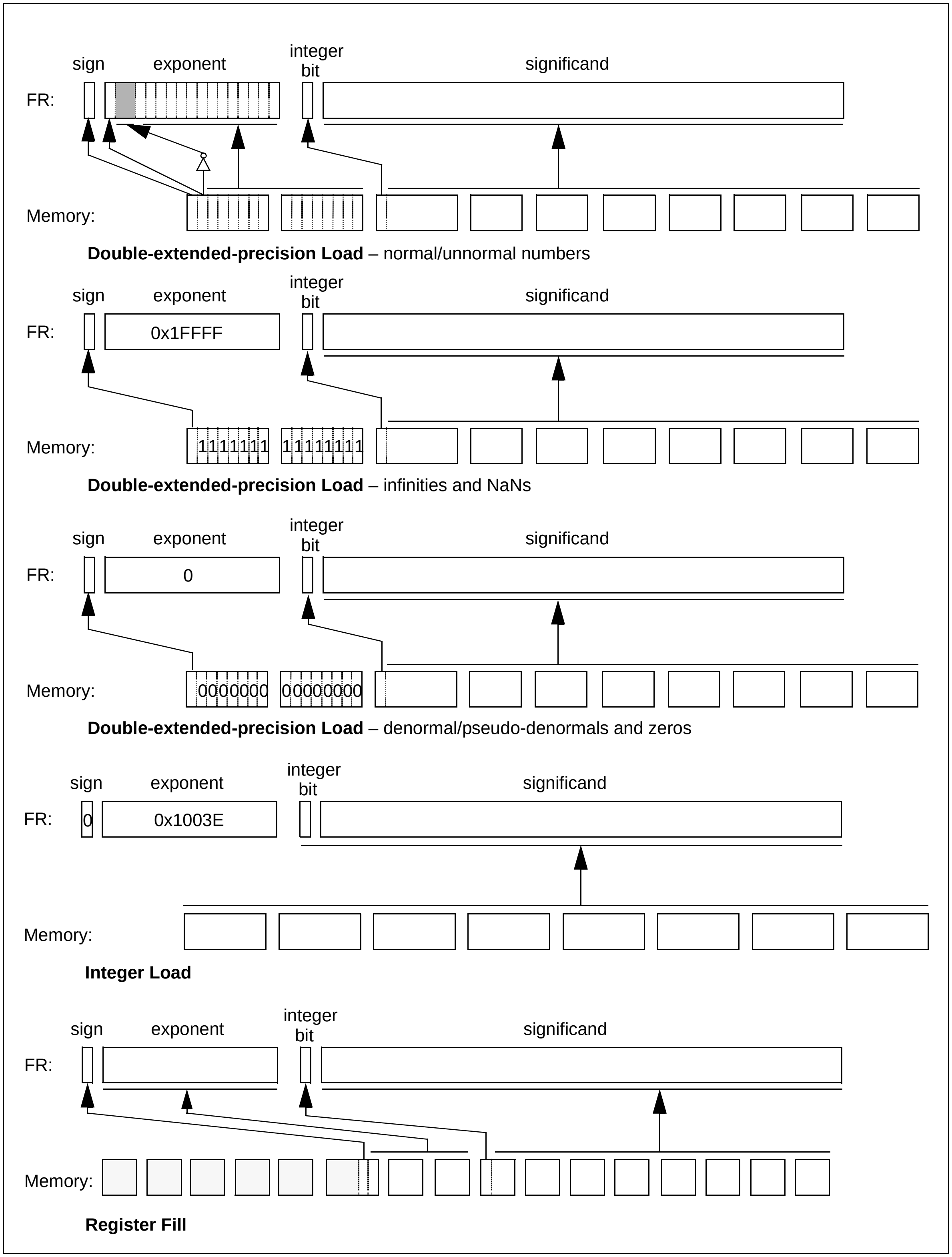
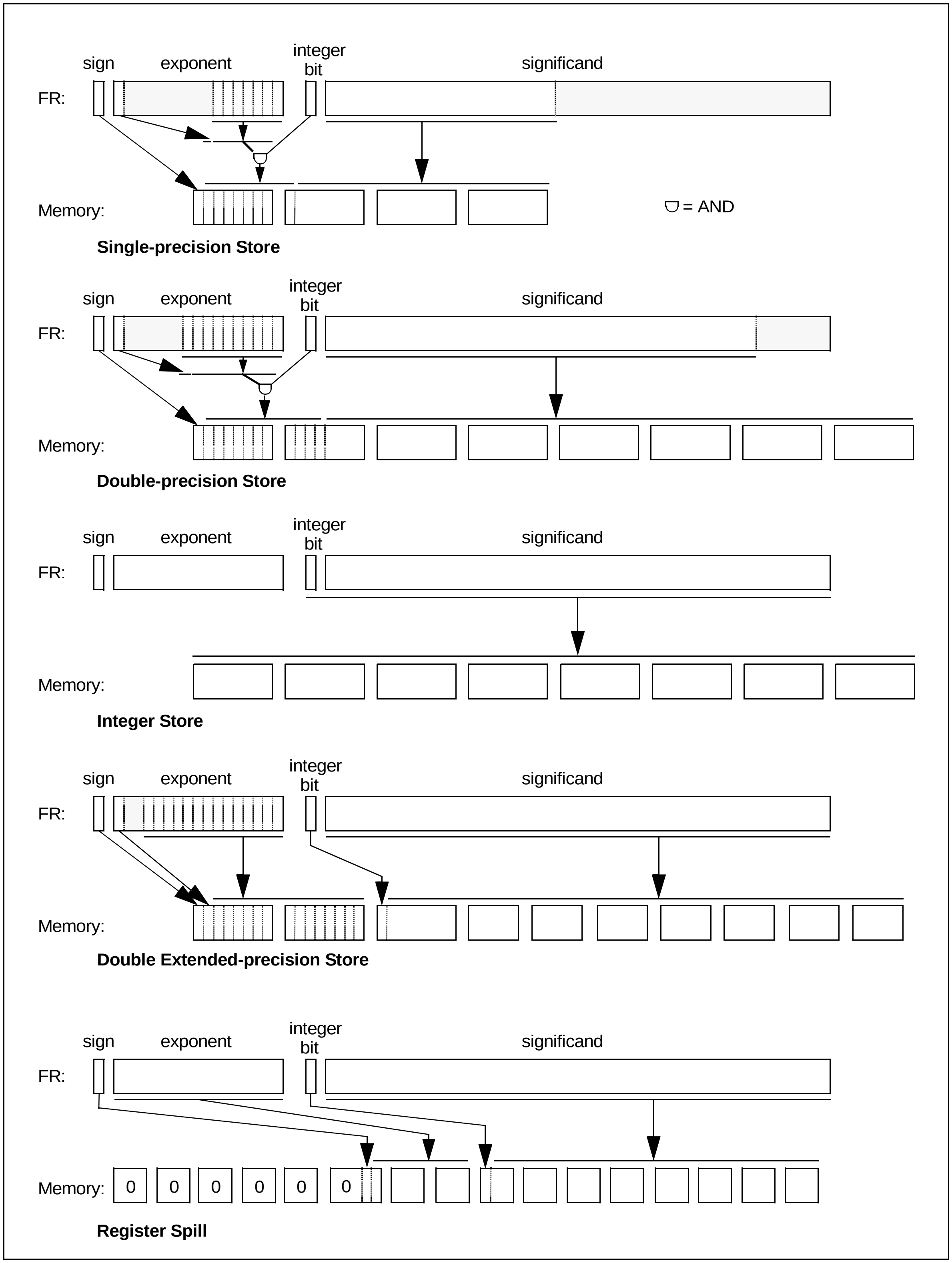
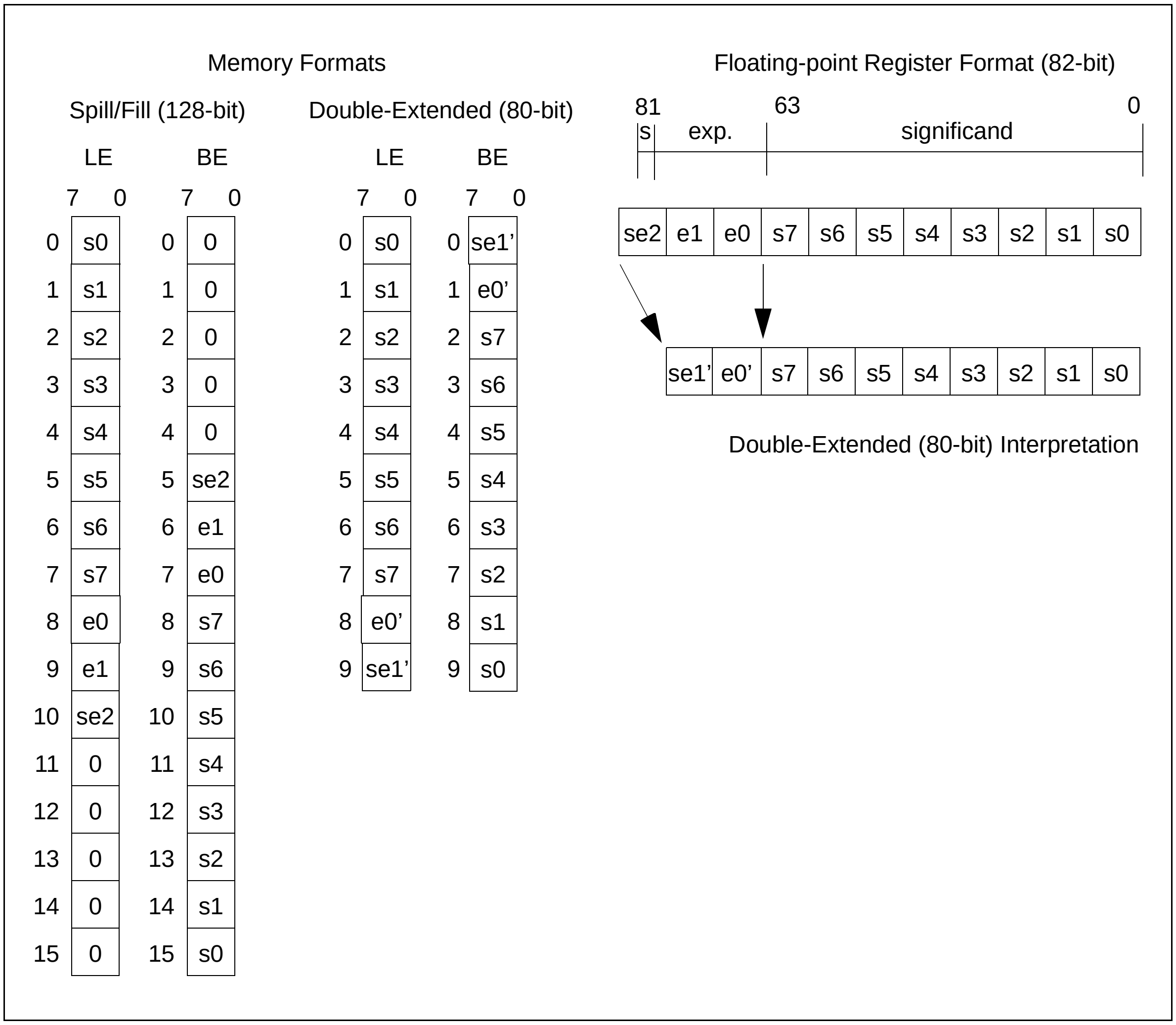

sign
exponent
integer bit
significand
FR:
Memory:
Double-extended-precision Load – normal/unnormal numbers
0x1FFFF
1111111
11111111
Double-extended-precision Load – infinities and NaNs
0
0000000
00000000
Double-extended-precision Load – denormal/pseudo-denormals and zeros
0
0x1003E
Integer Load
Register Fill

Figure 5-7. Floating-point Register to Memory Data Translation



Both little-endian and big-endian byte ordering is supported on floating-point loads and stores. For both single and double memory formats, the byte ordering is identical to the 32-bit and 64-bit integer data types (see Section 3.2.3). The byte-ordering for the spill/fill memory and double-extended formats is shown in Figure 5-8.

Figure 5-8. Spill/Fill and Double-Extended (80-bit) Floating-point Memory Formats



Memory Formats

	Spill/Fill (128-bit)		Double-Extended (80-bit)	
	LE	BE	LE	BE
	7 0	7 0	7 0	7 0
0	s0	0	s0	se1’
1	s1	0	s1	e0’
2	s2	0	s2	s7
3	s3	0	s3	s6
4	s4	0	s4	s5
5	s5	se2	s5	s4
6	s6	e1	s6	s3
7	s7	e0	s7	s2
8	e0	s7	e0’	s1
9	e1	s6	se1’	s0
10	se2	s5		
11	0	s4		
12	0	s3		
13	0	s2		
14	0	s1		
15	0	s0		

Floating-point Register Format (82-bit)
81
63
0
s
exp.
significand
se2 e1 e0 s7 s6 s5 s4 s3 s2 s1 s0
se1’ e0’ s7 s6 s5 s4 s3 s2 s1 s0
Double-Extended (80-bit) Interpretation

5.3.2 Floating-point Register to/from General Register Transfer Instructions

The `setf` and `getf` instructions (see Table 5-8) transfer data between floating-point registers (FR) and general registers (GR). These instructions will translate a general register NaT to/from a floating-point register NaTVal. For all other operands, the `.s` and `.d` variants of the `setf` and `getf` instructions translate to/from FR as per Figure 5-4, Figure 5-5 and Figure 5-7. The memory representation is read from or written to the GR. The `.exp` and `.sig` variants of the `setf` and `getf` instructions operate on the sign/exponent and significand portions of a floating-point register, respectively, and their translation formats are described in Table 5-9 and Table 5-10.

Table 5-8. Floating-point Register Transfer Instructions

Operations	GR to FR	FR to GR
Single	`setf.s`	`getf.s`
Double	`setf.d`	`getf.d`
Sign and Exponent	`setf.exp`	`getf.exp`
Significand/Integer	`setf.sig`	`getf.sig`

Table 5-9. General Register (Integer) to Floating-point Register Data Translation

Class	General Register		Floating-point Register (.sig)			Floating-point Register (.exp)		
	NaT	Integer	Sign	Exponent	Significand	Sign	Exponent	Significand
NaT	1	ignore	NaTVal			NaTVal		
integers	0	000...00 through 111...11	0	0x1003E	integer	integer{17}	integer{16:0}	0x8000000000000000

Table 5-10. Floating-point Register to General Register (Integer) Data Translation

Class	Floating-point Register			General Register (.sig)		General Register (.exp)	
	Sign	Exponent	Significand	NaT	Integer	NaT	Integer
NaTVal	0	0x1FFFE	0.000...00	1	0x0000000000000000	1	0x1FFFE
integers or parallel FP	0	0x1003E	0.000...00 through 1.111...11	0	significand	0	0x1003E
other	ignore	ignore	ignore	0	significand	0	((sign<<17) | exponent)

5.3.3 Arithmetic Instructions

All of the arithmetic floating-point instructions (except `fcvt.xf` which is always exact) have a *.sf* specifier. This indicates which of the four FPSR's status fields will both control and record the status of the execution of the instruction (see Table 5-11). The status field specifies: enabled exceptions, rounding mode, exponent width, precision control, and which status field's flags to update. See "Floating-point Status Register" on page 5-5.

Table 5-11. Floating-point Instruction Status Field Specifier Definition

.sf Specifier	.s0	.s1	.s2	.s3
Status field	FPSR.sf0	FPSR.sf1	FPSR.sf2	FPSR.sf3

Most arithmetic floating-point instructions can specify the precision of the result statically by using a *.pc* completer, or dynamically using the *.pc* field of the FPSR status field. (see Table 5-6). Arithmetic instructions that do not have a *.pc* completer use the floating-point register file range and precision.

Table 5-12 lists the floating-point arithmetic instructions and Table 5-13 lists the pseudo-operation definitions.

Table 5-12. Floating-point Arithmetic Instructions

Operation	Normal FP Mnemonic(s)	Parallel FP Mnemonic(s)
Floating-point multiply and add	`fma.`*`pc.sf`*	`fpma.`*`sf`*
Floating-point multiply and subtract	`fms.`*`pc.sf`*	`fpms.`*`sf`*
Floating-point negate multiply and add	`fnma.`*`pc.sf`*	`fpnma.`*`sf`*
Floating-point reciprocal approximation	`frcpa.`*`sf`*	`fprcpa.`*`sf`*
Floating-point reciprocal square root approximation	`frsqrta.`*`sf`*	`fprsqrta.`*`sf`*
Floating-point compare	`fcmp.`*`frel.fctype.sf`*	`fpcmp.`*`frel.sf`*
Floating-point minimum	`fmin.`*`sf`*	`fpmin.`*`sf`*
Floating-point maximum	`fmax.`*`sf`*	`fpmax.`*`sf`*
Floating-point absolute minimum	`famin.`*`sf`*	`fpamin.`*`sf`*
Floating-point absolute maximum	`famax.`*`sf`*	`fpamax.`*`sf`*

Convert floating-point to signed integer	`fcvt.fx.`*`sf`*
	`fcvt.fx.trunc.`*`sf`*
		`fpcvt.fx.`*`sf`*
		`fpcvt.fx.trunc.`*`sf`*

Convert floating-point to unsigned integer	`fcvt.fxu.`*`sf`*
	`fcvt.fxu.trunc.`*`sf`*
		`fpcvt.fxu.`*`sf`*
		`fpcvt.fxu.trunc.`*`sf`*

Convert signed integer to floating-point	`fcvt.xf`	`N.A.`

Table 5-13. Floating-point Pseudo-Operations

Operation	Mnemonic	Operation Used

Floating-point multiplication (IEEE)
Parallel FP multiplication
	`fmpy.`*`pc.sf`*
	`fpmpy.`*`sf`*
		`fma`, using FR 0 for addend
		`fpma`, using FR 0 for addend

Floating-point negate multiplication (IEEE)
Parallel FP negate multiplication
	`fnmpy.`*`pc.sf`*
	`fpnmpy.`*`sf`*
		`fnma`, using FR 0 for addend
		`fpnma`, using FR 0 for addend

Floating-point addition (IEEE)	`fadd.`*`pc.sf`*	`fma`, using FR 1 for multiplicand
Floating-point subtraction (IEEE)	`fsub.`*`pc.sf`*	`fms`, using FR 1 for multiplicand
Floating-point negation (IEEE)	`fnma.`*`pc.sf`*	`fnma`, using FR 1 for multiplicand and FR 0 for addend
Floating-point absolute value Parallel FP absolute value	`fabs` `fpabs`	`fmerge.s`, with sign from FR 0 `fpmerge.s`, with sign from FR 0
Floating-point negate Parallel FP negate	`fneg` `fpneg`	`fmerge.ns` `fpmerge.ns`

Table 5-13. Floating-point Pseudo-Operations (Cont'd)

Operation	Mnemonic	Operation Used
Floating-point negate absolute value Parallel FP negate absolute value	`fnegabs` `fpnegabs`	`fmerge.ns`, with sign from FR 0 `fpmerge.ns`, with sign from FR 0
Floating-point normalization	`fnorm.`*`pc.sf`*	`fma`, using FR 1 for multiplicand and FR 0 for addend
Convert unsigned integer to floating-point	`fcvt.xuf.`*`pc.sf`*	`fma`, using FR 1 for multiplicand and FR 0 for addend

There are no pseudo-operations for Parallel FP addition, subtraction, negation or normalization since FR 1 does not contain a packed pair of single precision 1.0 values. A parallel FP addition can be performed by first forming a pair of 1.0 values in a register (using the `fpack` instruction) and then using the `fpma` instruction. Similarly, an integer add operation can be generated by first forming an integer 1 in a floating-point register and then using the `xma` instruction.

5.3.4 Non-Arithmetic Instructions

Table 5-14 lists the non-arithmetic floating-point instructions. The `fclass` instruction is used to classify the contents of a floating-point register. The `fmerge` instruction is used to merge data from two floating-point registers into one floating-point register. The `fmix`, `fsxt`, `fpack`, and `fswap` instructions are used to manipulate the Parallel FP data in the floating-point significand. The `fand`, `fandcm`, `for`, and `fxor` instructions are used to perform logical operations on the floating-point significand. The `fselect` instruction is used for conditional selects.

The non-arithmetic floating-point instructions always use the floating-point register (82-bit) precision since they do not have a *.pc* completer nor a *.sf* specifier.

Table 5-14. Non-Arithmetic Floating-point Instructions

Operation	Mnemonic(s)
Floating-point classify	`fclass.`*`fcrel.fctype`*
Floating-point merge sign Parallel FP merge sign	`fmerge.s` `fpmerge.s`
Floating-point merge negative sign Parallel FP merge negative sign	`fmerge.ns` `fpmerge.ns`
Floating-point merge sign and exponent Parallel FP merge sign and exponent	`fmerge.se` `fpmerge.se`
Floating-point mix left	`fmix.l`
Floating-point mix right	`fmix.r`
Floating-point mix left-right	`fmix.lr`
Floating-point sign-extend left	`fsxt.l`
Floating-point sign-extend right	`fsxt.r`
Floating-point pack	`fpack`
Floating-point swap	`fswap`
Floating-point swap and negate left	`fswap.nl`
Floating-point swap and negate right	`fswap.nr`

Table 5-14. Non-Arithmetic Floating-point Instructions (Cont'd)

Operation	Mnemonic(s)
Floating-point And	fand
Floating-point And Complement	fandcm
Floating-point Or	for
Floating-point Xor	fxor
Floating-point Select	fselect

5.3.5 Floating-point Status Register (FPSR) Status Field Instructions

Speculation of floating-point operations requires that the status flags be stored temporarily in one of the alternate status fields (not FPSR.sf0). After a speculative execution chain has been committed, a fchkf instruction can be used to update the normal flags (FPSR.sf0.flags). This operation will preserve the correctness of the IEEE flags. The fchkf instruction does this by comparing the flags of the status field with the FPSR.sf0.flags and FPSR.traps. If the flags of the alternate status field indicate the occurrence of an event that corresponds to an enabled floating-point exception in FPSR.traps, or an event that is not already registered in the FPSR.sf0.flags (i.e., the flag for that event in FPSR.sf0.flags is clear), then the fchkf instruction causes a Speculative Operation fault. If neither of these cases arise then the fchkf instruction does nothing.

The fsetc instruction allows bit-wise modification of a status field's control bits. The FPSR.sf0.controls are ANDed with a 7-bit immediate and-mask and ORed with a 7-bit immediate or-mask to produce the control bits for the status field. The fclrf instruction clears all of the status field's flags to zero.

Table 5-15. FPSR Status Field Instructions

Operation	Mnemonic(s)
Floating-point check flags	fchkf.*sf*
Floating-point clear flags	fclrf.*sf*
Floating-point set controls	fsetc.*sf*

5.3.6 Integer Multiply and Add Instructions

Integer (fixed-point) multiply is executed in the floating-point unit using the three-operand xma instructions. The operands and result of these instructions are floating-point registers. The xma instructions ignore the sign and exponent fields of the floating-point register, except for a NaTVal check. The product of two 64-bit source significands is added to the third 64-bit significand (zero extended) to produce a 128-bit result. The low and high versions of the instruction select the appropriate low/high 64-bits of the 128-bit result, respectively, and write it into the destination register as a canonical integer. The signed and unsigned versions of the instructions treat the input registers as signed and unsigned 64-bit integers respectively.

Table 5-16. Integer Multiply and Add Instructions

Integer Multiply and Add	Low	High
Signed	xma.l	xma.h
Unsigned	xma.lu (pseudo-op)	xma.hu

5.4 Additional IEEE Considerations

5.4.1 Definition of SNaNs, QNaNs, and Propagation of NaNs

Signaling NaNs have a zero in the most significant fractional bit of the significand. Quiet NaNs have a one in the most significant fractional bit of the significand. This definition of signaling and quiet NaNs easily preserves "NaNness" when converting between different precisions. When propagating NaNs in operations that have more than one NaN operand, the result NaN is chosen from one of the operand NaNs in the following priority based on register encoding fields: first *`f4,`* then *`f2`*, and lastly *`f3.`*

5.4.2 IEEE Standard Mandated Operations Deferred to Software

The following IEEE mandated operations will be implemented in software:

- String to floating-point conversion.
- Floating-point to string conversion.
- Divide (with help from `frcpa` or `fprcpa` instruction).
- Square root (with help from `frsqrta` or `fprsqrta` instruction).
- Remainder (with help from `frcpa` or `fprcpa` instruction).
- Floating-point to integer valued floating-point conversion.
- Correctly wrapping the exponent for single, double, and double-extended overflow and underflow values, as recommended by the IEEE standard.

5.4.3 Additions beyond the IEEE Standard

- The fused multiply and add (`fma, fms, fnma, fpma, fpms, fpnma`) operations enable efficient software divide, square root, and remainder algorithms.
- The extended range of the 17-bit exponent in the register file format allows simplified implementation of many basic numeric algorithms by the careful numeric programmer.
- The NaTVal is a natural extension of the IEEE concept of NaNs. It is used to support speculative execution.
- Flush-to-Zero mode is an industry standard addition.
- The minimum and maximum instructions allow the efficient execution of the common Fortran Intrinsic Functions: MIN(), MAX(), AMIN(), AMAX(); and C language idioms such as a<b?a:b.
- All mixed precision operations are allowed. The IEEE standard suggests that implementations allow lower precision operands to produce higher precision results; this is supported. The IEEE standard also suggests that implementations not allow higher precision operands to produce lower precision results; this suggestion is not followed.
- An IEEE style quad-precision real type that is supported in software.

IA-32 Application Execution Model in an IA-64 System Environment 6

The IA-64 architecture enables the execution of IA-32 application binaries unmodified on IA-32 legacy operating systems provided the required platform and firmware support exists in the system.

This chapter describes IA-32 instruction execution in an IA-64 System Environment. The IA-64 architecture supports 16-bit Real Mode, 16-bit VM86, and 16-bit/32-bit Protected Mode IA-32 applications running on IA-64 operating system. IA-64 operating system support for these capabilities is defined by the respective operating system vendors.

The main features covered in this chapter are:

- IA-32 and IA-64 instruction set transitions.
- IA-32 integer, segment, floating-point, MMX technology, and Streaming SIMD Extension register state mappings.
- IA-32 memory and addressing model overview.

This chapter does not cover the details of IA-32 application programming model, IA-32 instructions and registers. Refer to the *Intel Architecture Software Developer's Manual* for details regarding IA-32 application programming model.

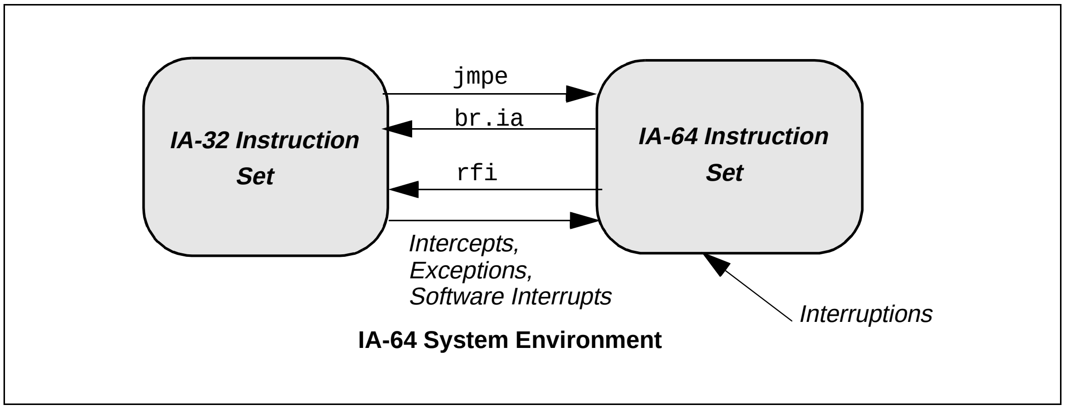
6.1 Instruction Set Modes

The processor can execute either IA-32 or IA-64 instructions. A bit in Processor Status Register (PSR) specifies the currently executing instruction set. Three special instructions and interruptions are defined to transition the processor between the IA-32 and the IA-64 instruction sets as shown in Figure 6-1.

- `jmpe` (IA-32 instruction) Jump to an IA-64 target instruction, and change the instruction set to IA-64.
- `br.ia` (IA-64 instruction) IA-64 branch to an IA-32 target instruction, and change the instruction set to IA-32.
- `rfi` (IA-64 instruction) Return from interruption, is defined to return to either an IA-32 or IA-64 instruction when resuming from an interruption.
- Interruptions transition the processor to the IA-64 instruction set for all interruption conditions.

The `jmpe` and `br.ia` instructions provide a low overhead mechanism to transfer control between the instruction sets. These primitives typically are incorporated into “thunks” or “stubs” that implement the required call linkage and calling conventions to call dynamic or statically linked libraries.

Figure 6-1. IA-64 Processor Instruction Set Transition Model



6.1.1 IA-64 Instruction Set Execution

While the processor executes from the IA-64 instruction set:

- IA-64 instructions are fetched, decoded and executed by the processor.
- IA-64 instructions can access the entire IA-64 and IA-32 application register state. This includes IA-32 segment descriptors, selectors, general registers, physical floating-point registers, MMX technology registers, and Streaming SIMD Extension registers. See Section 6.2 for a description of the register state mapping.
- Segmentation is disabled. No segmentation protection checks are applied nor are segment bases added to compute virtual addresses, i.e. all computed addresses are virtual addresses.
- 2^{64} virtual addresses can be generated and IA-64 memory management is used for all memory and I/O references.

6.1.2 IA-32 Instruction Set Execution

While the processor is executing the IA-32 instruction set within the IA-64 System Environment, the IA-32 application architecture as defined by the Pentium® III processor is used, namely:

- IA-32 16/32-bit application level, MMX technology instructions and Streaming SIMD Extension instructions are fetched, decoded, and executed by the processor. Instructions are confined to 32/16-bit operations.
- Only IA-32 application level register state is visible (i.e. IA-32 general registers, MMX technology registers and Streaming SIMD Extension registers, selectors, EFLAGS, FP registers and FP control registers). IA-64 application and control state is not visible, e.g. branch, predicate, application, etc.
- IA-32 Real Mode, VM86 and Protected Mode segmentation is in effect. Segment protection checks are applied and virtual addresses generated according to IA-32 segmentation rules. GDT and LDT segments are defined to support IA-32 segmented applications. Segmented 16- and 32-bit code is fully supported.
- IA-64 memory management is used to translate virtual to physical addresses for all IA-32 instruction set memory and I/O Port references.
- Instruction and Data memory references are forced to be little-endian. Memory ordering uses the Pentium III processor memory ordering model.

- IA-32 operating system resources; IA-32 paging, MTRRs, IDT, control registers, debug registers and privileged instructions are superceded by IA-64 defined resources. All accesses to these resources result in an interception fault.

6.1.3 Instruction Set Transitions

The following section summarizes behavior for each instruction set transition. Detailed instruction description on `jmpe` (IA-32 instruction) and `br.ia` (IA-64 instruction) should be consulted for details. Operating systems can disable instruction set transitions.

6.1.3.1 JMPE Instruction

`jmpe reg16/32; jmpe disp16/32` is used to jump and transfer control to the IA-64 instruction set. There are two forms; register indirect and absolute. The absolute form computes the virtual IA-64 target address as follows:

```
IP{31:0} =disp16/32 + CSD.base
IP{63:32} = 0
```

The indirect form reads a 16/32-bit register location and then computes the IA-64 target address as follows:

```
IP{31:0} = [reg16/32] + CSD.base
IP{63:32} = 0
```

IA-64 `jmpe` targets are forced to be 16-byte aligned, and are constrained to the lower 4G-bytes of the 64-bit virtual address space due to limited IA-32 addressability. If there are any pending IA-32 numeric exceptions, `jmpe` is nullified, and an IA-32 floating-point exception fault is generated.

6.1.3.2 Branch to IA Instruction

Unconditional branches to the IA-32 instruction set use the IA-64 defined indirect branch mechanism. IA-32 targets are specified by a 32-bit virtual address target (not an effective address). The IA-32 virtual address is truncated to 32-bits. The `br.ia` branch hints should always be set to predicted static taken. The processor transitions to the IA-32 instruction set as follows:

```
IP{31:0} = BR[b]{31:0}
IP{63:32} = 0
EIP{31:0} = IP{31:0} - CSD.base
```

Transitions into the IA-32 instruction set do not change the privilege level of the processor.

Software should ensure the code segment descriptor and selector are properly 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 GPFault(0) exception is reported on the target IA-32 instruction.

The processor does not ensure IA-64 instruction set generated writes into the IA-32 instruction stream are observed by the processor. For details, see "Self Modifying Code" on page 6-24. Before entering the IA-32 instruction set, IA-64 software must ensure all prior register stack frames have been flushed to memory. All registers left in the current and prior register stack frames are modified during IA-32 instruction set execution. For details, see "IA-64 Register Stack Engine" on page 6-24.

6.1.4 IA-32 Operating Mode Transitions

As described in "IA-32 Instruction Set Execution" on page 6-2, `jmpe`, `br.ia`, and `rfi` and interruptions can transition the processor between the two instruction set modes. Transitions are allowed between all major IA-32 modes and IA-64. `jmpe` and interruptions will transition the processor from either IA-32 VM86, Real Mode or Protected Mode into the IA-64 instruction set mode. Mode transitions between IA-32 Real Mode, Protected Mode and VM86 definitions are the same as those defined in the *Intel Architecture Software Developer's Manual*.

IA-64 interface code is responsible for setting up and loading a consistent Protected Mode, Real Mode, or VM86 environment (e.g. loading segment selectors and descriptors, etc.) as defined in "Segment Descriptor and Environment Integrity" on page 6-10. The processor applies additional segment descriptor checks to ensure operations are performed in a consistent manner.

6.2 IA-32 Application Register State Model

As shown in Figure 6-2 and Table 6-1, IA-32 general purpose registers, segment selectors, and segment descriptors, are mapped into the lower 32-bits of IA-64 general purpose registers GR8 to GR31. The floating-point register stack, MMX technology registers and Streaming SIMD Extension registers are mapped on IA-64 floating-point registers FR8 to FR31.

To promote straight-forward parameter passing, IA-32 and IA-64 integer and IEEE floating-point register and memory data types are binary compatible between both IA-32 and IA-64 instruction sets.

Some IA-64 registers are modified as a side-effect during IA-32 instruction set execution as noted in Figure 6-2 and Table 6-1. Generally, IA-64 system state is not affected by IA-32 instruction set execution. IA-64 code can reference all IA-64 and IA-32 registers, while IA-32 instruction set references are confined to the IA-32 visible application register state.

Registers are assigned the following conventions during transitions between IA-32 and IA-64 instruction sets.

- **IA-32 state**: The register contains an IA-32 register during IA-32 instruction set execution. Expected IA-32 values should be loaded before switching to the IA-32 instruction set. After completion of IA-32 instructions, these registers contain the results of the execution of IA-32 instructions. These registers may contain any value during IA-64 instruction execution according to IA-64 software conventions. Software should follow IA-32 and IA-64 calling conventions for these registers.
- **Modified**: Registers marked as modified are used as scratch areas for execution of IA-32 instructions by the processor and are not ensured to be preserved across instruction set transitions.
- **Shared**: Shared registers contain values that have similar functionality in either instruction set. For example, the stack pointer (ESP) and instruction pointer (IP) are shared.
- **Unmodified**: These registers are not altered by IA-32 execution. IA-64 code can rely on these values not being modified during IA-32 instruction set execution. The register will have the same contents when entering the IA-32 instruction set and when exiting the IA-32 instruction set.

Figure 6-2. IA-32 Application Register Model

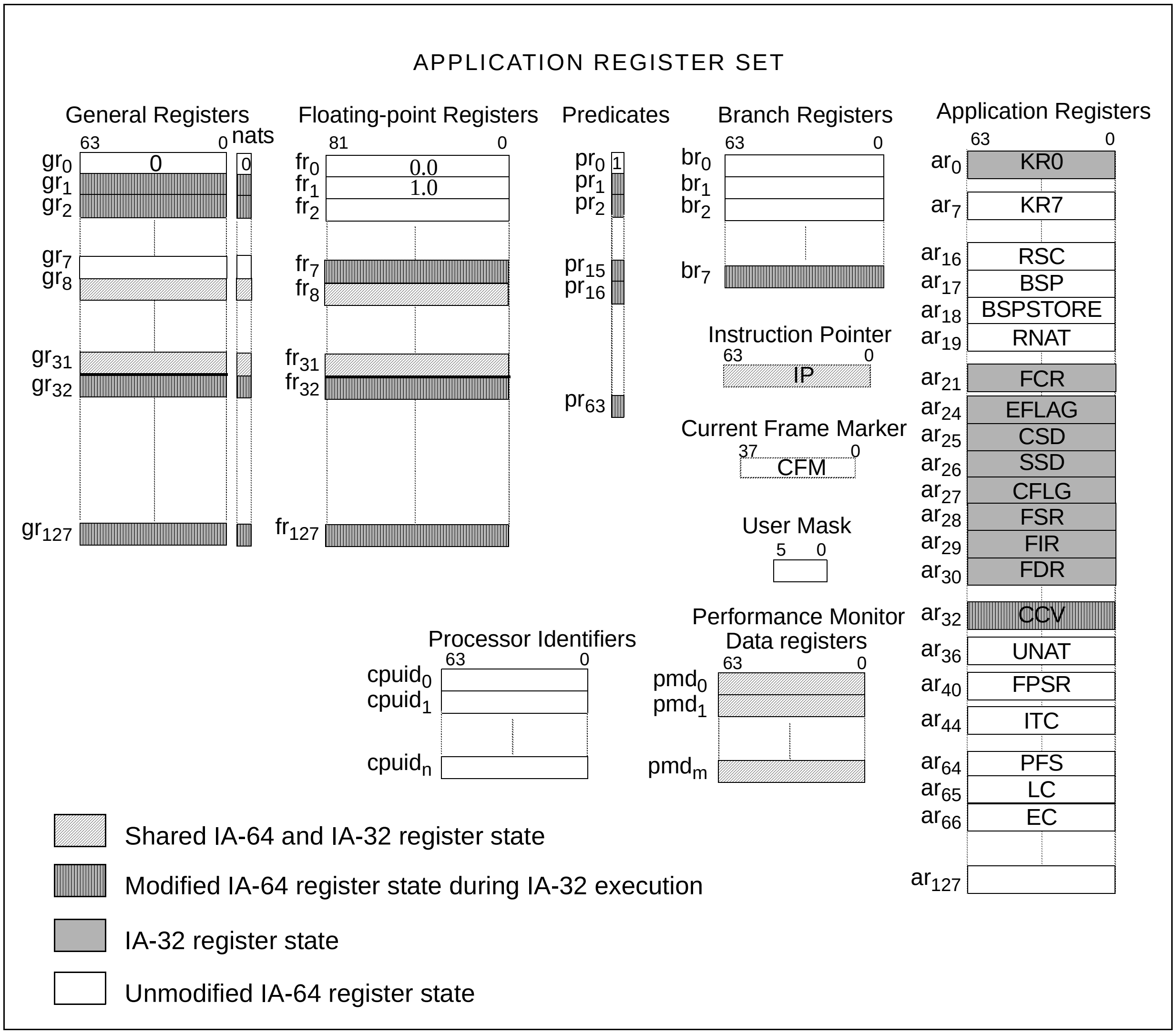


Table 6-1. IA-32 Application Register Mapping

IA-64 Reg	IA-32 Reg	Convention	Size	Description
General Purpose Integer Registers				
GR0				constant 0
GR1-3		modified[f]		scratch for IA-32 execution
GR4-7		unmodified		IA-64 preserved registers
GR8	EAX	IA-32 state	32[a]	IA-32 general purpose registers
GR9	ECX			
GR10	EDX			
GR11	EBX			
GR12	ESP			
GR13	EBP			
GR14	ESI			
GR15	EDI			
GR16{15:0}	DS		64	IA-32 selectors
GR16{31:16}	ES			
GR16{47:32}	FS			
GR16{63:48}	GS			
GR17{15:0}	CS			
GR17{31:16}	SS			
GR17{47:32}	LDT			
GR17{63:48}	TSS			
GR18-23		modified[f]		scratch for IA-32 execution
GR24	ESD	IA-32 state	64	IA-32 segment descriptors (register format)[b]
GR25-26		modified[f]		scratch for IA-32 execution
GR27	DSD	IA-32 state	64	IA-32 segment descriptors (register format)[b]
GR28	FSD			
GR29	GSD			
GR30	LDTD[c]			
GR31	GDTD			
GR32-127		modified[d]		IA-32 code execution space
Process Environment				
IP	IP	shared	64	shared IA-32 and IA-64 virtual Instruction Pointer
Floating-point Registers				
FR0				constant +0.0
FR1				constant +1.0
FR2-5		unmodified		IA-64 preserved registers
FR6-7		modified	64	IA-32 code execution space

Table 6-1. IA-32 Application Register Mapping (Cont'd)

IA-64 Reg	IA-32 Reg	Convention	Size	Description
FR8	MM0/FP0	IA-32 state	64/ 80	IA-32 MMX™ technology registers (aliased on 64-bit FP mantissa) IA-32 FP registers (physical registers mapping)[e]
FR9	MM1/ FP1			
FR10	MM2/FP2			
FR11	MM3/FP3			
FR12	MM4/ FP4			
FR13	MM5/FP5			
FR14	MM6/FP6			
FR15	MM7/FP7			

FR16-17	XMM0	IA-32 state	64	IA-32 Streaming SIMD Extension registers
				low order 64-bits of XMM0 are mapped to FR16{63:0}
				high order 64-bits of XMM0 are mapped to FR17{63:0}

FR18-19	XMM1			
FR20-21	XMM2			
FR22-23	XMM3			
FR24-25	XMM4			
FR26-27	XMM5			
FR28-29	XMM6			
FR30-31	XMM7			
FR32-127		modified[f]		IA-32 code execution space
Predicate Registers				
PR0				constant 1
PR1-63		modified[f]		IA-32 code execution space
Branch Registers				
BR0-5		unmodified		IA-64 preserved registers
BR6-7		modified		IA-32 code execution space
Application Registers				
RSC		unmodified		not used for IA-32 execution IA-64 preserved registers
BSP				
BSPSTORE				
RNAT				
CCV		modified[f]	64	IA-32 code execution space
UNAT		unmodified		not used for IA-32 execution, IA-64 preserved
FPSR.sf0		unmodified		IA-64 numeric status and controls
FPSR.sf1,2,3		modified[f]		IA-32 code execution space, modified during IA-32 execution.

Table 6-1. IA-32 Application Register Mapping (Cont'd)

IA-64 Reg	IA-32 Reg	Convention	Size	Description
FSR	FSW,FTW, MXCSR	IA-32 state	64	IA-32 numeric status and tag word and Streaming SIMD Extension status
FCR	FCW, MXCSR		64	IA-32 numeric and Streaming SIMD Extension control
FIR	FOP, FIP, FCS		64	IA-32 x87 numeric environment opcode, code selector and IP
FDR	FEA, FDS		64	IA-32 x87 numeric environment data selector and offset
ITC	TSC	shared	64	shared IA-32 time stamp counter (TSC) and IA-64 Interval Timer
PFS		unmodified		not used for IA-32 code execution, Prior EC is preserved in PFM IA-64 preserved registers
LC				
EC				
EFLAG	EFLAG	IA-32 state	32	IA-32 System/Arithmetic flags, writes of some bits condition by CPL and EFLAG.iopl.
CSD	CSD		64	IA-32 code segment (register format)[b]
SSD	SSD			IA-32 stack segment (register format)[b]
CFLG	CR0/CR4		64	IA-32 control flags CR0=CFLG{31:0}, CR4=CFLG{63:32}, writable at CPL=0 only.

a. On transitions into the IA-32 instruction set the upper 32-bits are ignored. On exit the upper 32-bits are sign extended from bit 31.
b. Segment descriptor formats differ from the IA-32 memory format, see "IA-32 Segment Registers" on page 6-9 for details. Modification of a selector or descriptor does not set the access/busy bit in memory.
c. The GDT/LDT descriptors are NOT protected from modification by IA-64 user level code.
d. All registers in the current and prior registers frames are modified during IA-32 execution.
e. IA-32 floating-point register mappings are physical and do not reflect the IA-32 top of stack value.
f. These registers are used by the processor and may be modified. Software should preserve required values before entering IA-32 code.

6.2.1 IA-32 General Purpose Registers

Integer registers are mapped into the lower 32-bits of IA-64 general registers GR8 to GR15. Values in the upper 32-bits of GR8 to 15 are ignored on entry to IA-32 execution. After the IA-32 instruction set completes execution, the upper 32-bits of GR8 - GR15 are sign-extended from bit 31.

Based on IA-32 and IA-64 calling conventions, the required IA-32 state must be loaded in memory or registers by IA-64 code before entering the IA-32 instruction set.

Figure 6-3. IA-32 General Registers (GR8 to GR15)

63 … 32	31 … 0
sign extended	EAX.. EDI{31:0}

6.2.2 IA-32 Instruction Pointer

The processor maintains two instruction pointers for IA-32 instruction set references, EIP (32-bit effective address) and IP (a 64-bit virtual address equivalent to the IA-64 instruction set IP). IP is generated by adding the code segment base to EIP and zero extending to 64-bits. IP should not be confused with the 16-bit effective address instruction pointer of the 8086. EIP is an offset within the current code segment, while IP is a 64-bit virtual pointer shared with the IA-64 instruction set. The following relationship is defined between EIP and IP while executing IA-32 instructions.

```
IP{63:32} = 0;
IP{31:0} = EIP{31:0} + CSD.Base;
```

EIP is added to the code segment base and zero extended into a 64-bit virtual address on every IA-32 instruction fetch. If during an IA-32 instruction fetch, EIP exceeds the code segment limit a GPFault is generated on the referencing instruction. Effective instruction addresses (sequential values or jump targets) above 4G-bytes are truncated to 32 bits, resulting in a 4-G byte wrap around condition.

6.2.3 IA-32 Segment Registers

IA-32 segment selectors and descriptors are mapped to GR16 - GR29 and AR25 - AR26. Descriptors are maintained in an unscrambled format shown in Figure 6-5. This format differs from the IA-32 scrambled memory descriptor format. The unscrambled register format is designed to support fast conversion of IA-32 segmented 16/32-bit pointers into virtual addresses by IA-64 code. IA-32 segment register load instructions unscramble the GDT/LDT memory format into the descriptor register format on a segment register load. IA-64 software can also directly load descriptor registers provided they are properly unscrambled by software. For a complete definition of all bit fields and field semantics refer to the *Intel Architecture Software Developer's Manual*.

Figure 6-4. IA-32 Segment Register Selector Format

63 48	47 32	31 16	15 0	
GS	FS	ES	DS	GR16
TSS	LDT	SS	CS	GR17

Figure 6-5. IA-32 Code/Data Segment Register Descriptor Format

63	62	61	60	59	58 57	56	55 52	51 32	31 0
g	d/b	ig	av	p	dpl	s	type	lim{19:0}	base{31:0}

Table 6-2. IA-32 Segment Register Fields

Field	Bits	Description
selector	15:0	Segment Selector value, see the *Intel Architecture Software Developer's Manual* for bit definition.
base	31:0	Segment Base value. This value when zero extended to 64-bits, points to the start of the segment in the 64-bit virtual address space for IA-32 instruction set memory references.
lim	51:32	Segment Limit. Contains the maximum effective address value within the segment for expand up segments for IA-32 instruction set memory references. For expand down segments, limit defines the minimum effective address within the segment. See the *Intel Architecture Software Developer's Manual* for details and segment limit fault conditions. The segment limit is scaled by (lim << 12) | 0xFFF if the segment's g-bit is 1.
type	55:52	Type identifier for data/code segments, including the Access bit (bit 52). See the *Intel Architecture Software Developer's Manual* for encodings and definition.
s	56	Non System Segment. If 1, a data segment, if 0 a system segment.
dpl	58:57	Descriptor Privilege Level. The DPL is checked for memory access permission for IA-32 instruction set memory references.

Table 6-2. IA-32 Segment Register Fields (Cont'd)

Field	Bits	Description
p	59	Segment Present bit. If 0, and a IA-32 memory reference uses this segment an IA_Exception (GPFault) is generated for data segments (CS, DS, ES, FS, GS) and an IA-32_Exception (StackFault) for SS.
av	60	Ignored - For the CS, SS descriptors reads of this field return zeros. For the DS, ES, FS, and GS descriptors reads of this field return the last value written by IA-64 code. Reads of this field return zero if written by IA-32 descriptor loads.This field is ignored by the processor during IA-32 instruction set execution. Available for software use, there will be no future use for this field.
ig	61	Ignored - For the CS, SS descriptors reads of this field return zeros. For the DS, ES, FS, and GS descriptors reads of this field return the last value written by IA-64 code. Reads of this field return zero if written by IA-32 descriptor loads.This field is ignored by the processor during IA-32 instruction set execution. This field may have a future use and should be set to zero by software.
d/b	62	Segment Size. If 0, IA-32 instruction set effective addresses within the segment are truncated to 16-bits. Otherwise, effective addresses are 32-bits. The code segment's d/b-bit also controls the default operand size for IA-32 instructions. If 1, the default operand size is 32-bits, otherwise 16-bits.
g	63	Segment Limit Granularity. If 1, scales the segment limit by lim=(lim<<12) | 0xFFF for IA-32 instruction set memory references. This field is ignored for IA-64 instruction set memory references.

6.2.3.1 Data and Code Segments

On the transition into IA-32 code, the IA-32 segment descriptor and selector registers (GDT, LDT, DS, ES, CS, SS, FS and GS) must be initialized by IA-64 code to the required values based on IA-32 and IA-64 calling conventions and the segmentation model used.

IA-64 code may manually load a descriptor with an 8-byte fetch from the LDT/GDT, unscramble the descriptor and write the segment base, limit and attribute. Alternately, IA-64 software can switch to the IA-32 instruction set and perform the required segment load with an IA-32 Mov Sreg instruction. If IA-64 code explicitly loads the segment descriptors, it is responsible for the integrity of the segment descriptor.

The processor does not ensure coherency between descriptors in memory and the descriptor registers, nor does the processor set segment access bits in the LDT/GDT if segment registers are loaded by IA-64 instructions.

6.2.3.2 Segment Descriptor and Environment Integrity

For IA-32 instruction set execution, most segment protection checks are applied by the processor when the segment descriptor is loaded by IA-32 instructions into a segment register. However, segment descriptor loads from the IA-64 instruction set into the general purpose register file perform no such protection checks, nor are segment Access-bits updated by the processor.

If IA-64 software directly loads a descriptor it is responsible for the validity of the descriptor, and ensuring integrity of the IA-32 Protected Mode, Real Mode or VM86 environments. Table 6-3 defines software guidelines for establishing the initial IA-32 environment. The processor checks the integrity of the IA-32 environment as defined in Section 6.2.3.3, "IA-32 Environment Run-time Integrity Checks" on page 6-13. On the transitions between IA-64 and IA-32 code, the processor does NOT alter the base, limit or attribute values of any segment descriptor, nor is there a change in privilege level.

Table 6-3. IA-32 Environment Initial Register State

Register	Field	Real Mode	Protected Mode	VM86Mode
PSR	**cpl**	**0**	**privilege level**	**3**
EFLAG	**vm**	**0**	**0**	**1**
CR0	**pe**	**0**	**1**	**1**
CS	selector	base >> 4[a]	selector	base >> 4
	base	selector << 4[b]	base	selector << 4
	dpl	PSR.cpl (0)	PSR.cpl[c]	PSR.cpl (3)
	d-bit	16-bit[d]	16/32-bit	16-bit
	type	data rd/wr, expand up	execute	data rd/wr, expand up
	s-bit	1	1	1
	p-bit	1	1	1
	a-bit	1	1	1
	g-bit/limit	0xFFFF[e]	limit	0xFFFF
SS	selector	base >> 4[a]	selector	base >> 4
	base	selector << 4[b]	base	selector << 4
	dpl	PSR.cpl (0)	PSR.cpl	PSR.cpl (3)
	d-bit	16-bit[d]	16/32-bit size	16-bit
	type	data rd/wr, expand up	data types	data rd/wr, expand up
	s-bit	1	1	1
	p-bit	1	1	1
	a-bit	1	1	1
	g-bit/limit	0xFFFF[e]	limit	0xFFFF
DS, ES, FS, GS	selector	base >> 4[a]	selector	base >> 4
	base	selector << 4[b]	base	selector << 4
	dpl	dpl >= PSR.cpl (0)	dpl >= PSR.cpl	dpl >= PSR.cpl (3)
	d-bit	16-bit[d]	16/32-bit	0
	type	data rd/wr, expand up	data types	data rd/wr, expand up
	s-bit	1	1	1
	a-bit	1	1	1
	p-bit	1	1/0[f]	1
	g-bit/limit	0xFFFF[e]	limit	0xFFFF

Table 6-3. IA-32 Environment Initial Register State (Cont'd)

Register	Field	Real Mode	Protected Mode	VM86Mode
PSR	cpl	0	privilege level	3
EFLAG	vm	0	0	1
CR0	pe	0	1	1
LDT,GDT, TSS	selector	na	selector	
	base		base	
	dpl		dpl >= PSR.cpl	
	d-bit		0	
	type		ldt/gdt/tss types	
	s-bit		0	
	p-bit		1	
	a-bit		1	
	g-bit/limit		limit	

a. Selectors should be set to 16*base for normal RM 64KB operation.
b. Segment base should be set to selector/16 for normal RM 64KB operation.
c. Unless a conforming code segment is specified.
d. Segment size should be set to 16-bits for normal RM 64KB operation.
e. Segment limit should be set to 0xFFFF for normal RM 64KB operation.
f. For valid segments the p-bit should be set to 1, for null segments the p-bit should be set to 0.

6.2.3.2.1 Protected Mode

IA-64 software should follow these rules for setting up the segment descriptors for Protected Mode environment before entering the IA-32 instruction set:

- IA-64 software should ensure the stack segment descriptor register's DPL==PSR.cpl.
- For DSD, ESD, FSD and GSD segment descriptor registers, IA-64 software should ensure DPL>=PSR.cpl.
- For CSD segment descriptor register, IA-64 software should ensure DPL==PSR.cpl (except for conforming code segments).
- Software should ensure that all code, stack and data segment descriptor registers do not contain encodings for any system segments.
- Software should ensure the a-bit of all segment descriptor registers are set to 1.
- Software should ensure the p-bit is set to 1 for all valid data segments and to 0 for all NULL data segments.

6.2.3.2.2 VM86

IA-64 software should follow these rules when setting up segment descriptors for the VM86 environment before entering the IA-32 instruction set:

- PSR.cpl must be 3 (or IPSR.cpl must be 3 for `rfi`).
- IA-64 software should ensure the stack segment descriptor register's DPL==PSR.cpl==3 and set to 16-bit, data read/write, expand up.
- For CSD, DSD, ESD, FSD and GSD segment descriptor registers, IA-64 software should ensure DPL==3, the segment is set to 16-bit, data read/write, expand up.

- Software should ensure that all code, stack and data segment descriptor registers do not contain encodings for any system segments.
- Software should ensure the P-bit and A-bit of all segment descriptor registers is one.
- Software should ensure that the relationship Base = Selector*16, is maintained for all DSD, CSD, ESD, SSD, FSD, and GSD segment descriptor registers, otherwise processor operation is unpredictable.
- Software should ensure that the DSD, CSD, ESD, SSD, FSD, and GSD segment descriptor register's limit value is set to 0xFFFF, otherwise spurious segment limit faults (GPFault or Stack Faults) may be generated.
- IA-64 software should ensure all segment descriptor registers are data read/write, including the code segment. The processor will ignore execute permission faults.

6.2.3.2.3 Real Mode

IA-64 software should follow these rules when setting up segment descriptors for the Real Mode environments before entering the IA-32 instruction set, otherwise software operation is unpredictable.

- IA-64 software should ensure PSR.cpl is 0.
- IA-64 software should ensure the stack segment descriptor register's DPL is 0.
- Software should ensure that all code, stack and data segment descriptor registers do not contain encodings for any system segments.
- Software should ensure the P-bit and A-bit of all segment descriptor registers is one.
- For normal real mode 64K operations, software should ensure that the relationship Base = Selector*16, is maintained for all DSD, CSD, ESD, SSD, FSD, and GSD segment descriptor registers.
- For normal real mode 64K operations, software should ensure that the DSD, CSD, ESD, SSD, FSD, and GSD segment descriptor register's limit value is set to 0xFFFF and the segment size is set to 16-bit (64K).
- IA-64 software should ensure all segment descriptor registers indicate readable, writable, including the code segment for normal Real Mode operation.

6.2.3.3 IA-32 Environment Run-time Integrity Checks

IA-64 processors perform additional run-time checks to verify the integrity of the IA-32 environments. These checks are in addition to the run-time checks defined on IA-32 processors and are high-lighted in Table 6-4. Existing IA-32 run-time checks are listed but not highlighted. Descriptor fields not listed in the table are not checked. As defined in the table, run-time checks are performed either on IA-32 instruction code fetches or on an IA-32 data memory reference to one of the specified segment registers. These run-time checks are not performed during IA-64 to IA-32 instruction set transitions.

Table 6-4. IA-32 Environment Run Time Integrity Checks

Reference	**Resource**	**Real Mode**	**Protected Mode**	**VM86Mode**	**Fault**
all code fetches	**PSR.cpl**	**is not 0**	**ignored**	**is not 3**	**Code Fetch Fault (GPFault(0))**[a]
	EFLAG.vm **CFLG.pe**	**EFLAG.vm is 1 and CFLG.pe is 0**			
	EFLAG.vif **EFLAG.vip**	**EFLAG.vip & EFLAG.vif & CFLG.pe & PSR.cpl==3 & (CFLG.pvi | (EFLAG.vm & CFLG.vme))**			
all code fetches CS	dpl	ignored		dpl is not 3	**Code Fetch Fault (GPFault(0))**
	d-bit			is not 16-bit	
	type	ignored (can be exec or data)			
		GPFault if data expand down			
	s, p, a-bits	are not 1			
	g-bit/limit	segment limit violation			
data memory references to SS	dpl	dpl!=PSR.cpl			Stack Fault
	d-bit	ignored		is not 16-bit	
	type	ignored		data expand down	
		read and not readable, write and not writeable			
	s, p, a-bits	are not 1			
	g-bit/limit	segment limit violation			
data memory references to DS, ES, FS and GS	dpl	ignored			GPFault(0)
	d-bit	ignored		is not 16-bit	
	type	ignored		data expand down	
		read and not readable, write and not writeable			
	s, p, a-bits	are not 1			
	g-bit/limit	segment limit violation			
data memory references to CS	dpl	ignored			GPFault(0)
	d-bit	ignored		is not 16-bit	
	type	ignored		data expand down	
		rd/wr checks are ignored	rd and not readable, wr and not writeable	rd/wr checks are ignored	
	s, p, a-bits	are not 1			
	g-bit/limit	segment limit violation			
memory references to LDT,GDT, TSS	dpl	ignored			GPFault (Selector/0)[b]
	type	ignored			
	s-bit	is not 0			
	a, d-bits	ignored			
	p-bit	is not 1			
	g-bit/limit	segment limit violation			

[a] Code Fetch Faults are delivered as higher priority GPFault(0).

[b] The GP Fault error code is the selector value if the reference is to GDT or LDT. Otherwise the error code is zero.

6.2.4 IA-32 Application EFLAG Register

The EFLAG (AR24) register is made up of two major components, user arithmetic flags (CF, PF, AF, ZF, SF, OF, and ID) and system control flags (TF, IF, IOPL, NT, RF, VM, AC, VIF, VIP). None of the arithmetic or system flags affect IA-64 instruction execution.

Figure 6-6. IA-32 EFLAG Register (AR24)

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
reserved (set to 0)										id	vip	vif	ac	vm	rf	0	nt	iopl		of	df	if	tf	sf	zf	0	af	0	pf	1	cf
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved (set to 0)																															

The arithmetic flags are used by the IA-32 instruction set to reflect the status of IA-32 operations, control IA-32 string operations, and control branch conditions for IA-32 instructions. These flags are ignored by IA-64 instructions. Flags ID, OF, DF, SF, ZF, AF, PF and CF are defined in the *Intel Architecture Software Developer's Manual.*

Table 6-5. IA-32 EFLAGS Register Fields

EFLAG[a]	Bits	Description
cf	0	IA-32 Carry Flag. See the *Intel Architecture Software Developer's Manual* for details.
	1	Ignored - Writes are ignored, reads return one for IA-32 and IA-64 instructions.
	3,5, 15	Ignored - Writes are ignored, reads return zero for IA-32 and IA-64 instructions. Software should set these bits to zero.
pf	2	IA-32 Parity Flag. See the *Intel Architecture Software Developer's Manual* for details.
af	4	IA-32 Aux Flag. See the *Intel Architecture Software Developer's Manual* for details.
zf	6	IA-32 Zero Flag. See the *Intel Architecture Software Developer's Manual* for details.
sf	7	IA-32 Sign Flag. See the *Intel Architecture Software Developer's Manual* for details.
tf	8	IA-32 System EFLAG Register
if	9	
df	10	IA-32 Direction Flag. See the *Intel Architecture Software Developer's Manual* for details.
of	11	IA-32 Overflow Flag. See the *Intel Architecture Software Developer's Manual* for details.
iopl	13:12	IA-32 System EFLAG Register
nt	14	
rf	16	
vm	17	
ac	18	
vif	19	
vip	20	
id	21	
	63:22	Reserved must be set to zero

[a] On entry into the IA-32 instruction set all bits may be read by subsequent IA-32 instructions, after exit from the IA-32 instruction set these bits represent the results of all prior IA-32 instructions. None of the EFLAG bits alter the behavior of IA-64 instruction set execution.

6.2.5 IA-32 Floating-point Registers

IA-32 floating-point register stack, numeric controls and environment are mapped into the IA-64 floating-point registers FR8 - FR15 and the application register name space as shown in Table 6-6.

Table 6-6. IA-32 Floating-point Register Mappings

IA-64 Reg	IA-32 Reg	Size (bits)	Description
FR8	ST[(TOS + N)==0]	80	IA-32 numeric register stack IA-64 accesses to FR8 - FR15 ignore the IA-32 TOS adjustment IA-32 accesses use the TOS adjustment for a given register N
FR9	ST[(TOS + N)==1]		
FR10	ST[(TOS + N)==2]		
FR11	ST[(TOS + N)==3]		
FR12	ST[(TOS + N)==4]		
FR13	ST[(TOS + N)==5]		
FR14	ST[(TOS + N)==6]		
FR15	ST[(TOS + N)==7]		
FCR (AR21)	FCW, MXCSR	64	IA-32 numeric and Streaming SIMD Extension control register
FSR (AR28)	FSW,FTW, MXCSR	64	IA-32 numeric and Streaming SIMD Extension status and tag word
FIR (AR29)	FOP, FCS, FIP	64	IA-32 numeric instruction pointer
FDR (AR30)	FDS, FEA	48	IA-32 numeric data pointer

6.2.5.1 IA-32 Floating-point Stack

IA-32 floating-point registers are defined as follows:

- IA-32 numeric register stack is mapped to FR8 - FR15, using the Intel 8087 80-bit IEEE floating-point format.
- For IA-32 instruction set references, floating-point registers are logically mapped into FR8 - FR15 based on the IA-32 top-of-stack (TOS) pointer held in FCR.top. FR8 represents a physical register after the TOS adjustment and is not necessarily the top of the logical floating-point register stack.
- For IA-64 instruction set references, the floating-point register numbers are physical and not a function of the numeric TOS pointer, e.g. references to FR8 always return the value in physical register FR8 regardless of the TOS value. IA-64 software cannot necessarily assume that FR8 contains the IA-32 logical register ST(0). It is highly recommended that typically IA-32 calling conventions be used which pass floating-point values through memory.

6.2.5.2 IA-32/IA-64 Special Cases

For IA-32 floating-point instructions, loading a single or double denormal results in a normalized double-extended value placed in the target floating-point register. For IA-64 instructions, loading a single or double denormal results in an un-normalized denormal value placed in the target floating-point register. There are two IA-64 canonical exponent values which indicate single precision and double precision denormals.

When transferring floating-point values from IA-64 to IA-32 instructions, it is highly recommended that typical IA-32 calling conventions be followed which pass floating-point values through the memory stack. If software does pass floating-point values from IA-64 to IA-32 code via the floating-point registers, software must ensure the following:

- IA-64 single or double precision denormals must be converted into a normalized double extended precision value expected by IA-32 instructions. Software can convert IA-64 denormals by multiplying by 1.0 in double extended precision (`fma.sfx fr = fr, f1,f0`). If an illegal single or double precision denormal is encountered in IA-32 floating-point operations, an IA-32 Exception (FPError Invalid Operand) fault is generated.
- Floating-point values must be within the range of the IA-32 80-bit (15-bit exponent) double extended precision format. IA-64 allows 82-bit (17-bit widest range exponent) for intermediate calculations. Software must ensure all IA-64 floating-point register values passed to IA-32 instructions are representable in double extended precision 80-bit format, otherwise processor operation is model specific and undefined. Undefined behavior can include but is not limited to: the generation of an IA-32_Exception (FPError Invalid Operation) fault when used by an IA-32 floating-point instruction, rounding of out-of-range values to zero/denormal/infinity and possible IA-32_Exception (FPError Overflow/Underflow) faults, or float-point register(s) containing out of range values silently converted to QNAN or SNAN (conversion could occur during entry to the IA-32 instruction set or on use by an IA-32 floating-point instruction). Software can ensure all passed floating-point register values are within range by multiplying by 1.0 in double extended precision format (with widest range exponent disabled) by using `fma.sfx fr = fr,f1,f0`.
- IA-64 floating-point NaTVal values must not be propagated into IA-32 floating-point instructions, otherwise processor operation is model specific and undefined. Processors may silently convert floating-point register(s) containing NaTVal to a SNAN (during entry to the IA-32 instruction set or on a consuming IA-32 floating-point instruction). Dependent IA-32 floating-point instructions that directly or indirectly consume a propagated NaTVal register will either propagate the NaTVal indication or generate an IA-32_Exception (FPError Invalid Operand) fault. Whether a processor generates the fault or propagates the NaTVal is model specific. In no case will the processor allow a NaTVal register to be used without either propagating the NaTVal or generating an IA-32_Exception (FPError Invalid Operand) fault. Note: it is not possible for IA-32 code to read a NaTVal from a memory location with an IA-32 floating-point load instruction, since a NatVal can not be expressed by a 80-bit double extended precision number.

It is highly recommended that floating-point values be passed on the memory stack per typical IA-32 calling conventions to avoid numeric problems with NatVal and IA-64 denormals.

6.2.5.3 IA-32 Floating-point Control Registers

FPSR controls IA-64 floating-point instructions control and status bits. FPSR does not control IA-32 floating-point instructions or reflect the status of IA-32 floating-point instructions. IA-32 floating-point and Streaming SIMD Extension instructions have separate control and status registers, namely FCR (floating-point control register) and FSR (floating-point status register).

FCR contains the IA-32 FCW bits and all Streaming SIMD Extension control bits as shown in Figure 6-7.

FSR contains the IA-32 floating-point status flags FSW, FTW, and Streaming SIMD Extension status fields as shown in Figure 6-8. The Tag fields indicate whether the corresponding IA-32 logical floating-point register is empty. Tag encodings for zero and special conditions such as Nan, Infinity or Denormal of each IA-32 logical floating-point register are not supported. However, IA-32 instruction set reads of FTW compute the additional special conditions of each IA-32 floating-point register. IA-64 code can issue a floating-point classify operation to determine the disposition of each IA-32 floating-point register.

FCR and FSR collectively hold all IA-32 floating-point control, status and tag information. IA-32 instructions that are updated and controlled by MXSCR, FCW, FSW and FTAG effectively update FSR and are controlled by FSR. IA-32 reads/writes of MXCSR, FSW, FCW and FTW return the same information as IA-64 reads/writes of FSR and FCR.

Software must ensure that FCR and FSR are properly loaded for IA-32 numeric execution before entering the IA-32 instruction set.

Figure 6-7. IA-32 Floating-point Control Register (FCR)

																			IA-32 FCW{12:0}												
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
reserved (set to 0)																			IC	RC		PC		0	1	PM	UM	OM	ZM	DM	IM

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved (set to 0)																FZ	RC		PM	UM	OM	ZM	DM	IM	rv	ignored					
																IA-32 MXCSR (control)															

Figure 6-8. IA-32 Floating-point Status Register (FSR)

IA-32 FTW{15:0}																IA-32 FSW{15:0}															
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
0	TG7	0	TG6	0	TG5	0	TG4	0	TG 3	0	TG2	0	TG1	0	TG0	B	C3	TOP			C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved (set to 0)																ignored									rv	PE	UE	OE	ZE	DE	IE
																IA-32 MXCSR (status)															

Table 6-7. IA-32 Floating-point Control Register Mapping (FCR)

IA-32 State	IA-64 State	Bits	IA-32 Usage	IA-64 Usage
FCW, MXCSR state in the FCR Register				
FCW.im	FCR.im	0	Invalid operation Mask	None of these IA-32 numeric and Streaming SIMD Extension control bits affect the execution of IA-64 floating-point instructions. See *Intel Architecture Software Developer's Manual* for details on each field.
FCW.dm	FCR.dm	1	Denormalized operand Mask	
FCW.zm	FCR.zm	2	Zero divide Mask	
FCW.om	FCR.om	3	Overflow Mask	
FCW.um	FCR.um	4	Underflow Mask	
FCW.pm	FCR.pm	5	Precision Mask	
ignored		6	Ignored - Writes are ignored, reads return 1.	
ignored		7, 32:37	Ignored - Writes are ignored, reads return 0	
reserved		13:31,38 ,48:63	Reserved	
FCW.pc	FCR.pc	8:9	Precision Control (00- single, 10- double, 11- extended)	
FCW.rc	FCR.rc	10:11	Rounding (00-even, 01-down, 10-up, 11-truncate)	
FCW.ic	FCR.ic	12	(Infinity Control) - Ignored by All IA-64 processors, provided for compatibility with IA-32 processors.	
MXCSR.im	FCR.im	39	Streaming SIMD Extension Invalid operation Mask	
MXCSR.dm	FCR.dm	40	Streaming SIMD Extension Denormalize operand Mask	
MXCSR.zm	FCR.zm	41	Streaming SIMD Extension Zero divide Mask	
MXCSR.om	FCR.om	42	Streaming SIMD Extension Overflow Mask	
MXCSR.um	FCR.um	43	Streaming SIMD Extension Underflow Mask	
MXCSR.pm	FCR.pm	44	Streaming SIMD Extension Precision Mask	
MXCSR.rc	FCR.rc	45:46	Streaming SIMD Extension Rounding (00-even,01-down, 10-up, 11-truncate)	
MXCSR.fz	FCR.fz	47	Streaming SIMD Extension Flush to Zero	

Table 6-8. IA-32 Floating-point Status Register Mapping (FSR)

IA-32 State	IA-64 State	Bits	IA-32 Usage	IA-64 Usage
FSW, FTW, MXCSR state in the FSR Register				
FSW.ie	FSR.ie	0	Invalid operation Exception	None of these bits reflect the status of IA-64 floating-point execution. See *Intel Architecture Software Developer's Manual* for IA-32 numeric flag details
FSW.de	FSR.de	1	Denormalized operand Exception	
FSW.ze	FSR.ze	2	Zero divide Exception	
FSW.oe	FSR.oe	3	Overflow Exception	
FSW.ue	FSR.ue	4	Underflow Exception	
FSW.pe	FSR.pe	5	Precision Exception	
FSW.sf	FSR.sf	6	Stack Fault	
FSW.es	FSR.es[a]	7	Error Summary	
FSW.c3:0	FSR.c3:0	8:10,14	Numeric Condition codes	
FSW.top	FSR.top	11:13	Top of IA-32 numeric stack	
FSW.b	FSR.b	15	IA-32 FPU Busy always equals state of FSW.ES	
FTW	FSR.tg {7:0}[b]	16,18,20,22,24,26,28,30	Numeric Tags 0-NotEmpty, 1-Empty[c]	
zeros		17,19,21,23,25,27,29,31, 39:47	Ignored - Writes are ignored, reads return zero	
MXCSR.ie	FSR.ie	32	Streaming SIMD Extension Invalid operation Exception	Does not reflect the status of IA-64 floating-point execution. See IA-32 *Intel Architecture Software Developer's Manual* for details.
MXCSR.de	FSR.de	33	Streaming SIMD Extension Denormalized operand Exception	
MXCSR.ze	FSR.ze	34	Streaming SIMD Extension Zero divide Exception	
MXCSR.oe	FSR.oe	35	Streaming SIMD Extension Overflow Exception	
MXCSR.ue	FSR.ue	36	Streaming SIMD Extension Underflow Exception	
MXCSR.pe	FSR.pe	37	Streaming SIMD Extension Precision Exception	
reserved		38, 48:63	Reserved	
ignored		39:47	Ignored - Writes are ignored, reads return zero	

a. Exception Summary bit, see Section 6.2.5.4 for details.

b. Tag encodings indicate whether each IA-32 numeric register contains an zero, NaN, Infinity or Denormal are not supported by IA-64 reads of FSR. IA-32 instruction set reads of the FTW field do return zero, Nan, Infinity and Denormal classifications.

c. All MMX™ technology instructions set all Numeric Tags to 0 = NotEmpty. However, MMX technology instruction EMMS sets all Numeric Tags to 1 = Empty.

6.2.5.4 IA-32 Floating-point Environment

To support the Intel 8087 delayed numeric exception model, FSR, FDR and FIR contain pending information related to the numeric exception. FDR contains the operand's effective address and segment selector. FIR contains the numeric instruction's effective address, code segment selector, and opcode bits. FSR summaries the type of numeric exception in the IE, DE, ZE, OE, UE, PE, SF and ES-bits. The ES-bit summarizes the IA-32 floating-point exception status as follows:

- When FSR.es is read by IA-64 code, the value returned is a summary of any unmasked pending exceptions contained in the FSR, IE, DE, ZE, OE, UE, PE, and SF bits. Note: reads of the ES-bit do not necessarily return the last value written if the ES-bit is inconsistent with the other pending exception bits in FSR.
- When FSR.es is set to a 1 by IA-64 code, delayed IA-32 numeric exceptions are generated on the next IA-32 floating-point instruction, regardless of numeric exception information written into FSR bits; IE, DE, ZE, OE, UE, PE, and SF.
- When FSR.es is written with inconsistent state with respect to the FSR bits (IE, DE, ZE, OE,PE and SF), subsequent numeric exceptions may report inconsistent floating-point status bits.

FSR, FDR, and FIR must be preserved across a context switch to generate and accurately report numeric exceptions.

Figure 6-9. Floating-point Data Register (FDR)

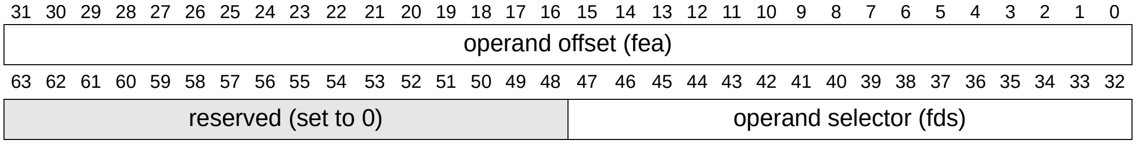
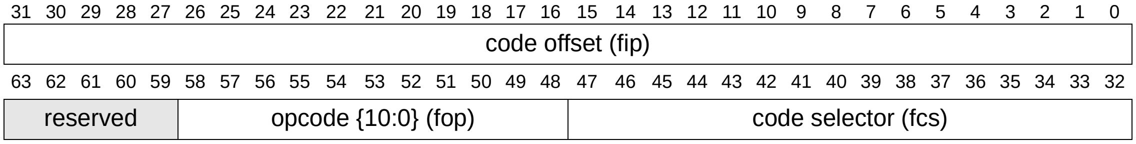


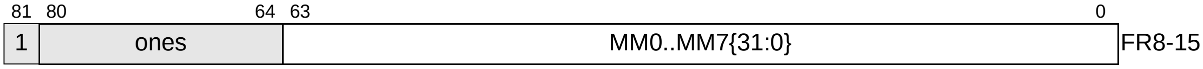
Figure 6-10. Floating-point Instruction Register (FIR)



6.2.6 IA-32 MMX™ Technology Registers

The eight IA-32 MMX technology registers are mapped on the eight IA-64 floating registers FR8 - FR15. Where MM0 is mapped to FR8 and MM7 is mapped to FR15. The MMX technology register mapping for the IA-32 floating-point stack view is dependent on the floating-point IA-32 Top-of-Stack value.

Figure 6-11. IA-32 MMX™ Technology Registers (MM0 to MM7)



- When a value is written to an MMX technology register using an IA-32 MMX technology instruction:
  - — The exponent field of the corresponding floating-point register (bits 80-64) and the sign bit (bit 81) are set to all ones.
  - — The mantissa (bits 63-0) is set to the MMX technology data value.

- When a value is read from a MMX technology register by an IA-32 MMX technology instruction:
  - — The exponent field of the corresponding floating-point register (bits 80-64) and its sign bit (bit 81) are ignored, including any NaTVal encodings.

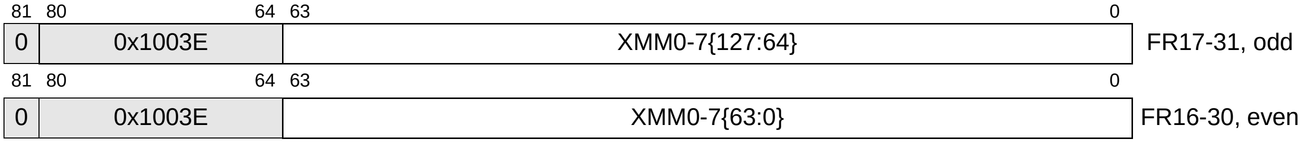
As a result of this mapping, the mantissa of a floating-point value written by either IA-32 or IA-64 floating-point instructions will also appear in an IA-32 MMX technology register. An IA-32 MMX technology register will also appear in one of the eight mapped floating-point register's mantissa field.

To avoid performance degradation, software programmers are strongly recommended not to intermix IA-32 floating and IA-32 MMX technology instructions. See the *Intel Architecture Software Developer's Manual* for MMX technology coding guidelines for details.

6.2.7 IA-32 Streaming SIMD Extension Registers

The eight 128-bit IA-32 Streaming SIMD Extension registers (XMM0-7) are mapped on sixteen physical IA-64 floating register pairs FR16 - FR31. The low order 64-bits of XMM0 are mapped to FR16{63:0}, and the high order 64-bits of XMM0 are mapped to FR17{63:0}.

Figure 6-12. Streaming SIMD Extension registers (XMM0-XMM7)

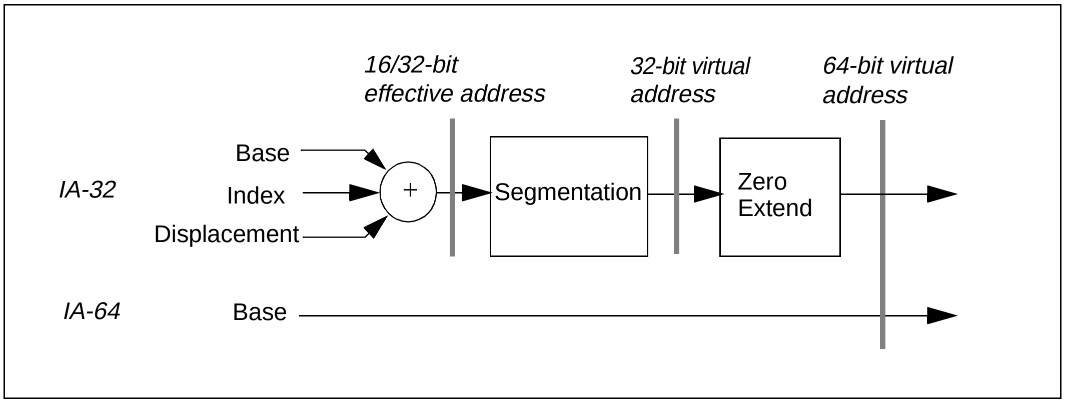


- When a value is written to an Streaming SIMD Extension register using IA-32 Streaming SIMD Extension instructions:
  - — The exponent field of the corresponding IA-64 floating-point register (bits 80-64) is set to 0x1003E and the sign bit (bit 81) is set to 0.
  - — The mantissa (bits 63-0) is set to the XMM data value bits{63:0} for even registers and bits{127:64} for odd registers.
- When a Streaming SIMD Extension register is read using IA-32 Streaming SIMD Extension instructions:
  - — The exponent field of the corresponding IA-64 floating-point register (bits 80-64) and the sign bit (bit 81) are ignored, including any NaTVal encodings.

6.3 Memory Model Overview

Virtual addresses within either the IA-64 or IA-32 instruction set are defined to address the same physical memory location. IA-64 instructions directly generate 64-bit virtual addresses. IA-32 instructions generate 16 or 32-bit effective addresses that are then converted into 32-bit virtual addresses by IA-32 segmentation. 32-bit virtual addresses are then converted into 64-bit virtual addresses by zero extending to 64-bits. Zero extension places all IA-32 memory references in the lower 4G-bytes of the 64-bit virtual address space. Virtual addresses generated by either instruction set are then translated into physical addresses using IA-64 memory management mechanisms.

Figure 6-13. Memory Addressing Model



6.3.1 Memory Endianess

Memory integer and floating-point (IEEE) data types are binary compatible between the IA-32 and IA-64 instruction sets. IA-64 applications and operating systems that interact with IA-32 code should use “little-endian” accesses to ensure that memory formats are the same. All IA-32 instruction data and instruction memory references are forced to “little-endian”.

6.3.2 IA-32 Segmentation

Segmentation is not used for IA-64 instruction set memory references. Segmentation is performed on IA-32 instruction set memory references based on the state of EFLAG.vm and CFLG.pe. Either Real Mode, VM86, or Protected Mode segmentation rules are followed as defined in the *Intel Architecture Software Developer's Manual*, specifically:

- **IA-32 Data 16/32-bit Effective Addresses:** 16 or 32-bit effective addresses are generated, based on CSD.d, SSD.b and prefix overrides, by the addition of a base register, scaled index register and 16/32-bit displacement value. Starting effective addresses (first byte of multi-byte operands) larger than 16 or 32 bits are truncated to 16 or 32-bits. Ending (last byte of multi-byte operands) 16-bit effective addresses can extend above the 64K byte boundary, however, ending 32-bit effective addresses are truncated to 32-bits and do not extend above the 4G-byte effective address boundary. Refer to the *Intel Architecture Software Developer's Manual* for complete details on wrap conditions.
- **IA-32 Code 16/32-bit Effective Addresses:** 16 or 32-bit EIP, based on CSD.d, is used as the effective address. Starting EIP values (first byte of multi-byte instruction) larger than 16 or 32 bits are truncated to 16 or 32-bits. Ending (last byte of multi-byte instruction) 16-bit effective addresses can extend above the 64K byte boundary, however, ending 32-bit EIP values are truncated to 32-bits and do not extend above the 4G-byte effective address boundary.
- **IA-32 32-bit Virtual Address Generation:** The resultant 16 or 32-bit effective address is mapped into the 32-bit virtual address space by the addition of a segment base. Full segment protection and limit checks are verified as specified by the *Intel Architecture Software Developer's Manual* and additional checks as specified in this section. Starting 32-bit virtual addresses are truncated to 32-bits after the addition of the segment base. Ending virtual address (last byte of a multiple byte operand or instruction) is truncated (wrapped) at the 4G-byte virtual boundary
- **IA-32 64-bit Address Generation:** The resultant 32-bit virtual address is converted into a 64-bit virtual address by zero extending to 64-bits, this places all IA-32 instruction set memory references within the first 4G-bytes of the 64-bit virtual address space.

If IA-32 code is utilizing a flat segmented model (segment bases are set to zero) then IA-32 and IA-64 code can freely exchange pointers after a pointer has been zero extended to 64-bits. For segmented IA-32 code, effective address pointers must be first transformed into a virtual address before they are shared with IA-64 code.

6.3.3 Self Modifying Code

While operating in the IA-32 instruction set, self modifying code and instruction cache coherency (coherency with respect to the local processor's data cache) is supported for all IA-32 programs. Self modifying code detection is directly supported at the same level of compatibility as the Pentium processor. Software must insert an IA-32 branch instruction between the store operation and the instruction modified for the updated instruction bytes to be recognized.

When switching from the IA-64 to the IA-32 instruction set, and while executing IA-64 instructions, self modifying code and instruction cache coherency are not directly supported by the processor hardware. Specifically, if a modification is made to IA-32 instructions by IA-64 instructions, IA-64 code must explicitly synchronize the instruction caches with the code sequence defined in Section 4.4.6.2, "Memory Consistency" on page 4-23. Otherwise the modification may or may not be observed by subsequent IA-32 instructions.

When switching from the IA-32 to the IA-64 instruction sets, modification of the local instruction cache contents by IA-32 instructions is detected by the processor hardware. The processor ensures that the instruction cache is made coherent with respect to the modification and all subsequent IA-64 instruction fetches see the modification.

6.4 IA-32 Usage of IA-64 Registers

This section lists software considerations for the IA-64 general and floating-point registers, and the ALAT when interacting with IA-32 code.

6.4.1 IA-64 Register Stack Engine

Software must ensure that all dirty registers in the register stack have been flushed to the backing store using a `flushrs` instruction before starting IA-32 execution either via the `br.ia` or `rfi`. Any dirty registers left in the current and prior register stack frames will be modified. For details on register stack, refer to Section 4.1, "Register Stack" on page 4-1.

Once IA-32 instruction set execution is entered, the RSE is effectively disabled, regardless of any RSE control register enabling conditions.

After exiting the IA-32 instruction set due to a `jmpe` instruction or interruption, all stacked registers are marked as invalid and the number of clean registers is set to zero.

6.4.2 IA-64 ALAT

IA-32 instruction set execution leaves the contents of the ALAT undefined. Software cannot rely on ALAT values being preserved across an instruction set transition. On entry to IA-32 code, existing entries in the ALAT are ignored. For details on ALAT, refer to Section 4.4.5.2, "Data Speculation and Instructions" on page 4-17.

6.4.3 IA-64 NaT/NaTVal Response for IA-32 Instructions

If IA-64 code sets a NaT condition in the integer registers or a NaTVal condition in a floating-point register, MMX technology register or Streaming SIMD Extension register before switching to the IA-32 instruction set the following conditions can arise:

- IA-32 dependent instructions that directly or indirectly consume a propagated NaT register will either propagate the NaT indication or generate a NaT Register Consumption abort. Whether a processor generates a NaT Register Consumption abort or propagates the NaT is model specific. NaT Register Consumption aborts encountered during IA-32 execution may terminate IA-32 instructions in the middle of execution with some architectural state already modified. In no case will the processor allow a NaTed input register to be used without either propagating the NaT or generating a NaT Register Consumption fault.
- IA-64 floating-point NaTVal values must not be propagated into IA-32 floating-point instructions, otherwise processor operation is model specific and undefined. Processors may convert floating-point register(s) containing NaTVal to a SNAN (during entry to the IA-32 instruction set or on a consuming IA-32 floating-point instruction). Dependent IA-32 floating-point instructions that directly or indirectly consume a propagated NaTVal register will either propagate the NaTVal indication or generate an IA-32_Exception (FPError Invalid Operand) fault. Whether a processor generates the fault or propagates the NaTVal is model specific. In no case will the processor allow a NaTVal register to be used without either propagating the NaTVal or generating an IA-32_Exception (FPError Invalid Operand) fault. Note: it is not possible for IA-32 code to read a NaTVal from a memory location with an IA-32 floating-point load instruction since a NatVal can not be expressed by a 80-bit double extended precision number. It is highly recommended that floating-point values be passed on the memory stack per typical IA-32 calling conventions to avoid problems with NatVal and IA-64 denormals.
- IA-32 Streaming SIMD Extension instructions that directly or indirectly consume a register containing a NaTVal encoding, will ignore the NaTVal encoding and interpret the register’s mantissa field as a legal data value.
- IA-32 MMX technology instructions that directly or indirectly consume a register containing a NaTVal encoding, will ignore the NaTVal encoding and interpret the register’s mantissa field as a legal data value.

Software should not rely on the behavior of NaT or NaTVal during IA-32 instruction execution, or propagate NaT or NaTVal into IA-32 instructions.

JMPE—Jump to IA-64 Instruction Set

Opcode	Instruction	Description
0F 00 /6	JMPE *r/m16*	Jump to IA-64, indirect address specified by r/m16
0F 00 /6	JMPE *r/m32*	Jump to IA-64, indirect address specified by r/m32
0F B8	JMPE *disp16*	Jump to IA-64, absolute address specified by addr16
0F B8	JMPE *disp32*	Jump to IA-64, absolute address specified by addr32

Description

This instruction is available only on IA-64 processors in the IA-64 System Environment.

JMPE switches the processor to the IA-64 instruction set and starts execution at the specified target address There are two forms; an indirect form, r/m*r16/32*, and an unsigned absolute form, *disp16/32*. Both 16 and 32-bit formats are supported.

The absolute form computes the 16-byte aligned 64-bit virtual target address in the IA-64 instruction set by adding the unsigned 16 or 32-bit displacement to the current CS base (*IP{31:0} = disp16/32 + CSD.base)*. The indirect form specifies the virtual IA-64 target address by the contents of a register or memory location (*IP{31:0} = [r/m16/32] + CSD.base)*.

GR[1] is loaded with the next sequential instruction address following `JMPE`.

`JMPE` performs a FWAIT operation, any pending IA-32 unmasked floating-point exceptions are reported as faults on the `JMPE` instruction.

`JMPE` does not perform a memory fence or serialization operation.

Successful execution of `JMPE` clears EFLAG.rf to zero.

If the IA-64 register stack engine is enabled for eager execution, the register stack engine may immediately start loading registers when the processor enters the IA-64 instruction set.

IA-64 Instruction Reference 7

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

7.1 Instruction Page Conventions

The instruction pages are divided into multiple sections as listed in Table 7-1. The first four sections are present on all instruction pages. The last three sections are present only when necessary. Table 7-2 lists the font conventions which are used by the instruction pages.

Table 7-1. 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

Table 7-2. 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 7-3. 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 7-3. 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 standard C is listed in Table 7-4.

Table 7-3. Register File Notation

Register File	C Notation	Assembly Mnemonic	Indirect Access
Application registers	AR	ar	
Branch registers	BR	b	
CPU identification registers	CPUID	cpuid	Y
Floating-point registers	FR	f	
General registers	GR	r	
Performance monitor data registers	PMD	pmd	Y
Predicate registers	PR	p	

Table 7-4. 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.

7.2 Instruction Descriptions

The remainder of this chapter provides a description of IA-64 instruction.

Add

Format:

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

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

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

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

Operation:

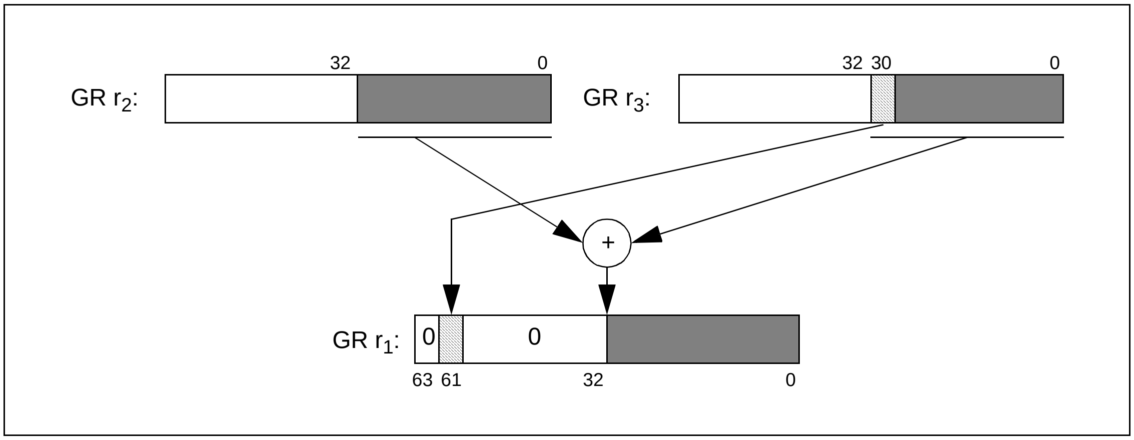
```
if (PR[qp]) {
    check_target_register(r1);

    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;
}
```

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 7-1. 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;
}
```

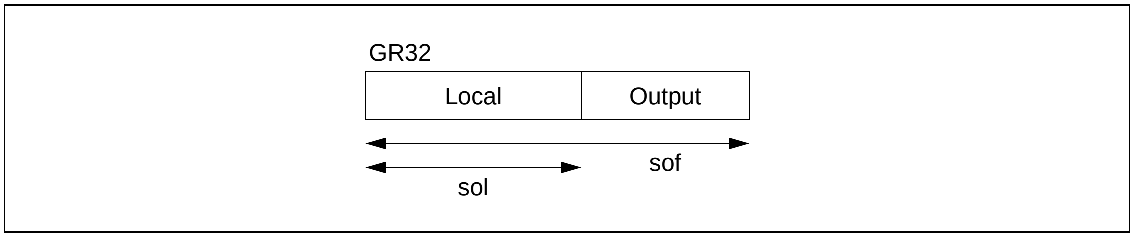
Allocate Stack Frame

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

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

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

Figure 7-2. Stack Frame



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

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

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

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

Operation:

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

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

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

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;
}
```

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;
}
```

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 calculation 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 7-5. Branch Types

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

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

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

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

- **ia:** The branch is taken unconditionally, if it is not intercepted by the OS. The effect of the branch is to invoke the IA-32 instruction set (by setting PSR.is to 1) and begin processing IA-32 instructions at the virtual linear target address contained in BR b_2{31:0}. If the qualifying predicate is not PR 0, an Illegal Operation fault is raised.

  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 instruction set transitions are not disabled. 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. EFLAG.rf is not cleared until the target IA-32 instruction successfully completes.

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

  IA-32 instruction set execution leaves the contents of the ALAT undefined. Software can not rely on ALAT values being preserved across an instruction set transition. 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. All registers left in the current register stack frame are left undefined during IA-32 instruction set execution. 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 proceeding the `br.ia` instruction. To enhance the performance of the instruction set transition, software can start the IA-64 register stack flush in parallel with starting the IA-32 instruction set by 1) ensuring `flushrs` is exactly one instruction group before the `br.ia`, and 2) `br.ia` is in the first B-slot. `br.ia` should always be executed in the first B-slot with a hint of "static-taken" (default), otherwise processor performance will be degraded.

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. 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).

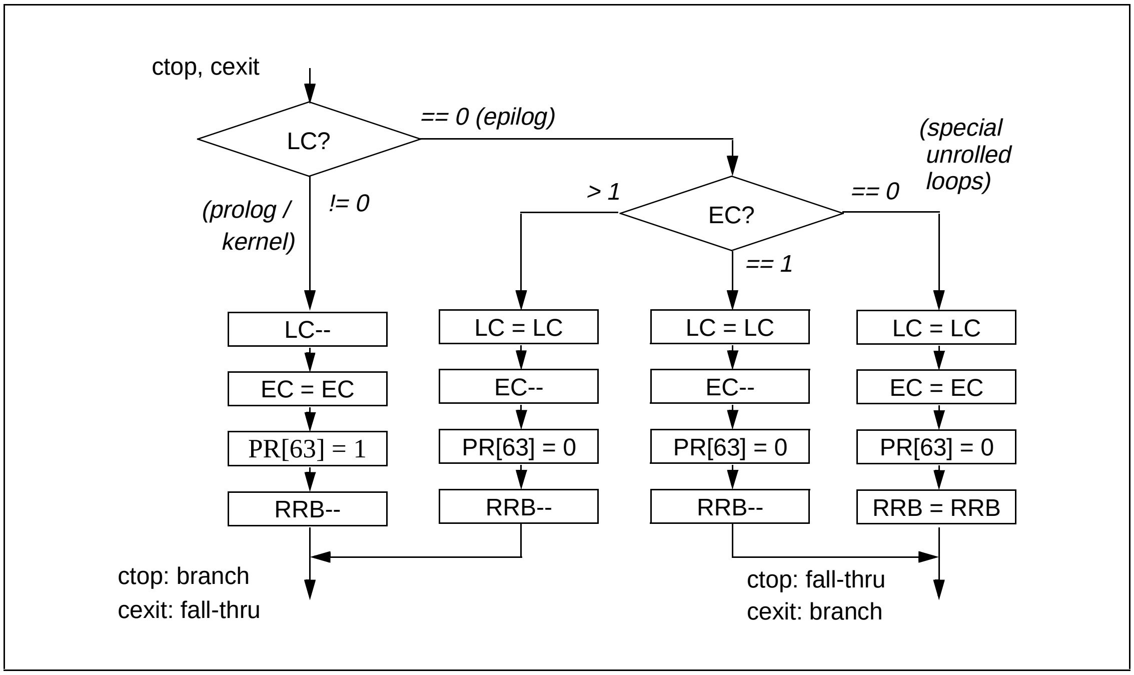
These branch types are of two categories: top and exit. The top types (ctop and wtop) are used when the loop decision is located at the bottom of the loop body and therefore a taken branch will continue the loop while a fall through branch will exit the loop. The exit types (cexit and wexit) are used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

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 7-3.

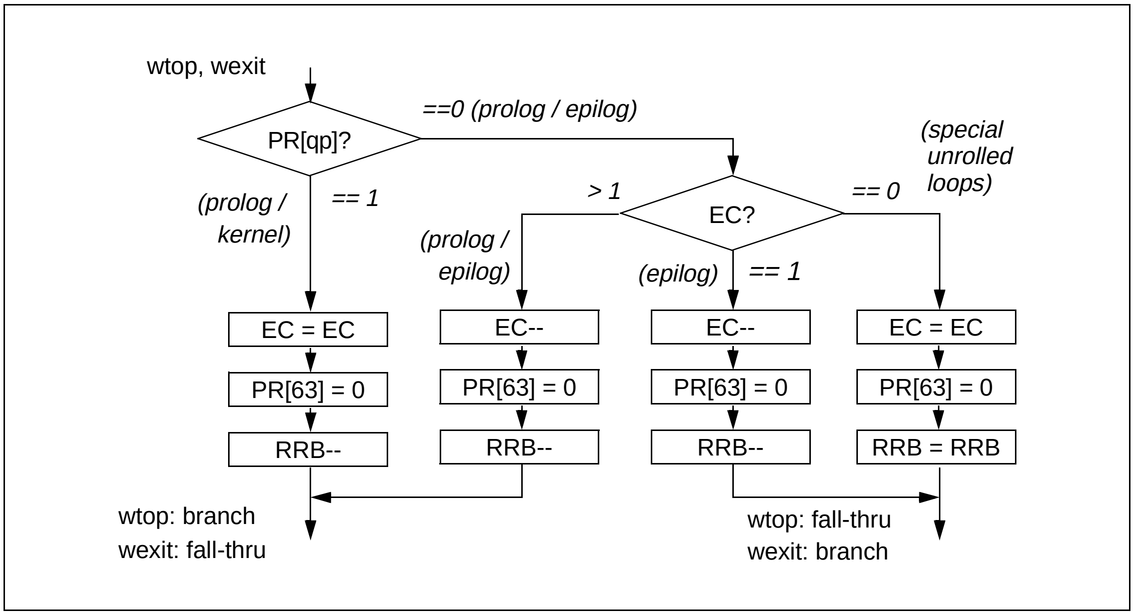
Figure 7-3. Operation of br.ctop and br.cexit



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 7-4.

Figure 7-4. 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 7-6. Sequential Prefetch hints are shown in Table 7-7. Branch Cache Deallocation hints are shown in Table 7-8.

Table 7-6. Branch Whether Hint

***bwh* Completer**	**Branch Whether Hint**
spnt	Static Not-Taken
sptk	Static Taken

Table 7-6. Branch Whether Hint

bwh Completer	Branch Whether Hint
dpnt	Dynamic Not-Taken
dptk	Dynamic Taken

Table 7-7. Sequential Prefetch Hint

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

Table 7-8. Branch Cache Deallocation Hint

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

Operation:

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

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

lower_priv_transition = 0;

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

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

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

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

    case ‘ret’:                              // return restores stack frame
        tmp_taken = PR[qp];
        if (tmp_taken) {
```

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

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

case ‘ia’:                                // switch to IA mode
   tmp_taken = 1;
   if (qp != 0)
      illegal_operation_fault();
   if (AR[BSPSTORE] != AR[BSP])
      illegal_operation_fault();
   if (PSR.di)
      disabled_instruction_set_transition_fault();
   PSR.is = 1;                            // set IA-32 Instruction Set Mode
   CFM.sof = 0;                           //force current stack frame
   CFM.sol = 0;                           //to zero
   CFM.sor = 0;
   CFM.rrb.gr = 0;
   CFM.rrb.fr = 0;
   CFM.rrb.pr = 0;
   rse_invalidate_non_current_regs();

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

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

case ‘ctop’:
case ‘cexit’:                             // SW pipelined counted loop
   if (slot != 2)
      illegal_operation_fault();
   if (btype == ‘ctop’) tmp_taken =  ((AR[LC] != 0) || (AR[EC] u> 1));
   if (btype == ‘cexit’)tmp_taken = !((AR[LC] != 0) || (AR[EC] u> 1));
   if (AR[LC] != 0) {
      AR[LC]--;
      AR[EC] = AR[EC];
```

```
         PR[63] = 1;
         rotate_regs();
      } else if (AR[EC] != 0) {
         AR[LC] = AR[LC];
         AR[EC]--;
         PR[63] = 0;
         rotate_regs();
      } else {
         AR[LC] = AR[LC];
         AR[EC] = AR[EC];
         PR[63] = 0;
         CFM.rrb.gr = CFM.rrb.gr;
         CFM.rrb.fr = CFM.rrb.fr;
         CFM.rrb.pr = CFM.rrb.pr;
      }
      break;

   case ‘wtop’:
   case ‘wexit’:                             // SW pipelined while loop
      if (slot != 2)
         illegal_operation_fault();
      if (btype == ‘wtop’) tmp_taken =  (PR[qp] || (AR[EC] u> 1));
      if (btype == ‘wexit’)tmp_taken = !(PR[qp] || (AR[EC] u> 1));
      if (PR[qp]) {
         AR[EC] = AR[EC];
         PR[63] = 0;
         rotate_regs();
      } else if (AR[EC] != 0) {
         AR[EC]--;
         PR[63] = 0;
         rotate_regs();
      } else {
         AR[EC] = AR[EC];
         PR[63] = 0;
         CFM.rrb.gr = CFM.rrb.gr;
         CFM.rrb.fr = CFM.rrb.fr;
         CFM.rrb.pr = CFM.rrb.pr;
      }
      break;
}
if (tmp_taken) {
   taken_branch = 1;
   IP = tmp_IP;                              // set the new value for IP
   if ((PSR.it && unimplemented_virtual_address(tmp_IP))
       || (!PSR.it && unimplemented_physical_address(tmp_IP)))
      unimplemented_instruction_address_trap(lower_priv_transition,tmp_IP);
   if (lower_priv_transition && PSR.lp)
      lower_privilege_transfer_trap();
   if (PSR.tb)
      taken_branch_trap();
}
```

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}*.

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($imm_{62}$, 21);
    else // i_unit_form || m_unit_form || f_unit_form
        immediate = zero_ext($imm_{21}$, 21);

    break_instruction_fault(immediate);
}
```

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.

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

The 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 7-9).

Table 7-9. 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($f_2$, 0, 0, 0)))
            disabled_fp_register_fault(tmp_isrcode, 0);
        check_type = gr_form ? CHKS_GENERAL : CHKS_FLOAT;
        fail = (gr_form && GR[$r_2$].nat) || (fr_form && FR[$f_2$] == NATVAL);
    } else {                                                          // data_form
        reg_type = gr_form ? GENERAL : FLOAT;
        alat_index = gr_form ? $r_1$ : (data_form ? $f_1$ : $f_2$);

        check_type = gr_form ? CHKA_GENERAL : CHKA_FLOAT;
        fail = !alat_cmp(reg_type, alat_index);
    }
    if (fail) {
            taken_branch = 1;
            IP = IP + sign_ext(($imm_{21}$ << 4), 25);
            if ((PSR.it && unimplemented_virtual_address(IP))
                || (!PSR.it && unimplemented_physical_address(IP)))
                unimplemented_instruction_address_trap(0, IP);
            if (PSR.tb)
                taken_branch_trap();
    }
    if (!fail && data_form && (aclr == ‘clr’))
        alat_inval_single_entry(reg_type, alat_index);
}
```

Clear RRB

Format:	clrrb	all_form	B8
	clrrb.pr	pred_form	B8

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

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

This instruction cannot be predicated.

Operation:

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

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

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 7-10. A blank entry indicates the predicate target is left unchanged.

Table 7-10. 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 set, 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 7-11 (for normal and unc type compares), and Table 7-12 (for parallel type compares).

Table 7-11. 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 7-12. 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[$r_2$].nat : 0) || GR[$r_3$].nat;
    if (register_form)
        tmp_src = GR[$r_2$];
    else if (imm8_form)
        tmp_src = sign_ext($imm_8$, 8);
    else // parallel_inequality_form
        tmp_src = 0;

    if      (crel == 'eq')  tmp_rel = tmp_src == GR[$r_3$];
    else if (crel == 'ne')  tmp_rel = tmp_src != GR[$r_3$];
    else if (crel == 'lt')  tmp_rel = lesser_signed(tmp_src,  GR[$r_3$]);
    else if (crel == 'le')  tmp_rel = lesser_equal_signed(tmp_src, GR[$r_3$]);
    else if (crel == 'gt')  tmp_rel = greater_signed(tmp_src,  GR[$r_3$]);
    else if (crel == 'ge')  tmp_rel = greater_equal_signed(tmp_src, GR[$r_3$]);
    else if (crel == 'ltu') tmp_rel = lesser(tmp_src, GR[$r_3$]);
    else if (crel == 'leu') tmp_rel = lesser_equal(tmp_src, GR[$r_3$]);
    else if (crel == 'gtu') tmp_rel = greater(tmp_src, GR[$r_3$]);
    else                    tmp_rel = greater_equal(tmp_src, GR[$r_3$]);     // 'geu'

    switch (ctype) {
        case 'and':                                            // and-type compare
            if (tmp_nat || !tmp_rel) {
                PR[$p_1$] = 0;
                PR[$p_2$] = 0;
            }
            break;
        case 'or':                                              // or-type compare
            if (!tmp_nat && tmp_rel) {
                PR[$p_1$] = 1;
                PR[$p_2$] = 1;
            }
            break;
        case 'or.andcm':                                  // or.andcm-type compare
            if (!tmp_nat && tmp_rel) {
                PR[$p_1$] = 1;
                PR[$p_2$] = 0;
            }
            break;
        case 'unc':                                            // unc-type compare
        default:                                                 // normal compare
            if (tmp_nat) {
                PR[$p_1$] = 0;
                PR[$p_2$] = 0;
            } else {
                PR[$p_1$] = tmp_rel;
                PR[$p_2$] = !tmp_rel;
            }
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[$p_1$] = 0;
        PR[$p_2$] = 0;
    }
}
```

Compare Word

Format:

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

Description: The least significant 32 bits from each of two source operands are compared for one of ten relations specified by *crel*. This produces a boolean result which is 1 if the comparison condition is true, and 0 otherwise. This result is written to the two predicate register destinations, p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 7-10 on page 7-19.

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 7-12 on page 7-20.

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.

Of the ten relations, not all are directly implemented in hardware. Some are actually pseudo-ops. See the Compare instruction and Table 7-11 and Table 7-12 on page 7-20. The range for immediates is given below.

Table 7-13. 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;
    }
}
```

Compare And Exchange

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

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

The values of the *sz* completer are given in Table 7-14. The *sem* completer specifies the type of semaphore operation. These operations are described in Table 7-15 "Compare and Exchange Semaphore Types".

Table 7-14. Memory Compare and Exchange Size

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

Table 7-15. Compare and Exchange Semaphore Types

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

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

The memory read and write are guaranteed to be atomic.

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

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 7-28 on page 7-105. Locality hints do not affect program functionality and may be ignored by the implementation. See "Memory Hierarchy Control and Consistency" on page 4-20 for details.

Operation:

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

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

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

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

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

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

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

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. 7-16 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 7-16. 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;
}
```

Deposit

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

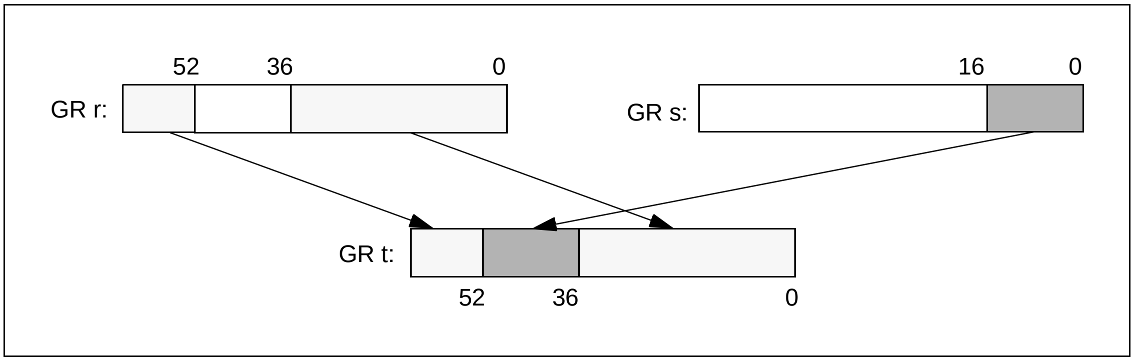
Description: In the merge_form, a right justified bit field taken from the first source operand is deposited into the value in GR r_3 at an arbitrary bit position and the result is placed in GR r_1. In the register_form the first source operand is GR r_2; and in the imm_form it is the sign-extended value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

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

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

The operation of `dep t = s, r, 36, 16` is illustrated in Figure 7-5.

Figure 7-5. Deposit Example



Operation:

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

    if (imm_form) {
        tmp_src = (merge_form ? sign_ext($imm_1$,1) : sign_ext($imm_8$, 8));
        tmp_nat = merge_form ? GR[$r_3$].nat : 0;
        tmp_len = $len_6$ ;
    } else {                                                     // register_form
        tmp_src = GR[$r_2$];
        tmp_nat = (merge_form ? GR[$r_3$].nat : 0) || GR[$r_2$].nat;
        tmp_len = merge_form ? $len_4$ : $len_6$ ;
    }
    if ($pos_6$ + tmp_len u> 64)
        tmp_len = 64 - $pos_6$;

    if (merge_form)
        GR[$r_1$] = GR[$r_3$];
    else // zero_form
        GR[$r_1$] = 0;

    GR[$r_1$]{($pos_6$ + tmp_len - 1):$pos_6$} = tmp_src{(tmp_len - 1):0};
    GR[$r_1$].nat = tmp_nat;
}
```

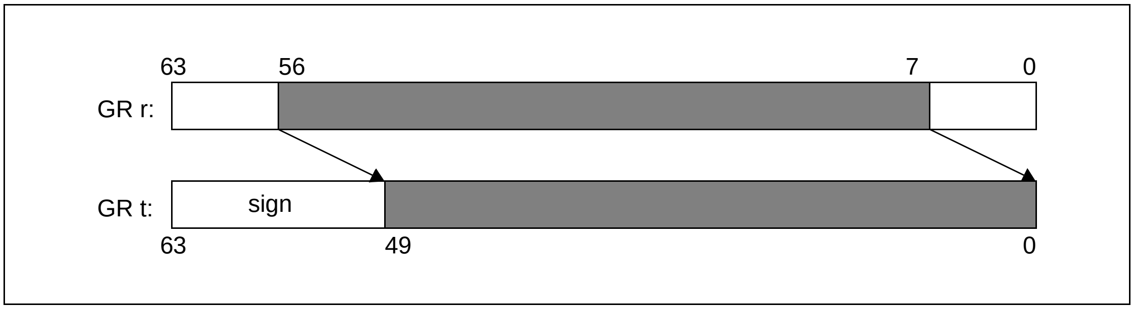
Extract

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

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

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

Figure 7-6. Extract Example



Operation:

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

    tmp_len = len6;

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

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

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

Floating-Point Absolute Value

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

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

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

Operation: `See “Floating-Point Merge” on page 7-50.`

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 7-17. The mnemonic values for *sf* are given in Table 7-18. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Table 7-17. Specified *pc* Mnemonic Values

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

Table 7-18. *sf* Mnemonic Values

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

Operation: `See “Floating-Point Multiply Add” on page 7-48.`

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 floating-point arithmetic operations. The Invalid Operation is signaled in the same manner as the fcmp.lt operation.

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

Operation:

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

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

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

        fp_update_fpsr($sf$, tmp_fp_env);
    }

    fp_update_psr($f_1$);
}
```

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

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 floating-point arithmetic operations. The Invalid Operation is signaled in the same manner as the fcmp.lt operation.

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

Operation:

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

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

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

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr($f_1$);
}
```

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

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

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

Flush Cache

Format: (*qp*) fc *r_3* M28

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

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

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

Operation:

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

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 mnemonic values for *sf* are given in Table 7-18 on page 7-31.

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(($imm_{21}$ << 4), 25);
            if ((PSR.it && unimplemented_virtual_address(IP))
                || (!PSR.it && unimplemented_physical_address(IP)))
                unimplemented_instruction_address_trap(0, IP);
            if (PSR.tb)
                taken_branch_trap();
        } else
            speculation_fault(FCHKF, zero_ext($imm_{21}$, 21));
    }
}
```

FP Exceptions: None

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 7-20. 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 7-21 on page 7-41. The assembly syntax allows the specification of membership or non-membership and the assembler swaps the target predicates to achieve the desired effect.

Table 7-19. Floating-point Class Relations

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

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

- the number is NaTVal and $fclass_9$ {8} is 1, or
- the number is a quiet NaN and $fclass_9$ {7} is 1, or
- the number is a signaling NaN and $fclass_9$ {6} is 1, or
- the sign of the number agrees with the sign specified by one of the two low-order bits of $fclass_9$, and the type of the number (disregarding the sign) agrees with the number-type specified by the next 4 bits of $fclass_9$, as shown in Table 7-20.

Note: An $fclass_9$ of 0x1FF is equivalent to testing for any supported operand.

The class names used in Table 7-20 are defined in Table 5-2 on page 5-3.

Table 7-20. Floating-point Classes

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

Operation:

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

    if (tmp_isrcode = fp_reg_disabled($f_2$, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    tmp_rel = (($fclass_9${0} && !FR[$f_2$].sign || $fclass_9${1} && FR[$f_2$].sign)
                && (($fclass_9${2} && fp_is_zero(FR[$f_2$]))||
                     ($fclass_9${3} && fp_is_unorm(FR[$f_2$])) ||
                     ($fclass_9${4} && fp_is_normal(FR[$f_2$])) ||
                     ($fclass_9${5} && fp_is_inf(FR[$f_2$]))
                    )
                )
            || ($fclass_9${6} && fp_is_snan(FR[$f_2$]))
            || ($fclass_9${7} && fp_is_qnan(FR[$f_2$]))
            || ($fclass_9${8} && fp_is_natval(FR[$f_2$]));

    tmp_nat = fp_is_natval(FR[$f_2$]) && (!$fclass_9${8});

    if (tmp_nat) {
        PR[$p_1$] = 0;
        PR[$p_2$] = 0;
    } else {
        PR[$p_1$] = tmp_rel;
        PR[$p_2$] = !tmp_rel;
    }
} else {
    if (fctype == ‘unc’) {
        if (p1 == p2)
            illegal_operation_fault();
        PR[$p_1$] = 0;
        PR[$p_2$] = 0;
    }
}
```

FP Exceptions: None

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 7-18 on page 7-31.

Operation:

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

FP Exceptions: None

Floating-Point Compare

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

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

Table 7-21. 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 7-18 on page 7-31.

The relations are defined for each of the comparison types in Table 7-22. 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 7-22. 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

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 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 7-18 on page 7-31.

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, sf,
                                        signed_form, trunc_form, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

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

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

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

FP Exceptions:

Invalid Operation (V)	Inexact (I)
Denormal/Unnormal Operand (D)	
Software Assist (SWA) fault	

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

Convert Unsigned Integer to Floating-point

Format: (qp) fcvt.xuf.$pc.sf$ $f_1 = f_3$ (unsigned form) 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 7-17 on page 7-31. The mnemonic values for *sf* are given in Table 7-18 on page 7-31. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation: `See “Floating-Point Multiply Add” on page 7-48`

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 7-23.

Table 7-23. 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.

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.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 7-28 on page 7-105. Locality hints do not affect program functionality and may be ignored by the implementation.

Operation:

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

    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;
}
```

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.

After this instruction completes execution AR[BSPSTORE] is equal to AR[BSP].

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

Operation:

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

Floating-Point Multiply Add

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

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

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

If f_2 is f0, an IEEE multiply operation is performed instead of a multiply and add. See "Floating-Point Multiply" on page 7-55.

The mnemonic values for the opcode's *pc* are given in Table 7-17 on page 7-31. The mnemonic values for *sf* are given in Table 7-18 on page 7-31. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation:

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

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

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

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

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

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 floating-point arithmetic operations. The Invalid Operation is signaled in the same manner as the fcmp.lt operation.

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

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

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 7-7. Floating-point Merge Negative Sign Operation

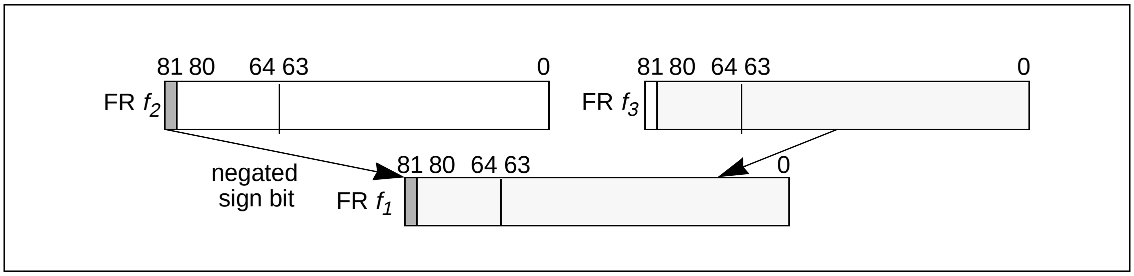


Figure 7-8. Floating-point Merge Sign Operation

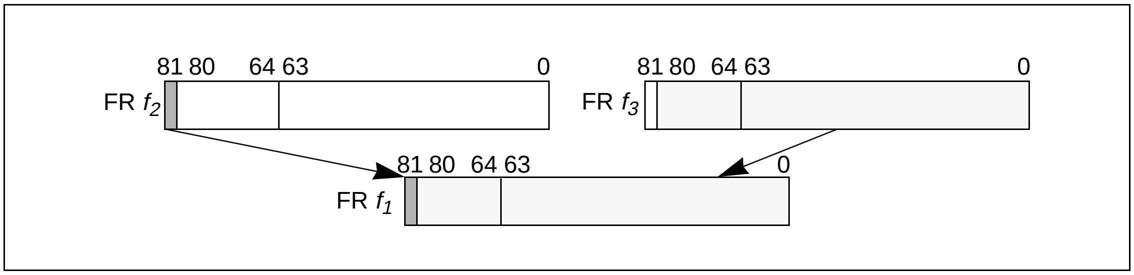
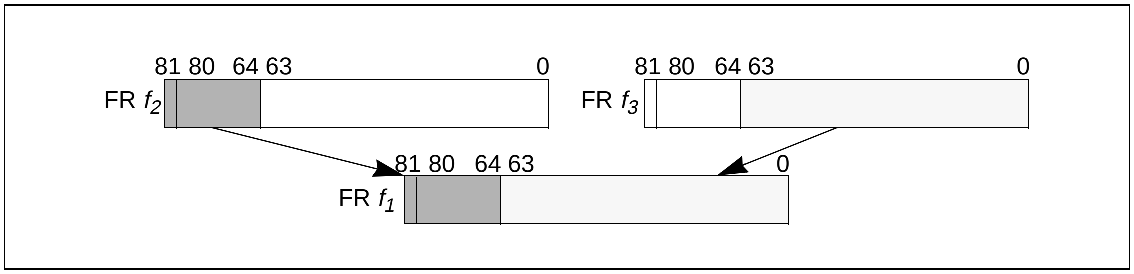


Figure 7-9. Floating-point Merge Sign and Exponent Operation



Operation:

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

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

    fp_update_psr(f1);
}
```

FP Exceptions: None

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 floating-point arithmetic operations. The Invalid Operation is signaled in the same manner as the fcmp.lt operation.

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

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

Floating-Point Parallel Mix

Format:

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

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

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

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

Figure 7-10. Floating-point Mix Left

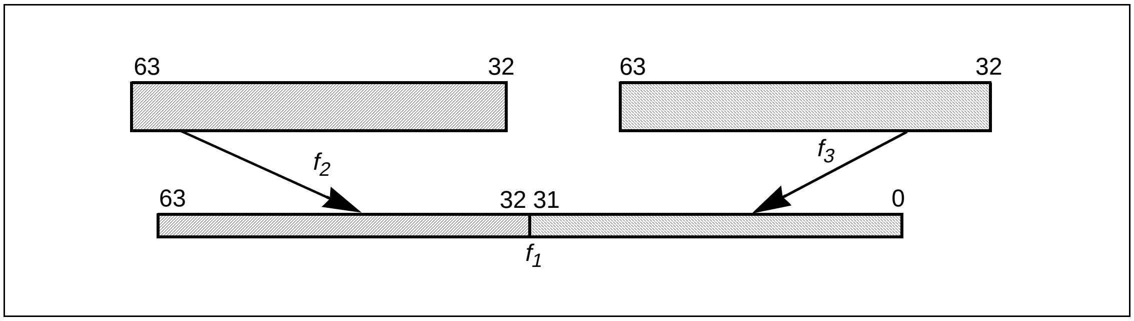


Figure 7-11. Floating-point Mix Right

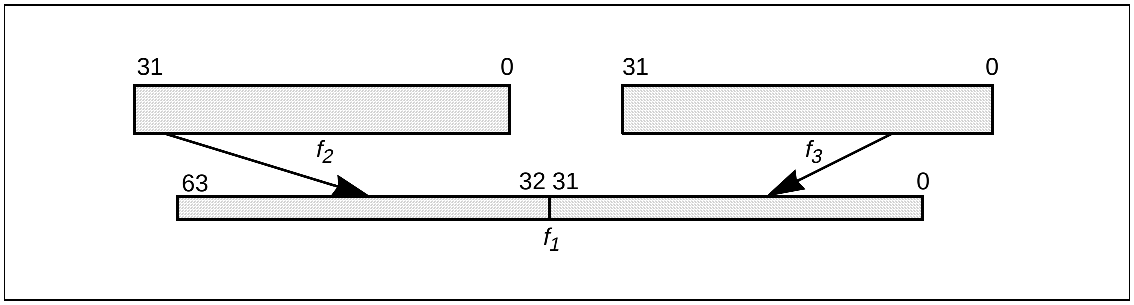
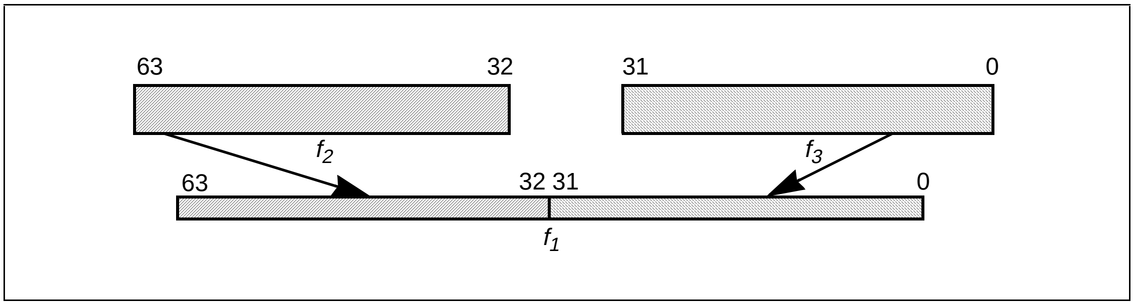


Figure 7-12. Floating-point Mix Left-Right



Operation:

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

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

    fp_update_psr($f_1$);
}
```

FP Exceptions: None

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 7-17 on page 7-31. The mnemonic values for *sf* are given in Table 7-18 on page 7-31. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation: `See “Floating-Point Multiply Add” on page 7-48.`

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.

The mnemonic values for the opcode's *pc* are given in Table 7-17 on page 7-31. The mnemonic values for *sf* are given in Table 7-18 on page 7-31. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation:

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

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

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

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

FP Exceptions:

Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault
Underflow (U)

Overflow (O)
Inexact (I)
Software Assist (SWA) trap

Floating-Point Negate

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

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

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

Operation: See “Floating-Point Merge” on page 7-50.

Floating-Point Negate Absolute Value

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

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

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

Operation: `See “Floating-Point Merge” on page 7-50.`

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.

The mnemonic values for the opcode's *pc* are given in Table 7-17 on page 7-31. The mnemonic values for *sf* are given in Table 7-18 on page 7-31. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation:

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

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

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

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

FP Exceptions:

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

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 7-17 on page 7-31. The mnemonic values for *sf* are given in Table 7-18 on page 7-31. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation: `See “Floating-Point Negative Multiply Add” on page 7-59.`

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 7-17 on page 7-31. The mnemonic values for *sf* are given in Table 7-18 on page 7-31. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation: `See "Floating-Point Multiply Add" on page 7-48.`

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

Floating-Point Parallel Absolute Value

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

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

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

Operation: `See “Floating-Point Parallel Merge” on page 7-74.`

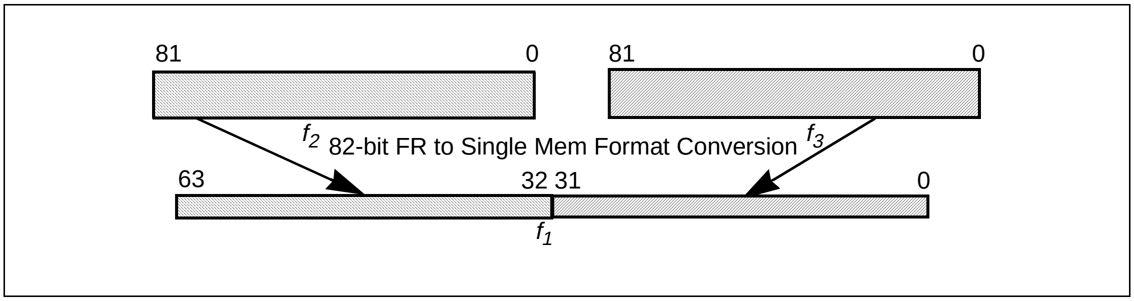
Floating-Point Pack

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 7-13. Floating-point Pack



Operation:

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

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

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

FP Exceptions: None

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 floating-point arithmetic operations. The Invalid Operation is signaled in the same manner as for the fpcmp.lt operation.

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

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

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 floating-point arithmetic operations. The Invalid Operation is signaled in the same manner as for the fpcmp.lt operation.

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

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

Floating-Point Parallel Compare

Format: (qp) fpcmp.*frel.sf* $f1 = f_2, f_3$ 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 7-24. 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 NaTVal's
unchanged	0...0	1...1	NaTVal

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

The relations are defined for each of the comparison types in Table 7-24. 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 7-25. 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($f_1$);
    if (tmp_isrcode = fp_reg_disabled($f_1$, $f_2$,$f_3$, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[$f_2$]) || fp_is_natval(FR[$f_3$])) {
        FR[$f_1$] = NATVAL;
    } else {
        fpcmp_exception_fault_check($f_2$, $f_3$, 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($f_2$);
        tmp_fr3 = fp_reg_read_hi($f_3$);

        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($f_2$);
        tmp_fr3 = fp_reg_read_lo($f_3$);

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

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

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

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 doesn't fit in 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 7-18 on page 7-31.

Operation:

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

    if (fp_is_natval(FR[$f_2$])) {
        FR[$f_1$] = NATVAL;
        fp_update_psr($f_1$);
    } else {
        tmp_default_result_pair = fpcvt_exception_fault_check($f_2$, sf,
                                        signed_form, trunc_form, &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($f_2$), 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($f_2$), 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[$f_1$].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[$f_1$].exponent = FP_INTEGER_EXP;
        FR[$f_1$].sign = FP_SIGN_POSITIVE;

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

Floating-Point Parallel Multiply Add

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

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

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

Note: If f_2 is f0 in the fpma instruction, just the IEEE multiply operation is performed. (See "Floating-Point Parallel Multiply" on page 7-78.) 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 7-18 on page 7-31.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 5-7.

Operation:

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

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

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

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

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

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

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

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 floating-point arithmetic operations. The Invalid Operation is signaled in the same manner as for the fpcmp.lt operation.

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

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

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 7-14. Floating-point Merge Negative Sign Operation

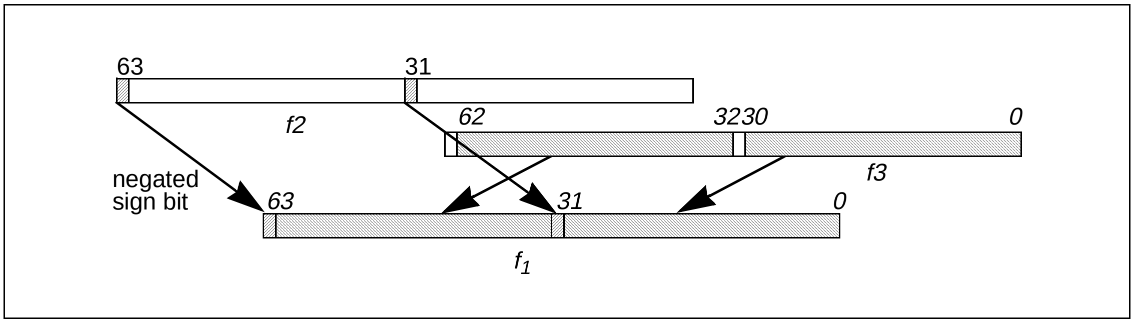


Figure 7-15. Floating-point Merge Sign Operation

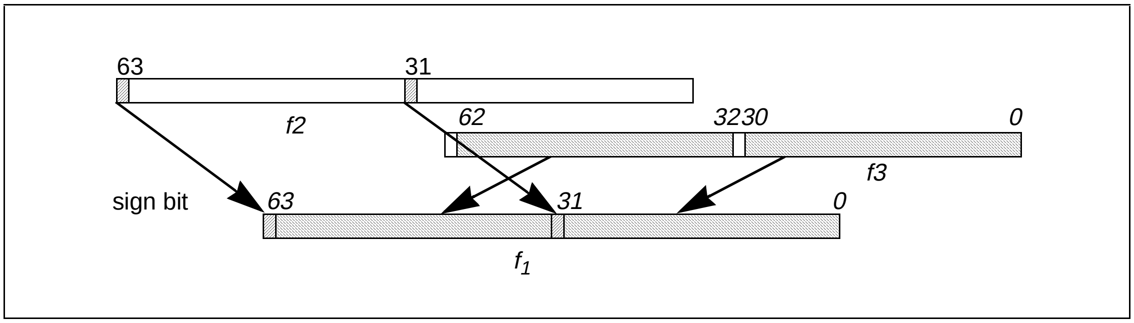
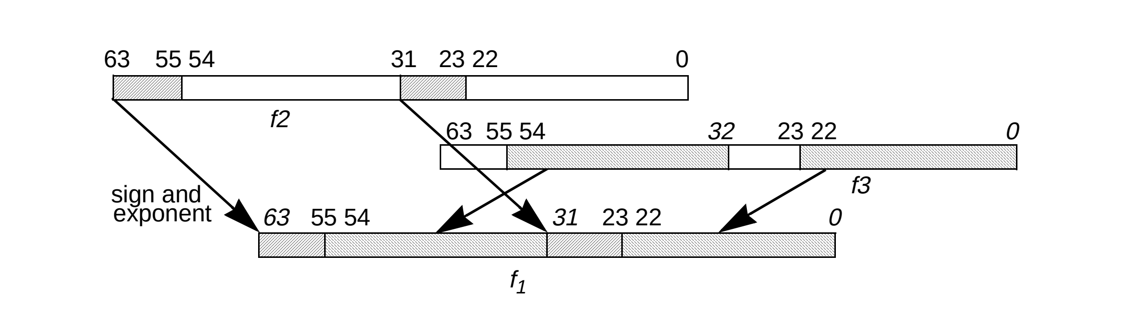


Figure 7-16. Floating-point Merge Sign and Exponent Operation



Operation:

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

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

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

    fp_update_psr(f1);
}
```

FP Exceptions: None

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 floating-point arithmetic operations. The Invalid Operation is signaled in the same manner as for the fpcmp.lt operation.

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

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

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 7-18 on page 7-31.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 5-7.

Operation: `See “Floating-Point Parallel Multiply Add” on page 7-71.`

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).

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 fpms instruction, just the IEEE multiply operation is performed.

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 5-7.

Operation:

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

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

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

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

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

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

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

Floating-Point Parallel Negate

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

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

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

Operation: `See “Floating-Point Parallel Merge” on page 7-74.`

Floating-Point Parallel Negate Absolute Value

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

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

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

Operation: `See “Floating-Point Parallel Merge” on page 7-74.`

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 7-18 on page 7-31.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 5-7.

Operation:

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

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

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

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

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

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

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

FP Exceptions:

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

Floating-Point Parallel Negative Multiply

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

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

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

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 5-7.

Operation: `See “Floating-Point Parallel Negative Multiply Add” on page 7-83.`

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 7-18 on page 7-31.

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($f_2$));
            den = fp_normalize(fp_reg_read_lo($f_3$));
            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[$f_1$].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[$f_1$].exponent = FP_INTEGER_EXP;
        FR[$f_1$].sign = FP_SIGN_POSITIVE;
        PR[$p_2$] = tmp_pred_hi && tmp_pred_lo;

        fp_update_fpsr($sf$, tmp_fp_env);
    }
    fp_update_psr($f_1$);
} else {
    PR[$p_2$] = 0;
}
```

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

Floating-Point Parallel Reciprocal Square Root Approximation

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

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

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

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

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

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

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

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 7-18 on page 7-31.

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)
{
   const EM_uint_t 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

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 7-18 on page 7-31.

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)
{
   const EM_uint_t 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

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

Floating-Point Set Controls

Format: (*qp*) fsetc.*sf* *amask7*, *omask7* F12

Description: The status field's control bits are initialized to the value obtained by logically AND-ing the sf0.controls and *amask7* immediate field and logically OR-ing the *omask7* immediate field.

The mnemonic values for *sf* are given in Table 7-18 on page 7-31.

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

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 7-17 on page 7-31. The mnemonic values for *sf* are given in Table 7-18 on page 7-31. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 5-7.

Operation: `See "Floating-Point Multiply Subtract" on page 7-56.`

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 7-17. Floating-point Swap

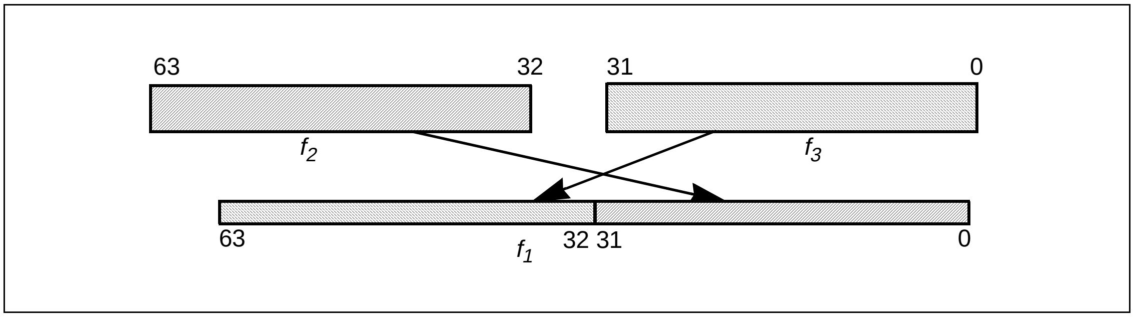
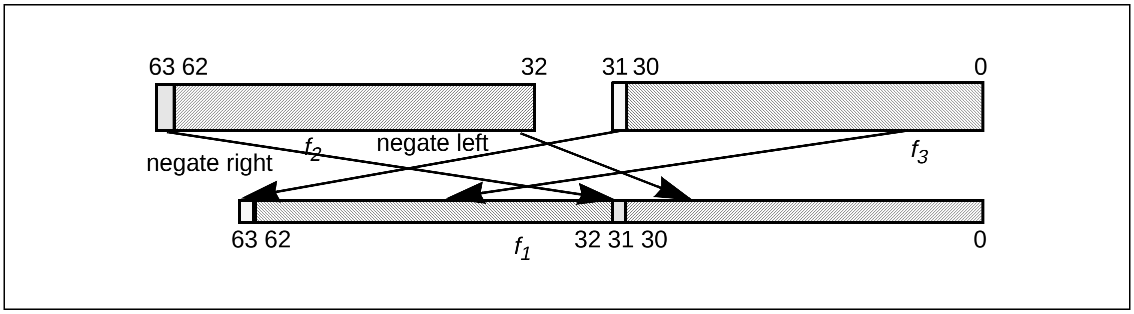


Figure 7-18. Floating-point Swap Negate Left or Right



Operation:

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

    if (fp_is_natval(FR[$f_2$]) || fp_is_natval(FR[$f_3$])) {
        FR[$f_1$] = NATVAL;
    } else {
        if (swap_form) {
            tmp_res_hi = FR[$f_3$].significand{31:0};
            tmp_res_lo = FR[$f_2$].significand{63:32};
        } else if (swap_nl_form) {
            tmp_res_hi = (!FR[$f_3$].significand{31} << 31)
                       | (FR[$f_3$].significand{30:0});
            tmp_res_lo = FR[$f_2$].significand{63:32};
        } else { // swap_nr_form
            tmp_res_hi = FR[$f_3$].significand{31:0};
            tmp_res_lo = (!FR[$f_2$].significand{63} << 31)
                       | (FR[$f_2$].significand{62:32});
        }

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

    fp_update_psr($f_1$);
}
```

FP Exceptions: None

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 7-19. Floating-point Sign Extend Left

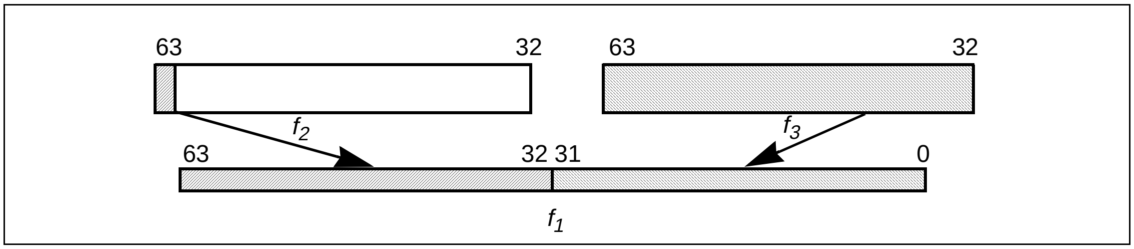
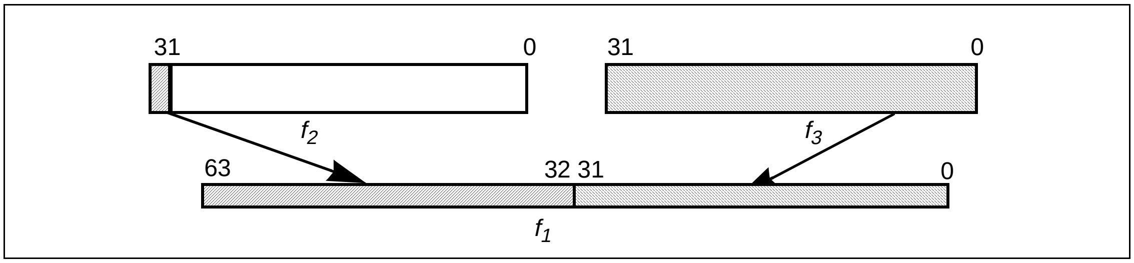


Figure 7-20. Floating-point Sign Extend Right



Operation:

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

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

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

    fp_update_psr(f1);
}
```

FP Exceptions: None

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

Get Floating-Point Value or Exponent or Significand

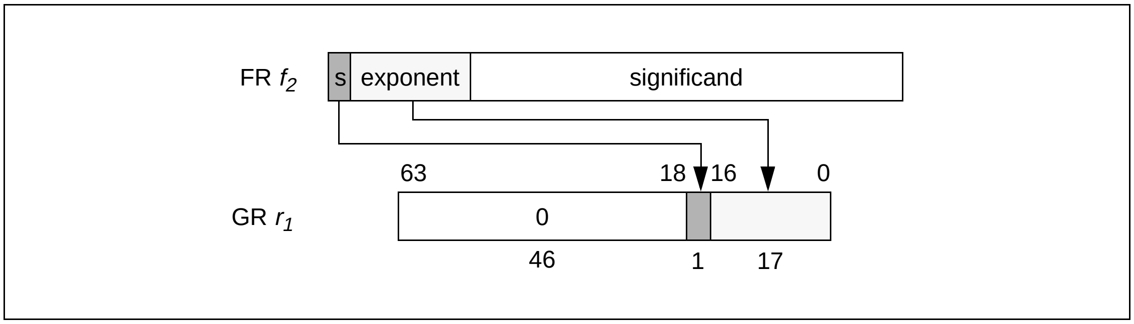
Format:

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

Description: In the single and double forms, the value in FR f_2 is converted into a single precision (single_form) or double precision (double_form) memory representation and placed in GR r_1. In the single_form, the most-significant 32 bits of GR r_1 are set to 0.

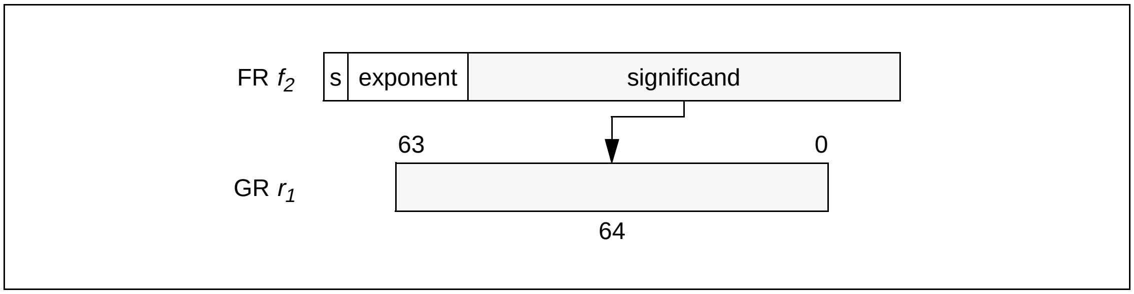
In the exponent_form, the exponent field of FR f_2 is copied to bits 16:0 of GR r_1 and the sign bit of the value in FR f_2 is copied to bit 17 of GR r_1. The most-significant 46-bits of GR r_1 are set to zero.

Figure 7-21. Function of getf.exp



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

Figure 7-22. Function of getf.sig



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

Operation:

```
if (PR[qp]) {
    check_target_register($r_1$);
    if (tmp_isrcode = fp_reg_disabled($f_2$, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (single_form) {
        GR[$r_1$]{31:0} = fp_fr_to_mem_format(FR[$f_2$], 4, 0);
        GR[$r_1$]{63:32} = 0;
    } else if (double_form) {
        GR[$r_1$] = fp_fr_to_mem_format(FR[$f_2$], 8, 0);
    } else if (exponent_form) {
        GR[$r_1$]{63:18} = 0;
        GR[$r_1$]{16:0} = FR[$f_2$].exponent;
        GR[$r_1$]{17} = FR[$f_2$].sign;
    } else // significand_form
        GR[$r_1$] = FR[$f_2$].significand;
    if (fp_is_natval(FR[$f_2$]))
        GR[$r_1$].nat = 1;
    else
        GR[$r_1$].nat = 0;
}
```

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);
    }
}
```

Load

Format:

(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3]	no_base_update_form	M1
(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3], r_2	reg_base_update_form	M2
(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3], imm_9	imm_base_update_form	M3
(*qp*) ld8.fill.*ldhint* r_1 = [r_3]	fill_form, no_base_update_form	M1
(*qp*) ld8.fill.*ldhint* r_1 = [r_3], r_2	fill_form, reg_base_update_form	M2
(*qp*) ld8.fill.*ldhint* r_1 = [r_3], imm_9	fill_form, imm_base_update_form	M3

Description: A value consisting of *sz* bytes is read from memory starting at the address specified by the value in GR r_3. The value is then zero extended and placed in GR r_1. The values of the *sz* completer are given in Table 7-26. 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 7-27.

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 "Control Speculation" on page 4-13 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 7-26. *sz* Completers

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

Table 7-27. 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.
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).

Table 7-27. Load Types (Cont'd)

ldtype Completer	Interpretation	Special Load Operation
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 "Control Speculation" on page 4-13 and "Data Speculation" on page 4-16. For more details on ordered loads see "Memory Access Ordering" on page 4-23. See "Memory Hierarchy Control and Consistency" on page 4-20 for details on biased loads.

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 7-28. 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 "Memory Hierarchy Control and Consistency" on page 4-20 for details.

Table 7-28. Load Hints

ldhint Completer	Interpretation
none	Temporal locality, level 1
nt1	No temporal locality, level 1
nta	No temporal locality, all levels

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

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

Operation:

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

    speculative = (ldtype == ‘s’ || ldtype == ‘sa’);
    advanced = (ldtype == ‘a’ || ldtype == ‘sa’);
    check_clear = (ldtype == ‘c.clr’ || ldtype == ‘c.clr.acq’);
    check_no_clear = (ldtype == ‘c.nc’);
    check = check_clear || check_no_clear;
    acquire = (ldtype == ‘acq’ || ldtype == ‘c.clr.acq’);
    bias = (ldtype == ‘bias’) ? BIAS : 0 ;

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

    if ((reg_base_update_form || imm_base_update_form) && ($r_1$ == $r_3$))
        illegal_operation_fault();
    check_target_register($r_1$, itype);
    if (reg_base_update_form || imm_base_update_form)
        check_target_register($r_3$);

    if (reg_base_update_form) {
        tmp_r2 = GR[$r_2$];
        tmp_r2nat = GR[$r_2$].nat;
    }

    if (!speculative && GR[$r_3$].nat)                  // fault on NaT address
        register_nat_consumption_fault(itype);
    defer = speculative && (GR[$r_3$].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(GENERAL, $r_1$)) {            // no load on ld.c & ALAT hit
        if (check_clear)              // remove entry on ld.c.clr or ld.c.clr.acq
            alat_inval_single_entry(GENERAL, $r_1$);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[$r_3$], size, itype, PSR.cpl, &mattr,
                                          &defer);
            if (!defer) {
                otype = acquire ? ACQUIRE : UNORDERED;
                val = mem_read(paddr, size, UM.be, mattr, otype, bias | ldhint);
            }
        }
        if (check_clear || advanced)                    // remove any old ALAT entry
            alat_inval_single_entry(GENERAL, $r_1$);
        if (defer) {
            if (speculative) {
                GR[$r_1$] = natd_gr_read(paddr, size, UM.be, mattr, otype,
                                          bias | ldhint);
                GR[$r_1$].nat = 1;
            } else {
                GR[$r_1$] = 0;                          // ld.a to sequential memory
                GR[$r_1$].nat = 0;
            }
        } else {                                        // execute load normally
            if (fill_form) {                            // fill NaT on ld8.fill
                bit_pos = GR[$r_3$]{8:3};
                GR[$r_1$] = val;
                GR[$r_1$].nat = AR[UNAT]{bit_pos};
            } else {                                    // clear NaT on other types
                GR[$r_1$] = zero_ext(val, size * 8);
                GR[$r_1$].nat = 0;
            }
```

```
         if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                // add entry to ALAT
            alat_write(GENERAL, $r_1$, paddr, size);
      }
   }

   if (imm_base_update_form) {                  // update base register
      GR[$r_3$] = GR[$r_3$] + sign_ext($imm_9$, 9);
      GR[$r_3$].nat = GR[$r_3$].nat;
   } else if (reg_base_update_form) {
      GR[$r_3$] = GR[$r_3$] + tmp_r2;
      GR[$r_3$].nat = GR[$r_3$].nat || tmp_r2nat;
   }

   if ((reg_base_update_form || imm_base_update_form) && !GR[$r_3$].nat)
      mem_implicit_prefetch(GR[$r_3$], bias | ldhint);
}
```

Floating-Point Load

Format:

(*qp*) ldf*fsz*.*fldtype*.*ldhint* f_1 = [r_3]	no_base_update_form	M6
(*qp*) ldf*fsz*.*fldtype*.*ldhint* f_1 = [r_3], r_2	reg_base_update_form	M7
(*qp*) ldf*fsz*.*fldtype*.*ldhint* f_1 = [r_3], imm_9	imm_base_update_form	M8
(*qp*) ldf8.*fldtype*.*ldhint* f_1 = [r_3]	integer_form, no_base_update_form	M6
(*qp*) ldf8.*fldtype*.*ldhint* f_1 = [r_3], r_2	integer_form, reg_base_update_form	M7
(*qp*) ldf8.*fldtype*.*ldhint* f_1 = [r_3], imm_9	integer_form, imm_base_update_form	M8
(*qp*) ldf.fill.*ldhint* f_1 = [r_3]	fill_form, no_base_update_form	M6
(*qp*) ldf.fill.*ldhint* f_1 = [r_3], r_2	fill_form, reg_base_update_form	M7
(*qp*) ldf.fill.*ldhint* f_1 = [r_3], imm_9	fill_form, imm_base_update_form	M8

Description: A value consisting of *fsz* bytes is read from memory starting at the address specified by the value in GR r_3. The value is then converted into the floating-point register format and placed in FR f_1. See "Data Types and Formats" on page 5-1 for details on conversion to floating-point register format. The values of the *fsz* completer are given in Table 7-29. The *fldtype* completer specifies special load operations, which are described in Table 7-30.

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 "Control Speculation" on page 4-13 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 7-29. *fsz* Completers

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

Table 7-30. 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 7-30. FP Load Types (Cont'd)

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 "Control Speculation" on page 4-13 and "Data Speculation" on page 4-16.

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 7-28 on page 7-105. 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 "Memory Hierarchy Control and Consistency" on page 4-20 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.

Operation:

```
if (PR[qp]) {
    size = (fill_form ? 16 : (integer_form ? 8 : fsz));
    speculative = (fldtype == ‘s’ || fldtype == ‘sa’);
    advanced = (fldtype == ‘a’ || fldtype == ‘sa’);
    check_clear = (fldtype == ‘c.clr’ );
    check_no_clear = (fldtype == ‘c.nc’);
    check = check_clear || check_no_clear;

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

    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, itype);

    if (!speculative && GR[r3].nat)                 // fault on NaT address
        register_nat_consumption_fault(itype);

    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(FLOAT, f1)) {             // no load on ldf.c & ALAT hit
        if (check_clear)                            // remove entry on ldf.c.clr
            alat_inval_single_entry(FLOAT, f1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                    &defer);
            if (!defer)
                val = mem_read(paddr, size, UM.be, mattr, UNORDERED, ldhint);
        }
        if (check_clear || advanced)                // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
        if (speculative && defer) {
            FR[f1] = NATVAL;
        } else if (advanced && !speculative && defer) {
            FR[f1] = (integer_form ? FP_INT_ZERO : FP_ZERO);
        } else {                                    // execute load normally
            FR[f1] = fp_mem_to_fr_format(val, size, integer_form);

            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                    // add entry to ALAT
                alat_write(FLOAT, f1, paddr, size);
        }
    }

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

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

    fp_update_psr(f1);
}
```

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 "Data Types and Formats" on page 5-1 for details on conversion to floating-point register format. The *fldtype* completer specifies special load operations, which are described in Table 7-30 on page 7-108.

For more details on speculative, advanced and check loads see "Control Speculation" on page 4-13 and "Data Speculation" on page 4-16.

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 7-28 on page 7-105. 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 "Memory Hierarchy Control and Consistency" on page 4-20 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;
    itype = READ;
    if (speculative) itype |= SPEC;
    if (advanced) itype |= ADVANCE;

    if (fp_reg_bank_conflict(f1, f2))
        illegal_operation_fault();

    if (base_update_form)
        check_target_register(r3);

    fp_check_target_register(f1);
    fp_check_target_register(f2);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, itype);

    if (!speculative && GR[r3].nat)              // fault on NaT address
        register_nat_consumption_fault(itype);

    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(FLOAT, f1)) {          // no load on ldfp.c & ALAT hit
        if (check_clear)                         // remove entry on ldfp.c.clr
            alat_inval_single_entry(FLOAT, f1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                           &defer);
            if (!defer)
                val = mem_read(paddr, size, UM.be, mattr, UNORDERED, ldhint);
        }
        if (check_clear || advanced)             // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
        if (speculative && defer) {
            FR[f1] = NATVAL;
            FR[f2] = NATVAL;
        } else if (advanced && !speculative && defer) {
            FR[f1] = (integer_form ? FP_INT_ZERO : FP_ZERO);
            FR[f2] = (integer_form ? FP_INT_ZERO : FP_ZERO);
        } else {                                 // execute load normally
            if (UM.be) {
                FR[f1] = fp_mem_to_fr_format(val u>> (size/2*8), size/2,
                       integer_form);
                FR[f2] = fp_mem_to_fr_format(val, size/2, integer_form);
            } else {
                FR[f1] = fp_mem_to_fr_format(val, size/2, integer_form);
                FR[f2] = fp_mem_to_fr_format(val u>> (size/2*8), size/2,
                                             integer_form);
            }

            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                 // add entry to ALAT
                alat_write(FLOAT, f1, paddr, size);
        }
    }

    if (base_update_form) {                      // update base register
        GR[r3] = GR[r3] + size;
        GR[r3].nat = GR[r3].nat;
        if (!GR[r3].nat)
            mem_implicit_prefetch(GR[r3], ldhint);
    }

    fp_update_psr(f1);
    fp_update_psr(f2);
}
```

Line Prefetch

Format:

(*qp*) lfetch.*lftype*.*lfhint* [r_3]	no_base_update_form	M13
(*qp*) lfetch.*lftype*.*lfhint* [r_3], r_2	reg_base_update_form	M14
(*qp*) lfetch.*lftype*.*lfhint* [r_3], imm_9	imm_base_update_form	M15
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3]	no_base_update_form, exclusive_form	M13
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], r_2	reg_base_update_form, exclusive_form	M14
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], imm_9	imm_base_update_form, exclusive_form	M15

Description: The line containing the address specified by the value in GR r_3 is moved to the highest level of the data memory hierarchy. The value of the *lfhint* modifier specifies the locality of the memory access. The mnemonic values of *lfhint* are given in Table 7-32.

The behavior of the memory read is also determined by the memory attribute associated with the accessed page. 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 7-31 defines these two options.

Table 7-31. *lftype* Mnemonic Values

lftype Mnemonic	Interpretation
none	Ignore faults
fault	Raise faults

In the base update forms, after being used to address memory, the value in GR r_3 is incremented by either the sign extended value in imm_9 (in the imm_base_update_form) or the value in GR r_2 (in the reg_base_update_form). In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set – no fault is raised.

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

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

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

Table 7-32. *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

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($r_3$);

    if (lftype == 'fault') {            // faulting form
        if (GR[$r_3$].nat && !PSR.ed)      // fault on NaT address
            register_nat_consumption_fault(itype);
    }

    if (exclusive_form)
        excl_hint = EXCLUSIVE;
    else
        excl_hint = 0;

    if (!GR[$r_3$].nat && !PSR.ed) {// faulting form already faulted if $r_3$ is nat'ed
        paddr = tlb_translate(GR[$r_3$], 1, itype, PSR.cpl, &mattr, &defer);
        if (!defer)
            mem_promote(paddr, mattr, lfhint | excl_hint);
    }

    if (imm_base_update_form) {
        GR[$r_3$] = GR[$r_3$] + sign_ext($imm_9$, 9);
        GR[$r_3$].nat = GR[$r_3$].nat;
    } else if (reg_base_update_form) {
        GR[$r_3$] = GR[$r_3$] + GR[$r_2$];
        GR[$r_3$].nat = GR[$r_2$].nat || GR[$r_3$].nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[$r_3$].nat)
        mem_implicit_prefetch(GR[$r_3$], lfhint | excl_hint);
}
```

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 have returned data, and
- all prior stores 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 IO transaction has been “accepted”, before initiating additional external transactions. The acceptance_form does not ensure ordering.

Operation:

```
if (PR[qp]){
    if (acceptance_form)
        acceptance_fence();
    else
        ordering_fence();
}
```

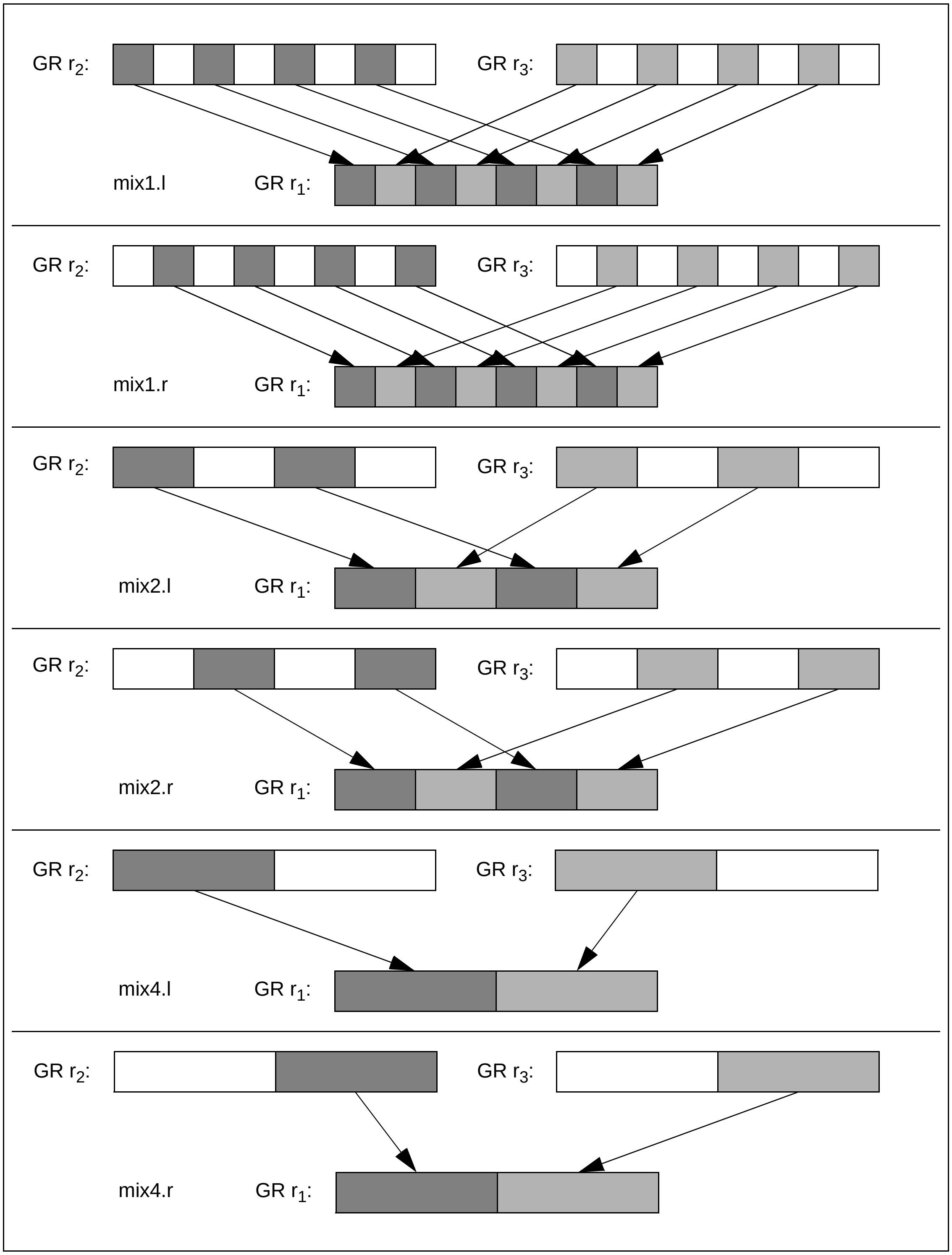
Mix

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 7-23, and the result placed in GR r_1. The data elements in the source registers are grouped in pairs, and one element from each pair is selected for the result. In the left_form, the result is formed from the leftmost elements from each of the pairs. In the right_form, the result is formed from the rightmost elements. Elements are selected alternately from the two source registers.

Figure 7-23. Mix Example



Operation:

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

    if (one_byte_form) {                                           // one-byte elements
        x[0] = GR[r2]{7:0};     y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};    y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};   y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};   y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};   y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};   y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};   y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};   y[7] = GR[r3]{63:56};

        if (left_form)
            GR[r1] = concatenate8(x[7], y[7], x[5], y[5],
                                x[3], y[3], x[1], y[1]);
        else
            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
            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
            GR[r1] = concatenate2(x[0], y[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Move Application Register

Format:

(*qp*) mov $r_1 = ar_3$	pseudo-op	
(*qp*) mov $ar_3 = r_2$	pseudo-op	
(*qp*) mov $ar_3 = imm_8$	pseudo-op	
(*qp*) mov.i $r_1 = ar_3$	i_form, from_form	I28
(*qp*) mov.i $ar_3 = r_2$	i_form, register_form, to_form	I26
(*qp*) mov.i $ar_3 = imm_8$	i_form, immediate_form, to_form	I27
(*qp*) mov.m $r_1 = ar_3$	m_form, from_form	M31
(*qp*) mov.m $ar_3 = r_2$	m_form, register_form, to_form	M29
(*qp*) mov.m $ar_3 = imm_8$	m_form, immediate_form, to_form	M30

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

In the from_form, the application register specified by ar_3 is copied into GR r_1 and the corresponding NaT bit is cleared.

In the to_form, the value in GR r_2 (in the register_form), or the sign extended value in imm_8 (in the immediate_form), is placed in AR ar_3. In the register_form if the NaT bit corresponding to GR r_2 is set, then a Register NaT Consumption fault is raised.

Only a subset of the application registers can be accessed by each execution unit (M or I). Table 3-3 on page 3-6 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 dependencies 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, $ar_3$))
        illegal_operation_fault();

    if (from_form) {
        check_target_register($r_1$);
        if ((($ar_3$ == BSPSTORE) || ($ar_3$ == RNAT)) && (AR[RSC].mode != 0))
            illegal_operation_fault();

        if ($ar_3$ == ITC && PSR.si && PSR.cpl != 0)
            privileged_register_fault();

        GR[$r_1$] = (is_ignored_reg($ar_3$)) ? 0 : AR[$ar_3$];
        GR[$r_1$].nat = 0;
    } else {                                                        // to_form
        tmp_val = (register_form) ? GR[$r_2$] : sign_ext($imm_8$, 8);

        if ($ar_3$ == BSP)
            illegal_operation_fault();

        if ((($ar_3$ == BSPSTORE) || ($ar_3$ == RNAT)) && (AR[RSC].mode != 0))
            illegal_operation_fault();

        if (register_form && GR[$r_2$].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(AR_TYPE, $ar_3$, tmp_val))
            reserved_register_field_fault();

        if ((is_kernel_reg($ar_3$) || $ar_3$ == ITC) && (PSR.cpl != 0))
            privileged_register_fault();

        if (!is_ignored_reg($ar_3$)) {
            tmp_val = ignored_field_mask(AR_TYPE, $ar_3$, tmp_val);
            // check for illegal promotion
            if ($ar_3$ == RSC && tmp_val{3:2} u< PSR.cpl)
                tmp_val{3:2} = PSR.cpl;
            AR[$ar_3$] = tmp_val;

            if ($ar_3$ == BSPSTORE) {
                AR[BSP] = rse_update_internal_stack_pointers(tmp_val);
                AR[RNAT] = undefined();
            }
        }
    }
}
```

Move Branch Register

Format:

(*qp*) mov $r_1 = b_2$	from_form	I22
(*qp*) mov$b_1 = r_2$	to_form	I21
(*qp*) mov.ret.$b_1 = r_2$	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.

Operation:

```
if (PR[qp]) {
    if (from_form) {
        check_target_register($r_1$);
        GR[$r_1$] = BR[$b_2$];
        GR[$r_1$].nat = 0;
    } else { // to_form
        if (GR[$r_2$].nat)
            register_nat_consumption_fault(0);
        BR[$b_1$] = GR[$r_2$];
    }
}
```

Move Floating-Point Register

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

Description: The value of FR f_3 is copied to FR f_1.

Operation: `See “Floating-Point Merge” on page 7-50.`

Move General Register

Format: (*qp*) mov $r_1 = r_3$ pseudo-op of: (*qp*) adds $r_1 = 0, r_3$

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

Operation: `See “Add” on page 7-3.`

Move Immediate

Format: (*qp*) mov r_1 = imm_{22} pseudo-op of: (*qp*) addl r_1 = imm_{22}, r0

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

Operation: `See “Add” on page 7-3.`

Move Indirect Register

Format: (*qp*) mov r_1 = *ireg*[r_3] from_form M43

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 7-33 below). The indexed register is read and its value is copied into GR r_1.

Table 7-33. Indirect Register File Mnemonics

ireg	Register File
cpuid	Processor Identification Register
pmd	Performance Monitor Data Register

Bits {7:0} of GR r_3 are used as the index. The remainder of the bits are ignored.

Apart from the PMD register file, access of a non-existent register results in a Reserved Register/Field fault. All accesses to the implementation-dependent portion of the PMD register file result in implementation dependent behavior but do not fault.

Operation:

```
if (PR[qp]) {
    tmp_index = GR[r3]{7:0};

    if (from_form) {
        check_target_register(r1);

        if (GR[r3].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_reg(ireg, tmp_index))
            reserved_register_field_fault();

        if (ireg == PMD_TYPE) {
            GR[r1] = pmd_read(tmp_index);
        } else
            switch (ireg) {
                case CPUID_TYPE:  GR[r1] = CPUID[tmp_index]; break;
            }
        GR[r1].nat = 0;
    }
}
```

Move Instruction Pointer

Format: (*qp*) mov r_1 = ip I25

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

Operation:

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

    GR[$r_1$] = IP;
    GR[$r_1$].nat = 0;
}
```

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};
        }
    }
}
```

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.

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{5:0} = GR[r2]{5:0};
    }
}
```

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;
}
```

Mux

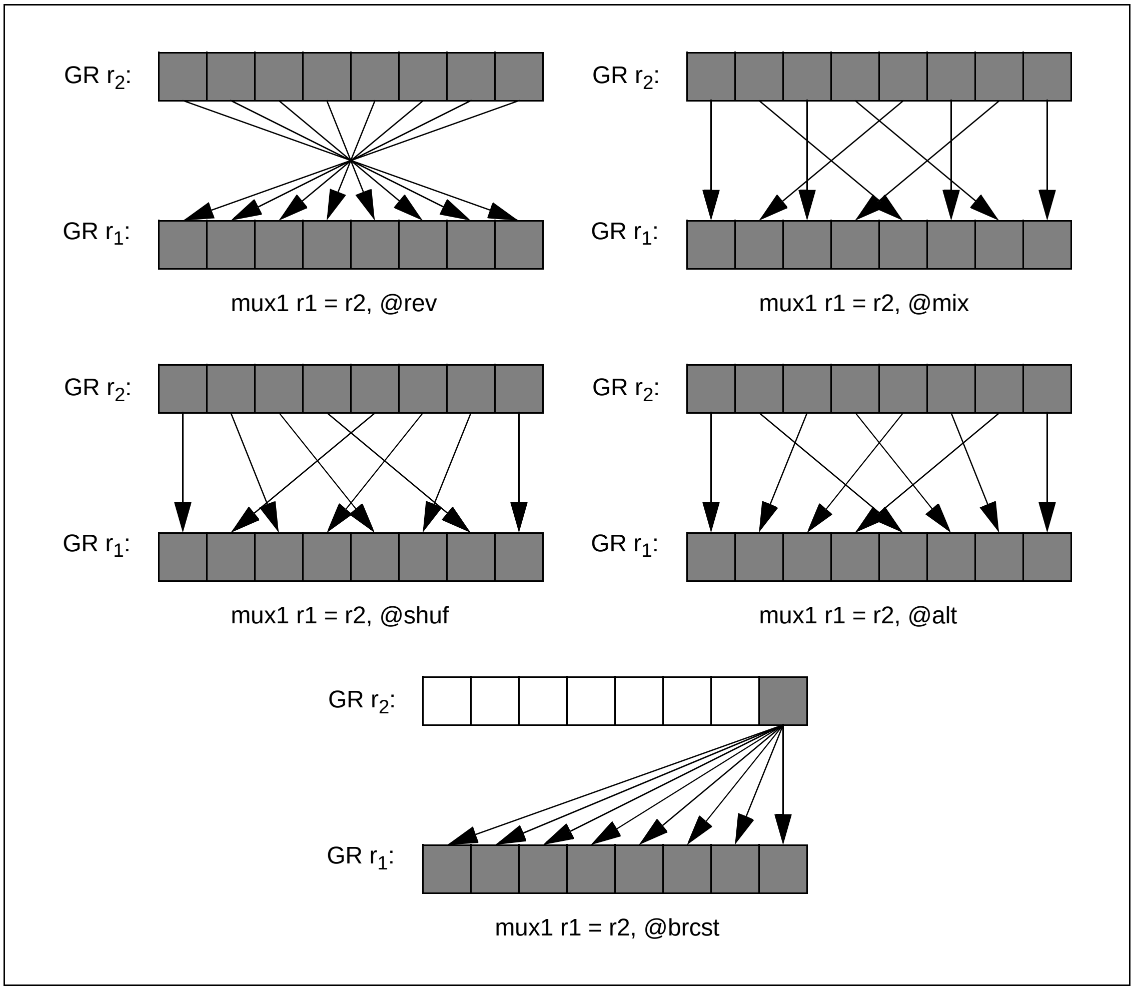
Format: (qp) mux1 $r_1 = r_2, mbtype_4$ one_byte_form I3
(qp) mux2 $r_1 = r_2, mhtype_8$ two_byte_form I4

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 7-34 and shown in Figure 7-24.

Table 7-34. Mux Permutations for 8-bit Elements

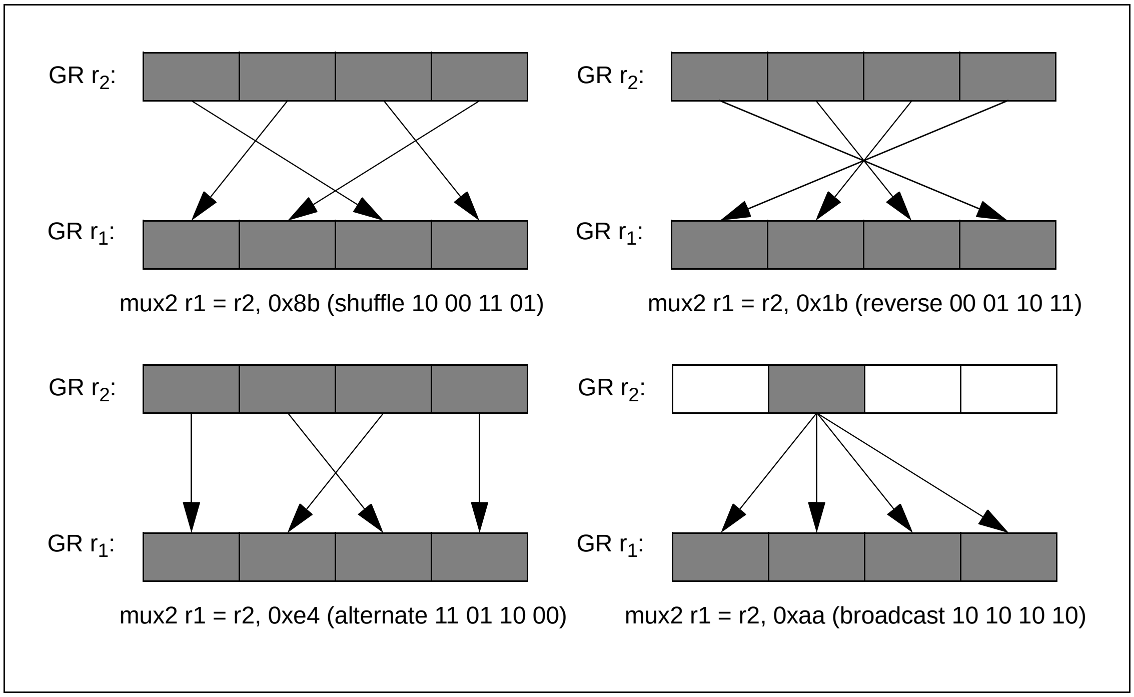
$mbtype_4$	Function
@rev	Reverse the order of the bytes
@mix	Perform a Mix operation on the two halves of GR r_2
@shuf	Perform a Shuffle operation on the two halves of GR r_2
@alt	Perform an Alternate operation on the two halves of GR r_2
@brcst	Perform a Broadcast operation on the least significand byte of GR r_2

Figure 7-24. Mux1 Operation (8-bit elements)



For 16-bit elements, all possible permutations, with and without repetitions can be specified. They are expressed with an 8-bit $mhtype_8$ field, which encodes the indices of the four 16-bit data elements. The indexed 16-bit elements of GR r_2 are copied to corresponding 16-bit positions in the target register GR r_1. The indices are encoded in little-endian order. (The 8 bits of $mhtype_8$[7:0] are grouped in pairs of bits and named $mhtype_8$[3], $mhtype_8$[2], $mhtype_8$[1], $mhtype_8$[0] in the Operation section).

Figure 7-25. Mux2 Examples (16-bit elements)


GR r_2:
GR r_1:
mux2 r1 = r2, 0x8b (shuffle 10 00 11 01)
GR r_2:
GR r_1:
mux2 r1 = r2, 0x1b (reverse 00 01 10 11)
GR r_2:
GR r_1:
mux2 r1 = r2, 0xe4 (alternate 11 01 10 00)
GR r_2:
GR r_1:
mux2 r1 = r2, 0xaa (broadcast 10 10 10 10)

Operation:

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

    if (one_byte_form) {
        x[0] = GR[$r_2$]{7:0};
        x[1] = GR[$r_2$]{15:8};
        x[2] = GR[$r_2$]{23:16};
        x[3] = GR[$r_2$]{31:24};
        x[4] = GR[$r_2$]{39:32};
        x[5] = GR[$r_2$]{47:40};
        x[6] = GR[$r_2$]{55:48};
        x[7] = GR[$r_2$]{63:56};

        switch (mbtype) {
            case ‘@rev’:
                GR[$r_1$] = concatenate8(x[0], x[1], x[2], x[3],
                                         x[4], x[5], x[6], x[7]);
                break;

            case ‘@mix’:
                GR[$r_1$] = concatenate8(x[7], x[3], x[5], x[1],
                                         x[6], x[2], x[4], x[0]);
                break;

            case ‘@shuf’:
                GR[$r_1$] = concatenate8(x[7], x[3], x[6], x[2],
                                         x[5], x[1], x[4], x[0]);
                break;

            case ‘@alt’:
                GR[$r_1$] = concatenate8(x[7], x[5], x[3], x[1],
                                         x[6], x[4], x[2], x[0]);
                break;

            case ‘@brcst’:
                GR[$r_1$] = 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[$r_2$]{15:0};
        x[1] = GR[$r_2$]{31:16};
        x[2] = GR[$r_2$]{47:32};
        x[3] = GR[$r_2$]{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[$r_1$] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[$r_1$].nat = GR[$r_2$].nat;
}
```

No Operation

Format:

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

Description: No operation is done.

The immediate, imm_{21} or imm_{62}, can be used by software as a marker in program code. It is ignored by hardware.

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

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
}
```

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;
}
```

Pack

Format:

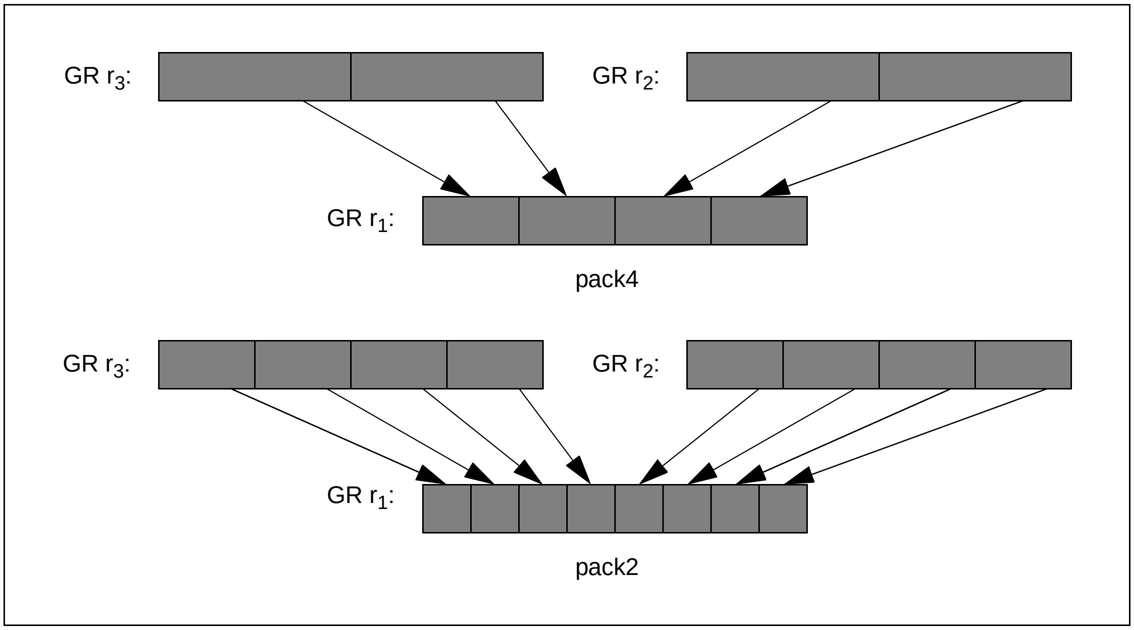
(*qp*)	pack2.sss $r_1 = r_2, r_3$	two_byte_form, signed_saturation_form	I2
(*qp*)	pack2.uss $r_1 = r_2, r_3$	two_byte_form, unsigned_saturation_form	I2
(*qp*)	pack4.sss $r_1 = r_2, r_3$	four_byte_form, signed_saturation_form	I2

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 7-35.

Table 7-35. 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 7-26. Pack Operation



Operation:

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

    if (two_byte_form) {                                         // two_byte_form
        if (signed_saturation_form) {                   // 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;
}
```

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 7-36. 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 7-37.

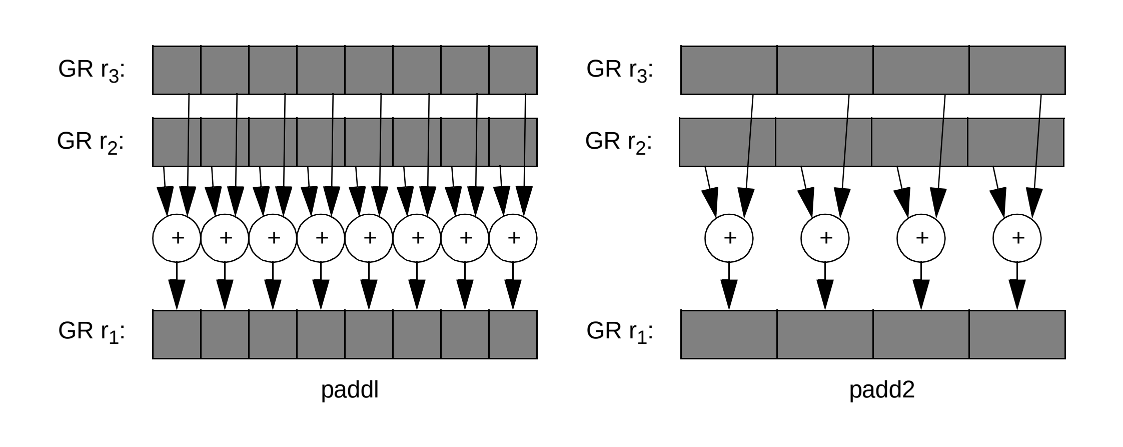
Table 7-36. Parallel Add Saturation Completers

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 7-37. 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 7-27. Parallel Add Examples



Operation:

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

   if (one_byte_form) {                                      // one-byte elements
      x[0] = GR[$r_2$]{7:0};       y[0] = GR[$r_3$]{7:0};
      x[1] = GR[$r_2$]{15:8};      y[1] = GR[$r_3$]{15:8};
      x[2] = GR[$r_2$]{23:16};     y[2] = GR[$r_3$]{23:16};
      x[3] = GR[$r_2$]{31:24};     y[3] = GR[$r_3$]{31:24};
      x[4] = GR[$r_2$]{39:32};     y[4] = GR[$r_3$]{39:32};
      x[5] = GR[$r_2$]{47:40};     y[5] = GR[$r_3$]{47:40};
      x[6] = GR[$r_2$]{55:48};     y[6] = GR[$r_3$]{55:48};
      x[7] = GR[$r_2$]{63:56};     y[7] = GR[$r_3$]{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[$r_1$] = 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[$r_2$]{15:0};      y[0] = GR[$r_3$]{15:0};
      x[1] = GR[$r_2$]{31:16};     y[1] = GR[$r_3$]{31:16};
      x[2] = GR[$r_2$]{47:32};     y[2] = GR[$r_3$]{47:32};
      x[3] = GR[$r_2$]{63:48};     y[3] = GR[$r_3$]{63:48};

      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;
}
```

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 7-28. Parallel Average Example

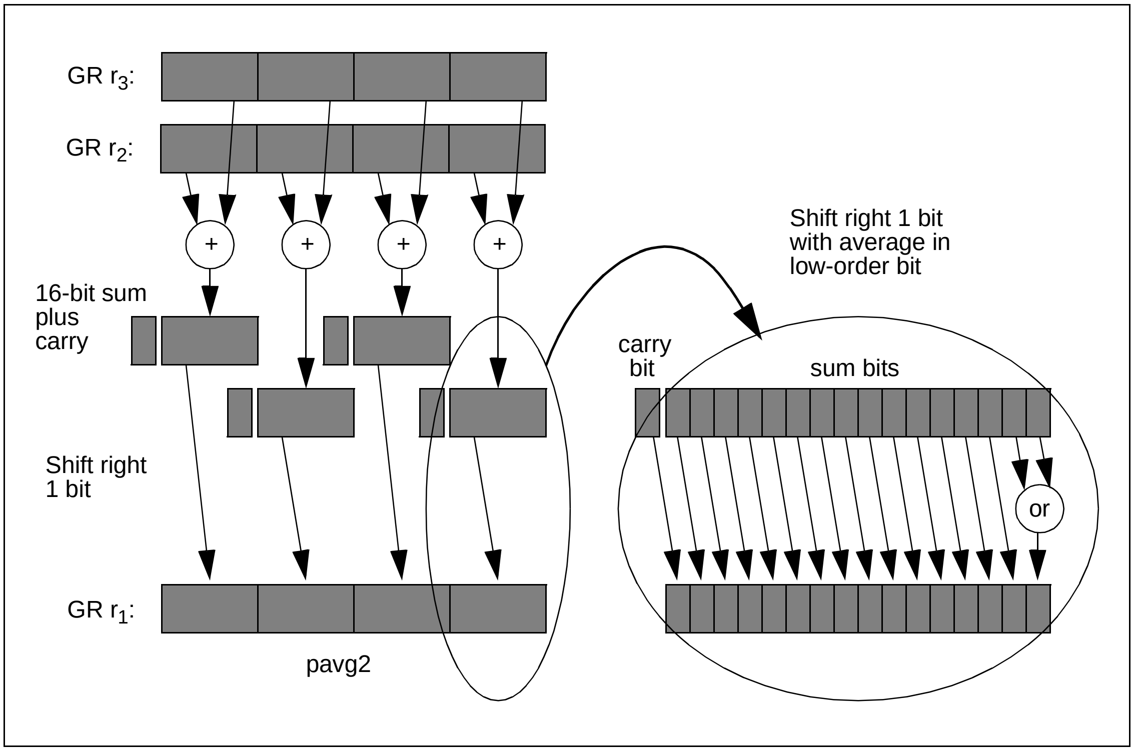
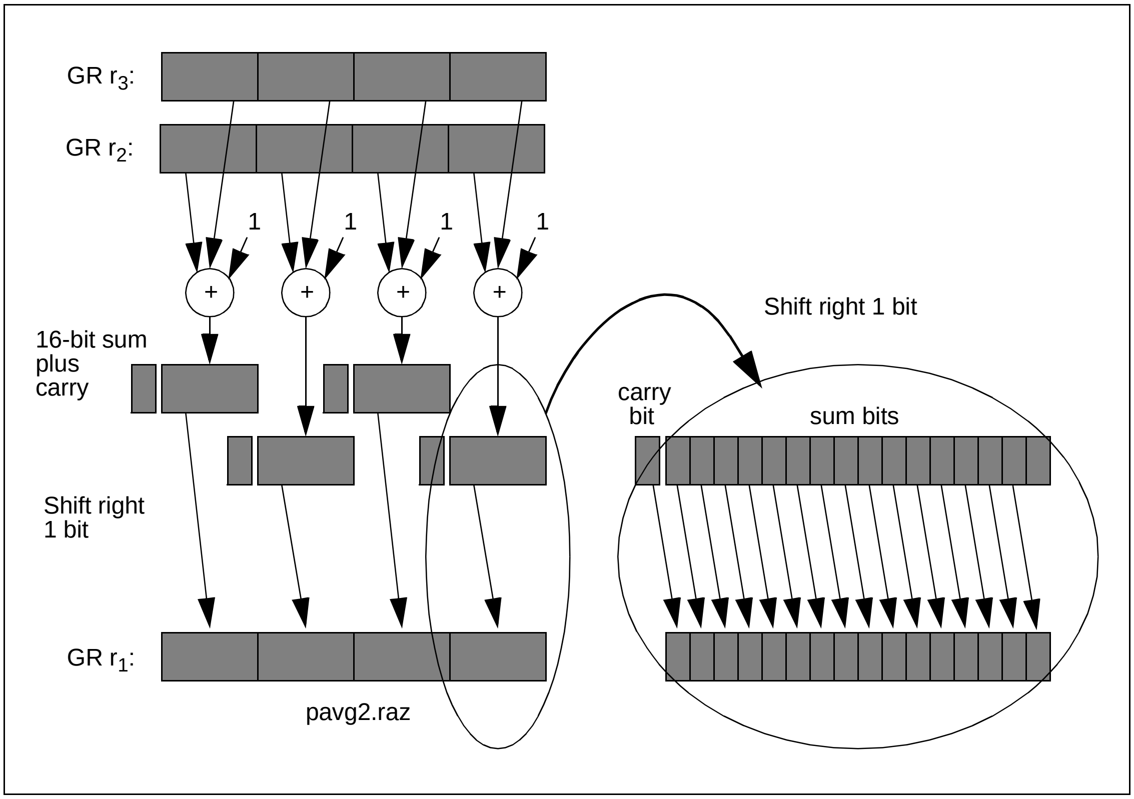


Figure 7-29. Parallel Average with Round Away from Zero Example



Operation:

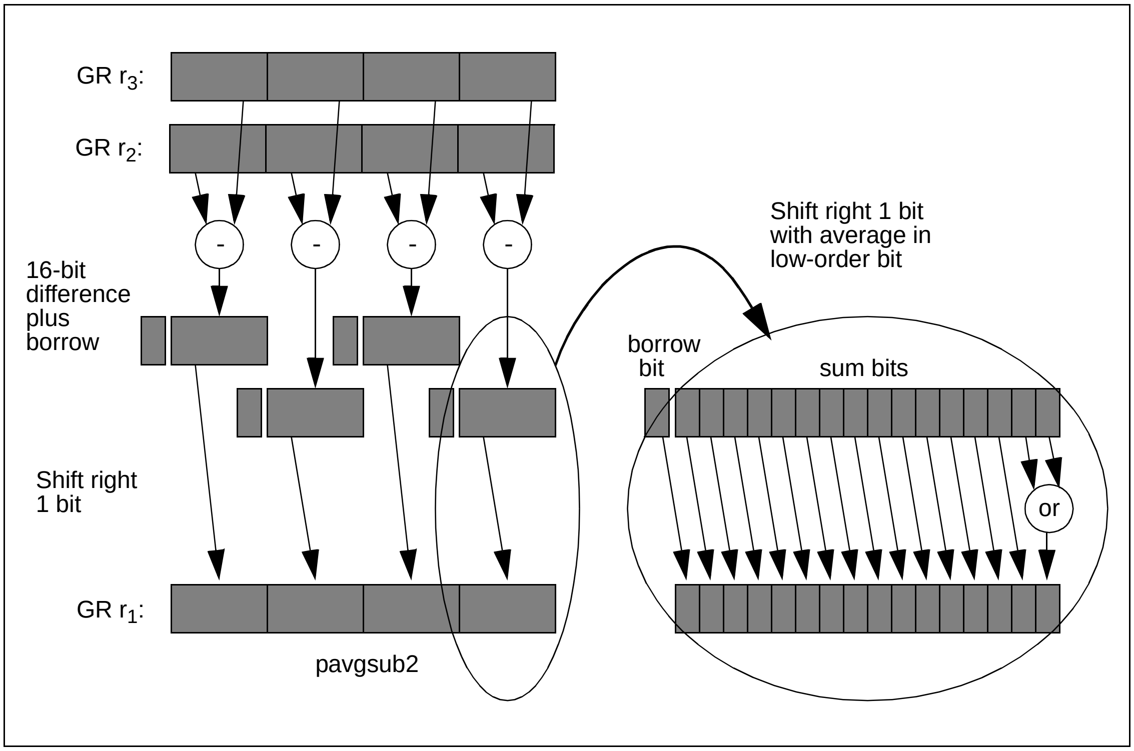
```
if (PR[qp]) {
    check_target_register($r_1$);

    if (one_byte_form) {                            // one_byte_form
        x[0] = GR[$r_2$]{7:0};     y[0] = GR[$r_3$]{7:0};
        x[1] = GR[$r_2$]{15:8};    y[1] = GR[$r_3$]{15:8};
        x[2] = GR[$r_2$]{23:16};   y[2] = GR[$r_3$]{23:16};
        x[3] = GR[$r_2$]{31:24};   y[3] = GR[$r_3$]{31:24};
        x[4] = GR[$r_2$]{39:32};   y[4] = GR[$r_3$]{39:32};
        x[5] = GR[$r_2$]{47:40};   y[5] = GR[$r_3$]{47:40};
        x[6] = GR[$r_2$]{55:48};   y[6] = GR[$r_3$]{55:48};
        x[7] = GR[$r_2$]{63:56};   y[7] = GR[$r_3$]{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[$r_1$] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);

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

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

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 7-30. Parallel Average Subtract Example


GR r3:
GR r2:
-
-
-
-
16-bit difference plus borrow
Shift right 1 bit
GR r1:
pavgsub2
Shift right 1 bit with average in low-order bit
borrow bit
sum bits
or

Operation:

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

   if (one_byte_form) {                            // 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;
}
```

Parallel Compare

Format:

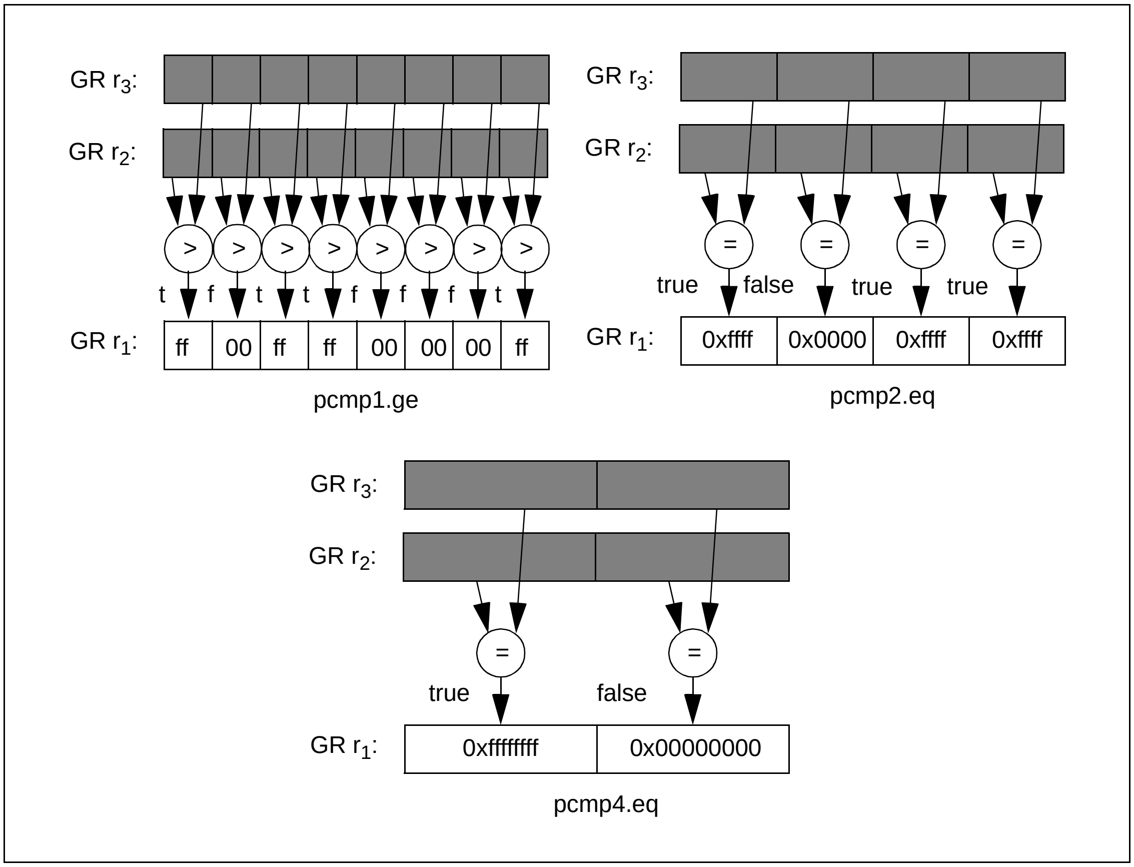
(qp) pcmp1.*prel* $r_1 = r_2, r_3$	one_byte_form	A9
(qp) pcmp2.*prel* $r_1 = r_2, r_3$	two_byte_form	A9
(qp) pcmp4.*prel* $r_1 = r_2, r_3$	four_byte_form	A9

Description: The two source operands are compared for one of the two relations shown in Table 7-38. 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, the corresponding data element in GR r_1 is set to all zeros. For the '>' relation, both operands are interpreted as signed.

Table 7-38. Pcmp Relations

prel	Compare Relation (r_2 *prel* r_3)
eq	$r_2 == r_3$
gt	$r_2 > r_3$ (signed)

Figure 7-31. Parallel Compare Example



Operation:

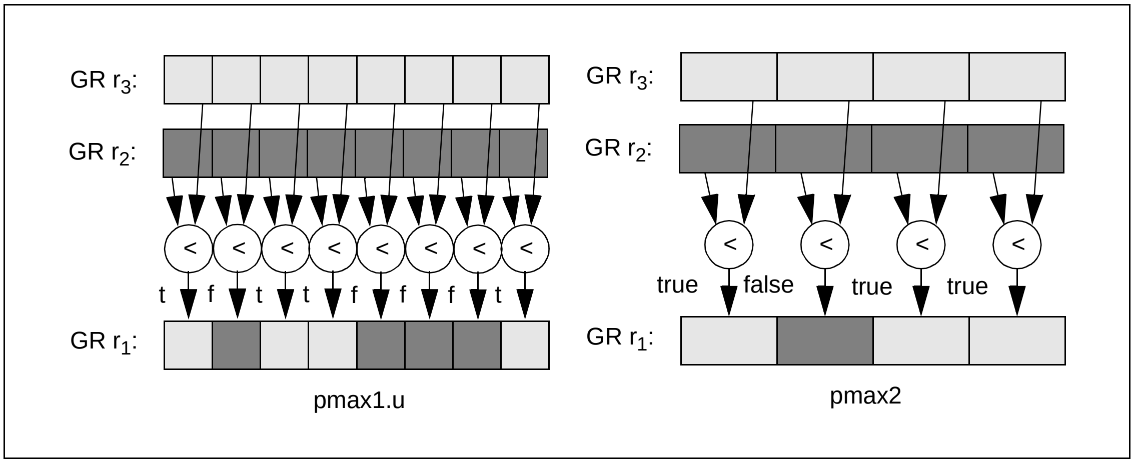
```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                   // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            if (prel == ‘eq’)
                tmp_rel = x[i] == y[i];
            else
                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
                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
                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;
}
```

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 7-32. Parallel Maximum Example



Operation:

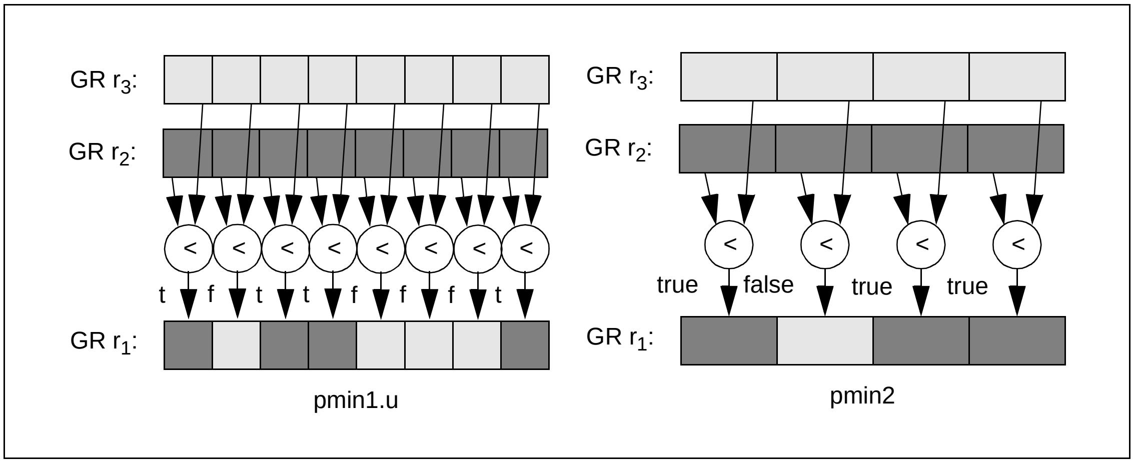
```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                        // one-byte elements
        x[0] = GR[r2]{7:0};       y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};      y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};     y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};     y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};     y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};     y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};     y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};     y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            res[i] = (zero_ext(x[i],8) < zero_ext(y[i],8)) ? y[i] : x[i];
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
    } else {                                                    // two-byte elements
        x[0] = GR[r2]{15:0};      y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};     y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};     y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};     y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            res[i] = (sign_ext(x[i],16) < sign_ext(y[i],16)) ? y[i] : x[i];
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

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 7-33. Parallel Minimum Example



Operation:

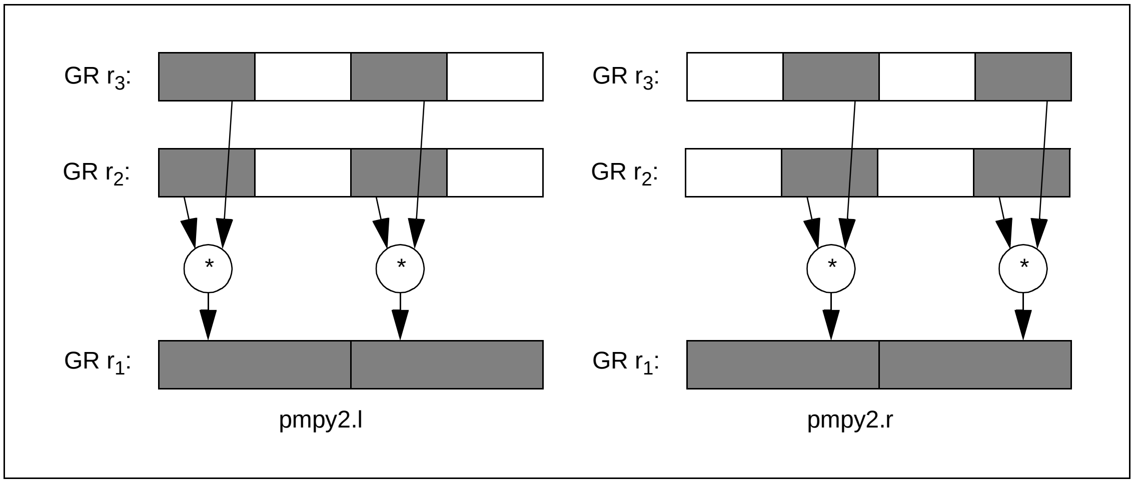
```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                        // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            res[i] = (zero_ext(x[i],8) < zero_ext(y[i],8)) ? x[i] : y[i];
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);
    } else {                                                    // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            res[i] = (sign_ext(x[i],16) < sign_ext(y[i],16)) ? x[i] : y[i];
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

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 7-34. The two 32-bit results are placed in GR r_1.

Figure 7-34. Parallel Multiply Operation



Operation:

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

    if (right_form) {
        GR[r1]{31:0} = sign_ext(GR[r2]{15:0}, 16) * sign_ext(GR[r3]{15:0}, 16);
        GR[r1]{63:32} = sign_ext(GR[r2]{47:32}, 16) * sign_ext(GR[r3]{47:32}, 16);
    } else {                                                            // left_form
        GR[r1]{31:0} = sign_ext(GR[r2]{31:16}, 16) * sign_ext(GR[r3]{31:16}, 16);
        GR[r1]{63:32} = sign_ext(GR[r2]{63:48}, 16) * sign_ext(GR[r3]{63:48}, 16);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Parallel Multiply and Shift Right

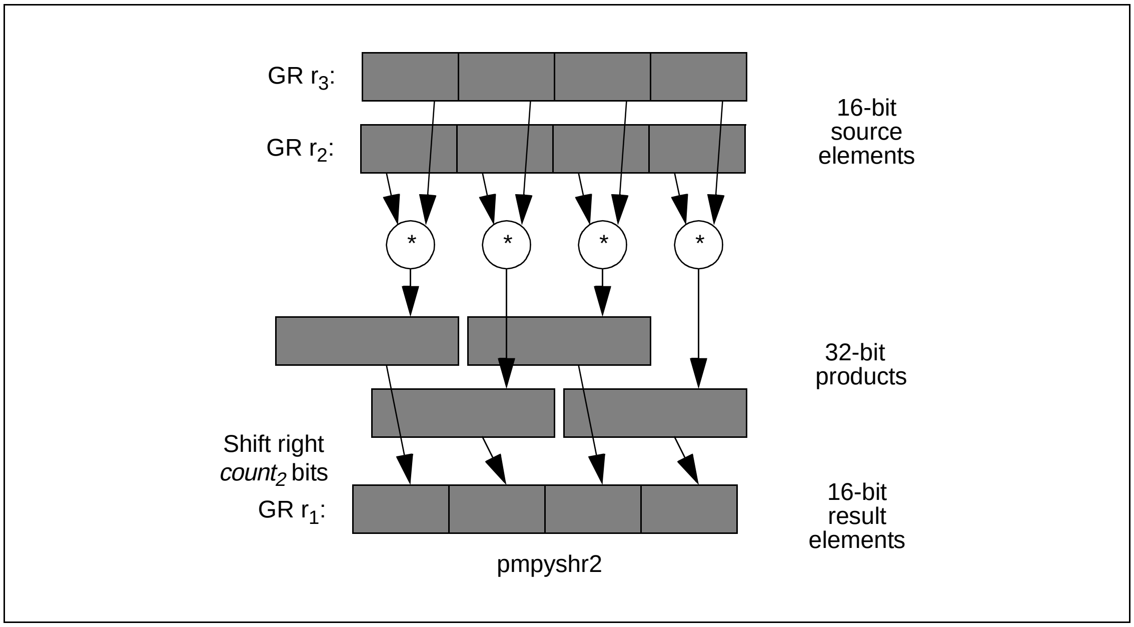
Format: (*qp*) pmpyshr2 $r_1 = r_2, r_3, count_2$ signed_form I1
(*qp*) pmpyshr2.u $r_1 = r_2, r_3, count_2$ unsigned_form I1

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 7-35. This multiplication can either be signed (pmpyshr2), or unsigned (pmpyshr2.u). Each product is then shifted to the right $count_2$ bits, and the least-significant 16-bits of each shifted product form 4 16-bit results, which are placed in GR r_1. A $count_2$ of 0 gives the 16 low bits of the results, a $count_2$ of 16 gives the 16 high bits of the results. The allowed values for $count_2$ are given in Table 7-39.

Table 7-39. PMPYSHR Shift Options

$count_2$	Selected Bit Field from each 32-bit Product
0	15:0
7	22:7
15	30:15
16	31:16

Figure 7-35. Parallel Multiply and Shift Right Operation



Operation:

```
if (PR[qp]) {
    check_target_register($r_1$);
    x[0] = GR[$r_2$]{15:0};     y[0] = GR[$r_3$]{15:0};
    x[1] = GR[$r_2$]{31:16};    y[1] = GR[$r_3$]{31:16};
    x[2] = GR[$r_2$]{47:32};    y[2] = GR[$r_3$]{47:32};
    x[3] = GR[$r_2$]{63:48};    y[3] = GR[$r_3$]{63:48};
    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]{($count_2$ + 15):$count_2$};
    }

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

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:

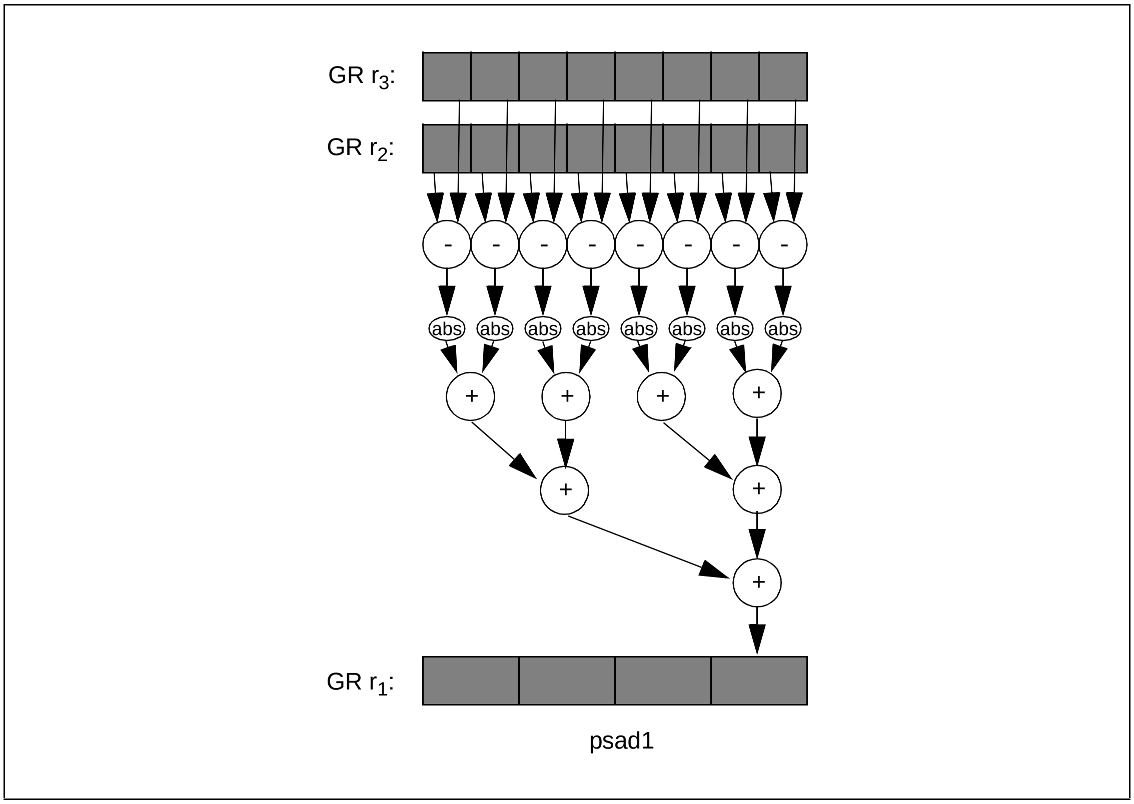
```
if (PR[qp]) {
    check_target_register(r1);

    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;
}
```

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 7-36. Parallel Sum of Absolute Difference Example



Operation:

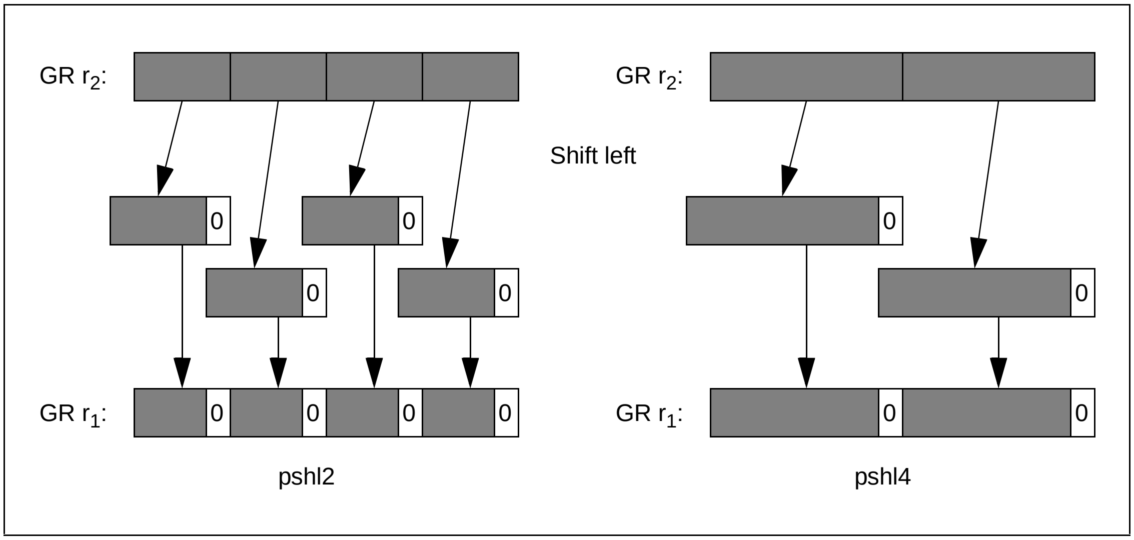
```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
    x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
    x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
    x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
    x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
    x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
    x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
    x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

    GR[r1] = 0;
    for (i = 0; i < 8; i++) {
        temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
        if (temp[i] < 0)
            temp[i] = -temp[i];
        GR[r1] += temp[i];
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

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 7-37. Parallel Shift Left Example



Operation:

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

    shift_count = (variable_form ? GR[r3] : count5);
    tmp_nat = (variable_form ? GR[r3].nat : 0);

    if (two_byte_form) {                                    // two_byte_form
        if (shift_count u> 16)
            shift_count = 16;
        GR[r1]{15:0}  = GR[r2]{15:0}  << shift_count;
        GR[r1]{31:16} = GR[r2]{31:16} << shift_count;
        GR[r1]{47:32} = GR[r2]{47:32} << shift_count;
        GR[r1]{63:48} = GR[r2]{63:48} << shift_count;
    } else {                                               // four_byte_form
        if (shift_count u> 32)
            shift_count = 32;
        GR[r1]{31:0}  = GR[r2]{31:0}  << shift_count;
        GR[r1]{63:32} = GR[r2]{63:32} << shift_count;
    }

    GR[r1].nat = GR[r2].nat || tmp_nat;
}
```

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;
}
```

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

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;
}
```

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 7-40. 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 7-41.

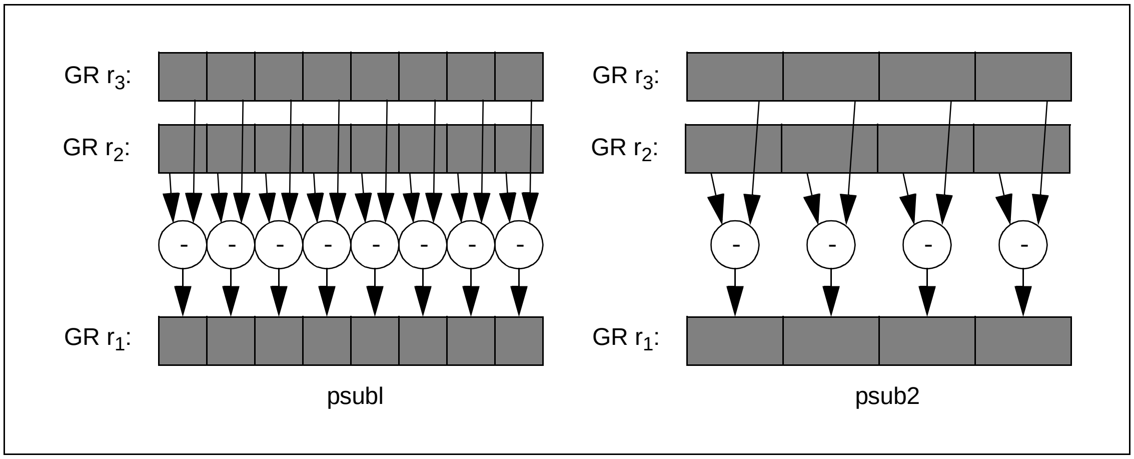
Table 7-40. Parallel Subtract Saturation Completers

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 7-41. 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 7-38. Parallel Subtract Example



Operation:

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

    if (one_byte_form) {                                        // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (sss_saturation_form) {                  // sss_saturation_form
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);
            for (i = 0; i < 8; i++) {
                temp[i] = sign_ext(x[i], 8) - sign_ext(y[i], 8);
            }
        } else if (uus_saturation_form) {           // uus_saturation_form
            max = 0xff;
            min = 0x00;
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - sign_ext(y[i], 8);
            }
        } else if (uuu_saturation_form) {           // uuu_saturation_form
            max = 0xff;
            min = 0x00;
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            }
        } else {                                    // modulo_form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            }
        }

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

        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                              temp[3], temp[2], temp[1], temp[0]);
    } else if (two_byte_form) {                                 // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (sss_saturation_form) {                  // sss_saturation_form
            max = sign_ext(0x7fff, 16);
            min = sign_ext(0x8000, 16);
            for (i = 0; i < 4; i++) {
                temp[i] = sign_ext(x[i], 16) - sign_ext(y[i], 16);
            }
        } else if (uus_saturation_form) {           // uus_saturation_form
            max = 0xffff;
            min = 0x0000;
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - sign_ext(y[i], 16);
            }
        } else if (uuu_saturation_form) {           // uuu_saturation_form
            max = 0xffff;
            min = 0x0000;
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            }
        } else {                                    // modulo_form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            }
        }

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

        GR[$r_1$] = concatenate4(temp[3], temp[2], temp[1], temp[0]);
    } else {                                                  // four-byte elements
        x[0] = GR[$r_2$]{31:0};     y[0] = GR[$r_3$]{31:0};
        x[1] = GR[$r_2$]{63:32};    y[1] = GR[$r_3$]{63:32};

        for (i = 0; i < 2; i++) {                                    // modulo_form
            temp[i] = zero_ext(x[i], 32) - zero_ext(y[i], 32);
        }

        GR[$r_1$] = concatenate2(temp[1], temp[0]);
    }

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

Reset User Mask

Format: (*qp*) rum imm_{24} M44

Description: The complement of the imm_{24} operand is ANDed with the user mask (PSR{5:0}) and the result is placed in the user mask.

PSR.up is only cleared if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Operation:

```
if (PR[qp]) {
    if (is_reserved_field(PSR_TYPE, PSR_UM, $imm_{24}$))
        reserved_register_field_fault();

    if ($imm_{24}${1})      PSR{1} = 0;
    if ($imm_{24}${2} && PSR.sp == 0)       //non-secure perf monitor
                            PSR{2} = 0;
    if ($imm_{24}${3})      PSR{3} = 0;
    if ($imm_{24}${4})      PSR{4} = 0;
    if ($imm_{24}${5})      PSR{5} = 0;
}
```

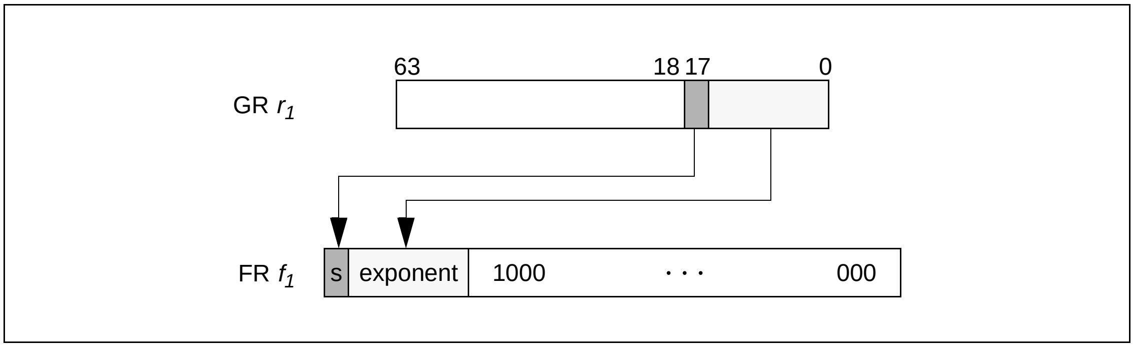
Set Floating-Point Value, Exponent, or Significand

Format:	(*qp*) setf.s $f_1 = r_2$	single_form	M18
	(*qp*) setf.d $f_1 = r_2$	double_form	M18
	(*qp*) setf.exp $f_1 = r_2$	exponent_form	M18
	(*qp*) setf.sig $f_1 = r_2$	significand_form	M18

Description: In the single and double forms, GR r_2 is treated as a single precision (in the single_form) or double precision (in the double_form) memory representation, converted into floating-point register format, and placed in FR f_1.

In the exponent_form, bits 16:0 of GR r_2 are copied to the exponent field of FR f_1 and bit 17 of GR r_2 is copied to the sign bit of FR f_1. The significand field of FR f_1 is set to one (0x800...000).

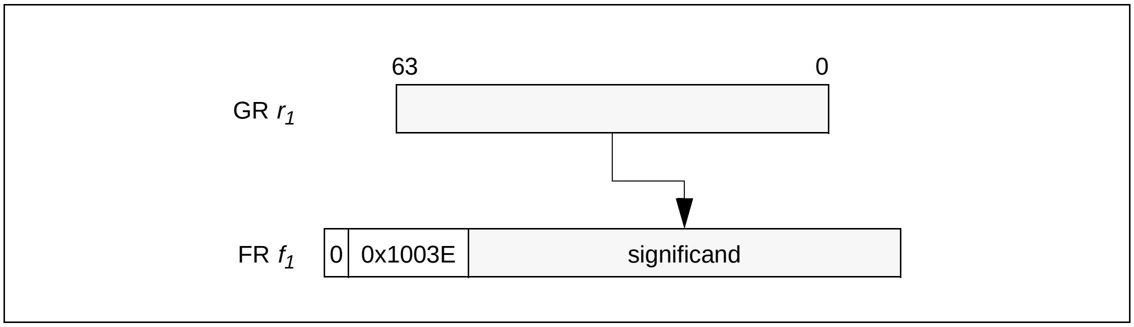
Figure 7-39. Function of setf.exp



In the significand_form, the value in GR r_2 is copied to the significand field of FR f_1.

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

Figure 7-40. Function of setf.sig



For all forms, if the NaT bit corresponding to r_2 is equal to 1, FR f_1 is set to NaTVal instead of the computed result.

Operation:

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

    if (!GR[r2].nat) {
        if (single_form)
            FR[f1] = fp_mem_to_fr_format(GR[r2], 4, 0);
        else if (double_form)
            FR[f1] = fp_mem_to_fr_format(GR[r2], 8, 0);
        else if (significand_form) {
            FR[f1].significand = GR[r2];
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = 0;
        } else {                                                   // exponent_form
            FR[f1].significand = 0x8000000000000000;
            FR[f1].exp = GR[r2]{16:0};
            FR[f1].sign = GR[r2]{17};
        }
    } else
        FR[f1] = NATVAL;

    fp_update_psr(f1);
}
```

Shift Left

Format: (qp) shl $r_1 = r_2, r_3$ I7
(qp) shl $r_1 = r_2, count_6$ pseudo-op of: (qp) dep.z $r_1 = r_2, count_6, 64–count_6$

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

For the immediate form, See "Deposit" on page 7-27.

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;
}
```

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:

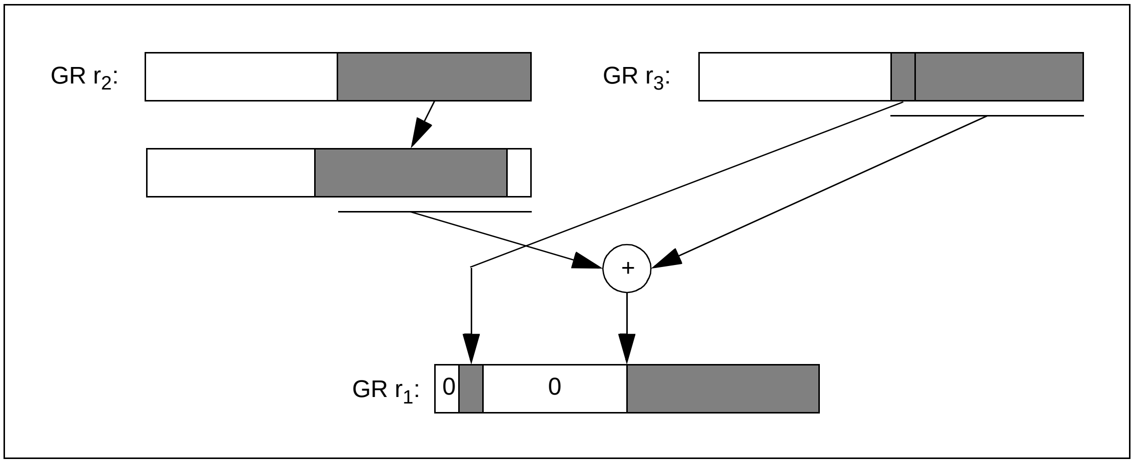
```
if (PR[qp]) {
      check_target_register($r_1$);

      GR[$r_1$] = (GR[$r_2$] << $count_2$) + GR[$r_3$];
      GR[$r_1$].nat = GR[$r_2$].nat || GR[$r_3$].nat;
}
```

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 7-41. Shift Left and Add Pointer



Operation:

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

    tmp_res = (GR[r2] << count2) + GR[r3];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[r3]{31:30};
    GR[r1] = tmp_res;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Shift Right

Format:

(*qp*) shr $r_1 = r_3, r_2$	signed_form	I5
(*qp*) shr.u $r_1 = r_3, r_2$	unsigned_form	I5
(*qp*) shr $r_1 = r_3, count_6$	pseudo-op of: (*qp*) extr $r_1 = r_3, count_6, 64–count_6$	
(*qp*) shr.u $r_1 = r_3, count_6$	pseudo-op of: (*qp*) extr.u $r_1 = r_3, count_6, 64–count_6$	

Description: The value in GR r_3 is shifted to the right and placed in GR r_1. In the signed_form the vacated bit positions are filled with bit 63 of GR r_3; in the unsigned_form the vacated bit positions are filled with zeroes. The number of bit positions to shift is specified by the value in GR r_2 or by an immediate value $count_6$. The shift count is interpreted as an unsigned number. If the value in GR r_2 is greater than 63, then the result is all zeroes (for the unsigned_form, or if bit 63 of GR r_3 was 0) or all ones (for the signed_form if bit 63 of GR r_3 was 1).

If the .u completer is specified, the shift is unsigned (logical), otherwise it is signed (arithmetic).

For the immediate forms, See “Extract” on page 7-29.

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;
}
```

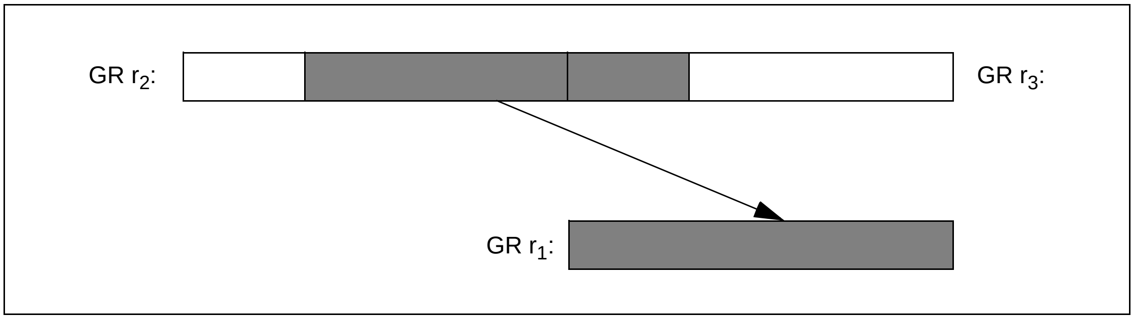
Shift Right Pair

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 7-42. Shift Right Pair



Operation:

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

    temp1 = shift_right_unsigned(GR[r3], count6);
    temp2 = GR[r2] << (64 - count6);
    GR[r1] = zero_ext(temp1, 64 - count6) | temp2;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Serialize

Format: (*qp*) srlz.i 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`.

Operation:

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

Store

Format:	(qp) st*sz*.*sttype*.*sthint* [r_3] = r_2	normal_form, no_base_update_form	M4
	(qp) st*sz*.*sttype*.*sthint* [r_3] = r_2, imm_9	normal_form, imm_base_update_form	M5
	(qp) st8.spill.*sthint* [r_3] = r_2	spill_form, no_base_update_form	M4
	(qp) st8.spill.*sthint* [r_3] = r_2, imm_9	spill_form, imm_base_update_form	M5

Description: A value consisting of the least significant *sz* bytes of the value in GR r_2 is written to memory starting at the address specified by the value in GR r_3. The values of the *sz* completer are given in Table 7-26 on page 7-104. The *sttype* completer specifies special store operations, which are described in Table 7-42. If the NaT bit corresponding to GR r_3 is 1 (or in the normal_form, if the NaT bit corresponding to GR r_2 is 1), a Register NaT Consumption fault is taken.

In the spill_form, an 8-byte value is stored, and the NaT bit corresponding to GR r_2 is copied to a bit in the UNAT application register. This instruction is used for spilling a register/NaT pair. See "Control Speculation" on page 4-13 for details.

In the imm_base_update form, the value in GR r_3 is added to a signed immediate value (imm_9) and the result is placed back in GR r_3. This base register update is done after the store, and does not affect the store address, nor the value stored (for the case where r_2 and r_3 specify the same register).

Table 7-42. 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 "Memory Access Ordering" on page 4-23 .

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 7-43. See "Memory Hierarchy Control and Consistency" on page 4-20.

Table 7-43. Store Hints

sthint Completer	Interpretation
none	Temporal locality, level 1
nta	Non-temporal locality, all levels

Operation:

```
if (PR[qp]) {
    size = spill_form ? 8 : sz;
    otype = (sttype == ‘rel’) ? RELEASE : UNORDERED;

    if (imm_base_update_form)
        check_target_register($r_3$);
    if (GR[$r_3$].nat || (normal_form && GR[$r_2$].nat))
        register_nat_consumption_fault(WRITE);

    paddr = tlb_translate(GR[$r_3$], size, WRITE, PSR.cpl, &mattr,
                            &tmp_unused);
    if (spill_form && GR[$r_2$].nat)
        natd_gr_write(GR[$r_2$], paddr, size, UM.be, mattr, otype, sthint);
    else
        mem_write(GR[$r_2$], paddr, size, UM.be, mattr, otype, sthint);

    if (spill_form) {
        bit_pos = GR[$r_3$]{8:3};
        AR[UNAT]{bit_pos} = GR[$r_2$].nat;
    }

    alat_inval_multiple_entries(paddr, size);

    if (imm_base_update_form) {
        GR[$r_3$] = GR[$r_3$] + sign_ext($imm_9$, 9);
        GR[$r_3$].nat = 0;
    }
}
```

Floating-Point Store

Format:	(*qp*) stf*fsz*.*sthint* [r_3] = f_2	normal_form, no_base_update_form	M9
	(*qp*) stf*fsz*.*sthint* [r_3] = f_2, imm_9	normal_form, imm_base_update_form	M10
	(*qp*) stf8.*sthint* [r_3] = f_2	integer_form, no_base_update_form	M9
	(*qp*) stf8.*sthint* [r_3] = f_2, imm_9	integer_form, imm_base_update_form	M10
	(*qp*) stf.spill.*sthint* [r_3] = f_2	spill_form, no_base_update_form	M9
	(*qp*) stf.spill.*sthint* [r_3] = f_2, imm_9	spill_form, imm_base_update_form	M10

Description: A value, consisting of *fsz* bytes, is generated from the value in FR f_2 and written to memory starting at the address specified by the value in GR r_3. In the normal_form, the value in FR f_2 is converted to the memory format and then stored. In the integer_form, the significand of FR f_2 is stored. The values of the *fsz* completer are given in Table 7-29 on page 7-108. 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 "Data Types and Formats" on page 5-1 for details on conversion from floating-point register format.

In the spill_form, a 16-byte value from FR f_2 is stored without conversion. This instruction is used for spilling a register. See "Control Speculation" on page 4-13 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 7-43 on page 7-171. See "Memory Hierarchy Control and Consistency" on page 4-20.

Operation:

```
if (PR[qp]) {
    if (imm_base_update_form)
        check_target_register($r_3$);
    if (tmp_isrcode = fp_reg_disabled($f_2$, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, WRITE);

    if (GR[$r_3$].nat || (!spill_form && (FR[$f_2$] == NATVAL)))
        register_nat_consumption_fault(WRITE);

    size = spill_form ? 16 : (integer_form ? 8 : fsz);

    paddr = tlb_translate(GR[$r_3$], size, WRITE, PSR.cpl, &mattr, &tmp_unused);
    val = fp_fr_to_mem_format(FR[$f_2$], 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[$r_3$] = GR[$r_3$] + sign_ext($imm_9$, 9);
        GR[$r_3$].nat = 0;
    }
}
```

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($r_1$);

    tmp_src = (register_form ? GR[$r_2$] : sign_ext($imm_8$, 8));
    tmp_nat = (register_form ? GR[$r_2$].nat : 0);

    if (minus1_form)
        GR[$r_1$] = tmp_src - GR[$r_3$] - 1;
    else
        GR[$r_1$] = tmp_src - GR[$r_3$];

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

Set User Mask

Format: (*qp*) sum *imm*$_{24}$ M44

Description: The *imm*$_{24}$ operand is ORed with the user mask (PSR{5:0}) and the result is placed in the user mask.

PSR.up can only be set if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Operation:

```
if (PR[qp]) {
    if (is_reserved_field(PSR_TYPE, PSR_UM, $imm_{24}$))
        reserved_register_field_fault();

    if ($imm_{24}${1})        PSR{1} = 1;
    if ($imm_{24}${2} && PSR.sp == 0)       //non-secure perf monitor
                              PSR{2} = 1;
    if ($imm_{24}${3})        PSR{3} = 1;
    if ($imm_{24}${4})        PSR{4} = 1;
    if ($imm_{24}${5})        PSR{5} = 1;
}
```

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 7-44.

Table 7-44. *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;
}
```

Memory Synchronization

Format: (*qp*) sync.i M24

Description: `sync.i` ensures that when previously initiated Flush Cache (`fc`) operations issued by the local processor become visible to local data memory references, prior Flush Cache operations are also observed by the local processor instruction fetch stream. `sync.i` also ensures that at the time previously initiated Flush Cache (`fc`) operations are observed on a remote processor by data memory references they are also observed by instruction memory references on the remote processor. `sync.i` is ordered with respect to all cache flush operations as observed by another processor. A `sync.i` and a previous `fc` must be in separate instruction groups. If semantically required, the programmer must explicitly insert ordered data references (acquire, release or fence type) to appropriately constrain `sync.i` (and hence `fc`) visibility to the data stream on other processors.

`sync.i` is used to maintain an ordering relationship between instruction and data caches on local and remote processors. An instruction serialize operation be used to ensure synchronization initiated by `sync.i` on the local processor has been observed by a given point in program execution.

An example of self-modifying code (local processor):

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

Operation:

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

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 7-10 on page 7-19.

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 7-45. 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 7-46. 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;
    }
}
```

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 7-10 on page 7-19.

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 7-47. 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 7-48. 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:

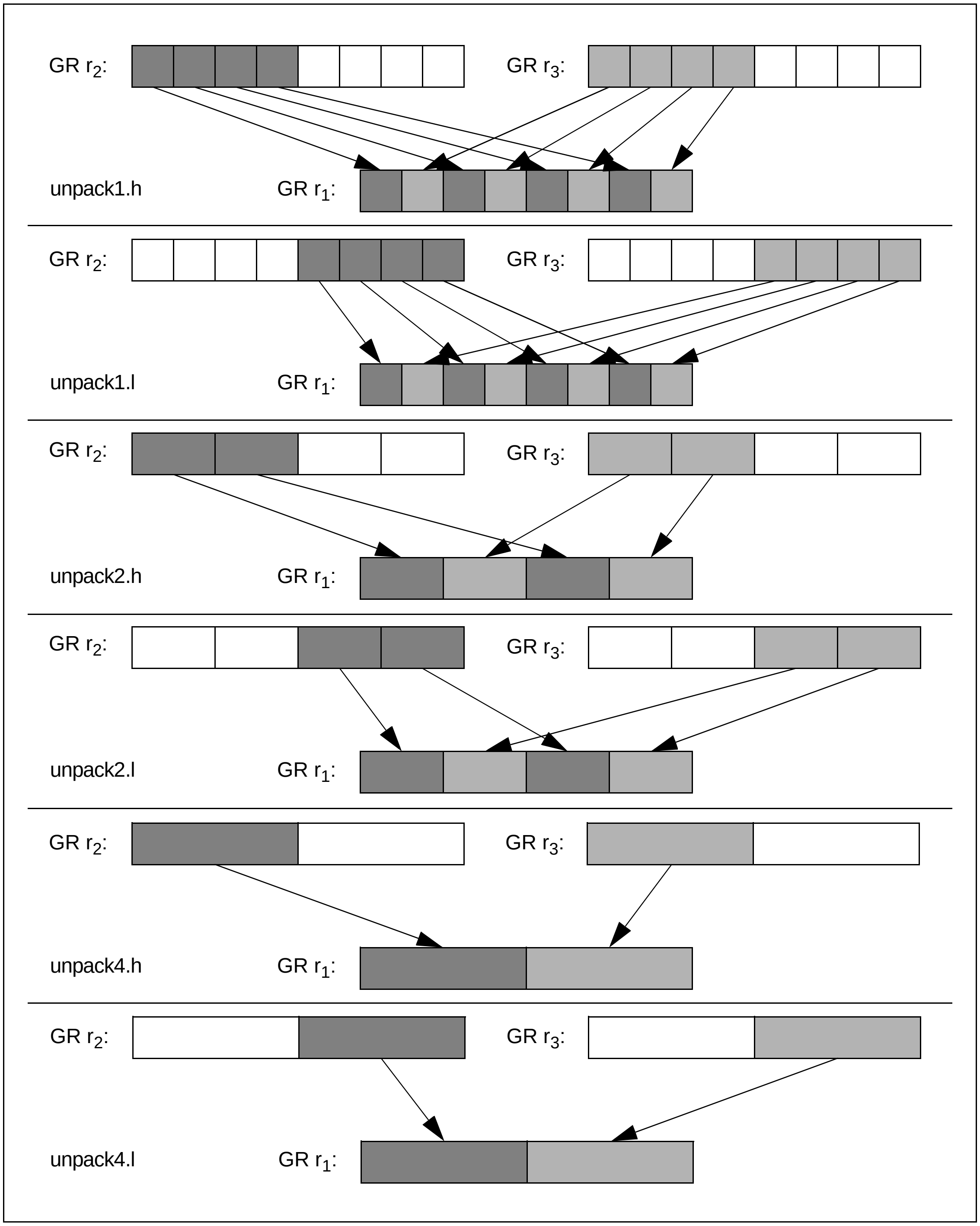
```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    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;
    }
}
```

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 7-43. Unpack Operation



Operation:

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

    if (one_byte_form) {                                          // one-byte elements
        x[0] = GR[$r_2$]{7:0};      y[0] = GR[$r_3$]{7:0};
        x[1] = GR[$r_2$]{15:8};     y[1] = GR[$r_3$]{15:8};
        x[2] = GR[$r_2$]{23:16};    y[2] = GR[$r_3$]{23:16};
        x[3] = GR[$r_2$]{31:24};    y[3] = GR[$r_3$]{31:24};
        x[4] = GR[$r_2$]{39:32};    y[4] = GR[$r_3$]{39:32};
        x[5] = GR[$r_2$]{47:40};    y[5] = GR[$r_3$]{47:40};
        x[6] = GR[$r_2$]{55:48};    y[6] = GR[$r_3$]{55:48};
        x[7] = GR[$r_2$]{63:56};    y[7] = GR[$r_3$]{63:56};

        if (high_form)
            GR[$r_1$] = concatenate8(  x[7], y[7], x[6], y[6],
                                     x[5], y[5], x[4], y[4]);
        else
            GR[$r_1$] = 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[$r_2$]{15:0};     y[0] = GR[$r_3$]{15:0};
        x[1] = GR[$r_2$]{31:16};    y[1] = GR[$r_3$]{31:16};
        x[2] = GR[$r_2$]{47:32};    y[2] = GR[$r_3$]{47:32};
        x[3] = GR[$r_2$]{63:48};    y[3] = GR[$r_3$]{63:48};

        if (high_form)
            GR[$r_1$] = concatenate4(x[3], y[3], x[2], y[2]);
        else
            GR[$r_1$] = concatenate4(x[1], y[1], x[0], y[0]);
    } else {                                                     // four-byte elements
        x[0] = GR[$r_2$]{31:0};     y[0] = GR[$r_3$]{31:0};
        x[1] = GR[$r_2$]{63:32};    y[1] = GR[$r_3$]{63:32};

        if (high_form)
            GR[$r_1$] = concatenate2(x[1], y[1]);
        else
            GR[$r_1$] = concatenate2(x[0], y[0]);
    }
    GR[$r_1$].nat = GR[$r_2$].nat || GR[$r_3$].nat;
}
```

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 7-49.

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 7-49. 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.

The memory read and write are guaranteed to be atomic.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 7-28 on page 7-105. Locality hints do not affect program functionality and may be ignored by the implementation. See "Memory Hierarchy Control and Consistency" on page 4-20 for details.

Operation:

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

    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;
}
```

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);
}
```

Fixed-Point Multiply

Format:

(*qp*) xmpy.l $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.l $f_1 = f_3, f_4$, f0
(*qp*) xmpy.lu $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.l $f_1 = f_3, f_4$, f0
(*qp*) xmpy.h $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.h $f_1 = f_3, f_{4,}$ f0
(*qp*) xmpy.hu $f_1 = f_3, f_4$	pseudo-op of: (*qp*) xma.hu $f_1 = f_3, f_4$, f0

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

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

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

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

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

Note: f1 as an operand is not an integer 1; it is just the register file format's 1.0 value.

Operation: `See “Fixed-Point Multiply Add” on page 7-186.`

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($r_1$);

    tmp_src = (register_form ? GR[$r_2$] : sign_ext($imm_8$, 8));
    tmp_nat = (register_form ? GR[$r_2$].nat : 0);

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

Zero Extend

Format: (qp) zxtxsz $r_1 = r_3$ 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 7-44 on page 7-176.

Operation:

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

    GR[r1] = zero_ext(GR[r3],xsz * 8);
    GR[r1].nat = GR[r3].nat;
}
```

Part II: IA-64 Optimization Guide

About the IA-64 Optimization Guide 8

The second portion of this document explains in detail optimization techniques associated with the IA-64 instruction set. It is intended for those interested in furthering their understanding of IA-64 application architecture features and optimization techniques that benefit application performance. Intel and the industry are developing compilers to take advantage of these techniques. Application developers are not advised to use this as a guide to IA-64 assembly language programming.

Note: **To demonstrate techniques, this guide contains code examples that are not targeted towards a specific IA-64 processor, but rather a hypothetical implementation. For these code examples, ALU operations are assumed to take one cycle and loads take two cycles to return from first level cache and that there are two load/store execution units and four ALUs. Other latencies and execution unit details are described as needed in the text. This guide will refer to this model as the "generic" implementation**.

8.1 Overview of the IA-64 Optimization Guide

Chapter 9, "Introduction to IA-64 Programming". Provides an overview of the IA-64 application programming environment.

Chapter 10, "Memory Reference". Discusses features and optimizations related to control and data speculation.

Chapter 11, "Predication, Control Flow, and Instruction Stream". Describes optimization features related to predication, control flow, and branch hints.

Chapter 12, "Software Pipelining and Loop Support". Provides a detailed discussion on optimizing loops through use of software pipelining.

Chapter 13, "Floating-point Applications". Discusses current performance limitations in floating-point applications and IA-64 features that address these limitations.

Introduction to IA-64 Programming 9

9.1 Overview

The IA-64 instruction set is designed to allow the compiler to communicate information to the processor to manage resource characteristics such as instruction latency, issue width, and functional unit assignment. Although such resources can be statically scheduled, IA-64 does not require that code be written for a specific microarchitecture implementation in order to be functional.

IA-64 includes a complete instruction set with new features designed to:

- Increase instruction-level parallelism (ILP).
- Better manage memory latencies.
- Improve branch handling and management of branch resources.
- Reduce procedure call overhead.

IA-64 also enables high floating-point performance and provides direct support for multimedia applications.

Complete descriptions of the syntax and semantics of IA-64 instructions can be found in *Part I: IA-64 Application Architecture Guide*. Though this chapter provides a high level introduction to application level IA-64 programming, it assumes prior experience with assembly language programming as well as some familiarity with the IA-64 application architecture. Optimization is explored in other chapters of this guide.

9.2 Registers

IA-64 architecture defines 128 general purpose registers, 128 floating-point registers, 64 predicate registers, and up to 128 special purpose registers. The large number of architectural registers in IA-64 enable multiple computations to be performed without having to frequently spill and fill intermediate data to memory.

There are 128, 64-bit **general purpose registers** (`r0-r127`) that are used to hold values for integer and multimedia computations. Each of the 128 registers has one additional NaT (Not a Thing) bit which is used to indicate whether the value stored in the register is valid. Execution of IA-64 speculative instructions can result in a register's NaT bit being set. Register `r0` is read-only and contains a value of zero (0). Attempting to write to `r0` will cause a fault.

There are 128, 82-bit **floating-point registers** (`f0-f127`) that are used for floating-point computations. The first two registers, `f0` and `f1`, are read-only and read as +0.0 and +1.0, respectively. Instructions that write to `f0` or `f1` will fault.

There are 64, one-bit **predicate registers** (`p0-p63`) that control conditional execution of instructions and conditional branches. The first register, `p0`, is read-only and always reads true (1). The results of instructions that write to `p0` are discarded.

There are 8, 64-bit **branch registers** (`b0-b7`) that are used to specify the target addresses of indirect branches.

There is space for up to 128 **application registers** (`ar0-ar127`) that support various functions. Many of these register slots are reserved for future use. Some application registers have assembler aliases. For example, `ar66` is the Epilogue Counter and is called `ar.ec`.

The **instruction pointer** is a 64-bit register that points to the currently executing instruction bundle.

9.3 Using IA-64 Instructions

IA-64 instructions are grouped into 128-bit *bundles* of three instructions. Each instruction occupies the first, second, or third *syllable* of a bundle.   Instruction format, expression of parallelism, and bundle specification are described below.

9.3.1 Format

A basic IA-64 instruction has the following syntax:

[*qp*] *mnemonic*[.*comp*]   *dest*=*srcs*

where:

qp — Specifies a qualifying predicate register. The value of the qualifying predicate determines whether the results of the instruction are committed in hardware or discarded. When the value of the predicate register is true (1), the instruction executes, its results are committed, and any exceptions that occur are handled as usual. When the value is false (0), the results are not committed and no exceptions are raised. Most IA-64 instructions can be accompanied by a qualifying predicate.

mnemonic — Specifies a name that uniquely identifies an IA-64 instruction.

comp — Specifies one or more instruction completers. Completers indicate optional variations on a base instruction mnemonic. Completers follow the mnemonic and are separated by periods.

dest — Represents the destination operand(s), which is typically the result value(s) produced by an instruction.

srcs — Represents the source operands. Most IA-64 instructions have at least two input source operands.

9.3.2 Expressing Parallelism

IA-64 requires the compiler or assembly writer to explicitly indicate groups of instructions, called *instruction groups*, that have no register read after write (RAW) or write after write (WAW) register dependences. Instruction groups are delimited by *stops* in the assembly source code. Since instruction groups have no RAW or WAW register dependences, they can be issued without hardware checks for register dependences between instructions. Both of the examples below show two instruction groups separated by stops (indicated by double semicolons):

```
ld8 r1=[r5] ;; // First group
add r3=r1,r4   // Second group
```

A more complex example with multiple register flow dependences is shown below:

```
ld8 r1=[r5]    // First group
sub r6=r8,r9 ;;// First group
add r3=r1,r4   // Second group
st8 [r6]=r12   // Second group
```

All instructions in a single instruction group may not necessarily issue in parallel because specific IA-64 implementations may not have sufficient resources to issue all instructions in an instruction group.

9.3.3 Bundles and Templates

In assembly code, each 128-bit bundle is enclosed in curly braces and contains a template specification and three instructions. Thus, a stop may be specified at the end of any bundle or in the middle of a bundle by using one of two special template types that implicitly include mid-bundle stops.

Each instruction in a bundle is 41-bits long. Five other bits are used by a template-type specification. Bundle templates enable IA-64 processors to dispatch instructions with simple instruction decoding, and stops enable explicit specification of parallelism.

There are five IA-64 syllable types (M, I, F, B, and L), six IA-64 instruction types (M, I, A, F, B, L), and 12 basic template types (MII, MI_I, MLX, MMI, M_MI, MFI, MMF, MIB, MBB, BBB, MMB, MFB). Each basic template type has two versions: one with a stop after the third syllable and one without. Instructions must be placed in syllables corresponding to their instruction types based on the template specification, except for A-type instructions that can go in either I or M syllables. For example, a template specification of `.MII` means that of the three instructions in a bundle, the first is a memory (`M`) or A-type instruction, and the next two are ALU integer (`I`) or A-type instructions:

```
{ .mii
    ld4  r28=[r8]  //  Load a 4-byte value
    add  r9=2,r1   //  2+r1 and put in r9
    add  r30=1,r1  //  1+r1 and put in r30
}
```

For readability, most code examples in this book do not specify templates or braces.

Note: Bundle boundaries have no direct correlation with instruction group boundaries as instruction groups can extend over an arbitrary number of bundles. Instruction groups begin and end where stops are set in assembly code, and dynamically whenever a branch is taken or a stop is encountered.

9.4 Memory Access and Speculation

IA-64 provides memory access only through register load and store instructions and special semaphore instructions. IA-64 also provides extensive support for hiding memory latency via programmer-controlled speculation.

9.4.1 Functionality

Data and instructions are referenced by 64-bit addresses. Instructions are stored in memory in little endian byte order, in which the *least* significant byte appears in the lowest addressed byte of a memory location. For data, modes for both big and little endian byte order are supported and can be controlled by a bit in the User Mask Register.

Integer loads of one, two, and four bytes are zero-extended, since all 64-bits of each register are always written. Integer stores write one, two, four, or eight bytes of registers to memory as specified.

9.4.2 Speculation

Speculation allows a programmer to break data or control dependences that would normally limit code motion. The two kinds of speculation are called control speculation and data speculation. This section summarizes IA-64 speculation. See Chapter 10, "Memory Reference" for more detailed descriptions of speculative instruction behavior and application.

9.4.3 Control Speculation

Control speculation allows loads and their dependent uses to be safely moved above branches. Support for this is enabled by special NaT bits that are attached to integer registers and by special NatVal values for floating-point registers. When a speculative load causes an exception, it is not immediately raised. Instead, the NaT bit is set on the destination register (or NatVal is written into the floating-point register). Subsequent speculative instructions that use a register with a set NaT bit propagate the setting until a non-speculative instruction checks for or raises the deferred exception.

For example, in the absence of other information, the compiler for a typical RISC architecture cannot safely move the load above the branch in the sequence below:

```
(p1) br.cond.dptk L1       // Cycle 0
     ld8 r3=[r5] ;;        // Cycle 1
     shr r7=r3,r87         // Cycle 3
```

Supposing that the latency of a load is 2 cycles, the shift right (`shr`) instruction will stall for 1. However, by using the speculative loads and checks provided in IA-64, two cycles can be saved by rewriting the above code as shown below:

```
     ld8.s r3=[r5]         // Earlier cycle
     // Other instructions

(p1) br.cond.dptk L1 ;;    // Cycle 0
     chk.s r3,recovery     // Cycle 1
     shr   r7=r3,r87       // Cycle 1
```

This code assumes `r5` is ready when accessed and that there are sufficient instructions to fill the latency between the `ld8.s` and the `chk.s`.

9.4.4 Data Speculation

Data speculation allows loads to be moved above possibly conflicting memory references. *Advanced loads* exclusively refer to data speculative loads. Review the order of loads and stores in this IA-64 assembly sequence:

```
st8 [r55]=r45  // Cycle 0
ld8 r3=[r5] ;; // Cycle 0
shr r7=r3,r87  // Cycle 2
```

IA-64 allows the programmer to move the load above the store even if it is not known whether the load and the store reference overlapping memory locations. This is accomplished using special advanced load and check instructions:

```
ld8.a r3=[r5]  // Advanced load
// Other instructions

st8 [r55]=r45  // Cycle 0
ld8.c r3=[r5]  // Cycle 0 - check
shr r7=r3,r87  // Cycle 0
```

Note: The `shr` instruction in this schedule could issue in cycle 0 if there were no conflicts between the advanced load and intervening stores. If there were a conflict, the check load instruction (`ld8.c`) would detect the conflict and reissue the load.

9.5 Predication

Predication is the conditional execution of an instruction based on a qualifying predicate. A qualifying predicate is a predicate register whose value determines whether the processor commits the results computed by an instruction.

The values of predicate registers are set by the results of instructions such as compare (`cmp`) and test bit (`tbit`). When the value of a qualifying predicate associated with an instruction is true (1), the processor executes the instruction, and instruction results are committed. When the value is false (0), the processor discards any results and raises no exceptions. Consider the following C code:

```
if (a) {
    b = c + d;
}
if (e) {
    h = i + j;
}
```

This code can be implemented in IA-64 using qualifying predicates so that branches are removed. The IA-64 pseudo-code shown below implements the C expressions without branches:

```
    cmp.ne  p1,p2=a,r0    // p1 <- a != 0
    cmp.ne  p3,p4=e,r0 ;; // p3 <- e != 0
(p1)add    b=c,d          // If a != 0 then add
(p3)sub    h=i,j          // If e != 0 then sub
```

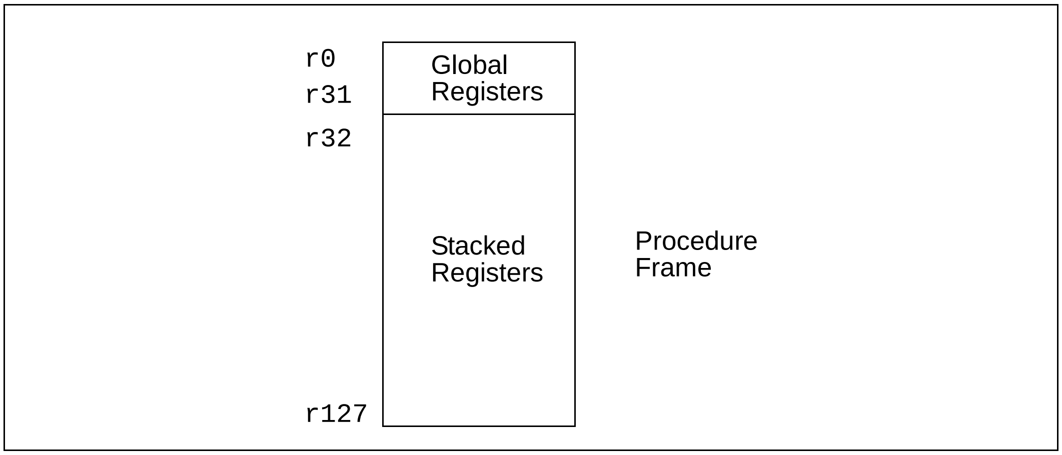
See Chapter 11, “Predication, Control Flow, and Instruction Stream” for detailed discussion of predication. There are a few special cases where predicated instructions read or write architectural resources regardless of their qualifying predicate.

9.6 IA-64 Support for Procedure Calls

Calling conventions normally require callee and caller saved registers which can incur significant overhead during procedure calls and returns. To address this problem, a subset of the IA-64 general registers are organized as a logically infinite set of stack frames that are allocated from a finite pool of physical registers.

9.6.1 Stacked Registers

Registers `r0` through `r31` are called global or static registers and are not part of the stacked registers. The stacked registers are numbered `r32` up to a user-configurable maximum of `r127`:



A called procedure specifies the size of its new stack frame using the `alloc` instruction. The procedure can use this instruction to allocate up to 96 registers per frame shared amongst input, output, and local values. When a call is made, the output registers of the calling procedure are overlapped with the input registers of the called procedure, thus allowing parameters to be passed with no register copying or spilling.

The hardware renames physical registers so that the stacked registers are always referenced in a procedure starting at `r32`.

9.6.2 Register Stack Engine

Management of the register stack is handled by a hardware mechanism called the Register Stack Engine (RSE). The RSE moves the contents of physical registers between the general register file and memory without explicit program intervention. This provides a programming model that looks like an unlimited physical register stack to compilers; however, saving and restoring of registers by the RSE may be costly, so compilers should still attempt to minimize register usage.

9.7 Branches and Hints

Since branches have a major impact on program performance, IA-64 includes features to improve their performance by:

- Using predication to reduce the number of branches in the code. This improves instruction fetching because there are fewer control flow changes, decreases the number of branch mispredicts since there are fewer branches, and it increases the branch prediction hit rates since there is less competition for prediction resources.
- Providing software hints for branches to improve hardware use of prediction and prefetching resources.
- Supplying explicit support for software pipelining of loops and exit prediction of counted loops.

9.7.1 Branch Instructions

Branching in IA-64 is largely expressed the same way as on other microprocessors. The major difference is that branch triggers are controlled by predicates rather than conditions encoded in branch instructions. IA-64 also provides a rich set of hints to control branch prediction strategy, prefetching, and specific branch types like loops, exits, and branches associated with software pipelining. Targets for indirect branches are placed in branch registers prior to branch instructions.

9.7.2 Loops and Software Pipelining

Compilers sometimes try to improve the performance of loops by using unrolling. However, unrolling is not effective on all loops for the following reasons:

- Unrolling may not fully exploit the parallelism available.
- Unrolling is tailored for a statically defined number of loop iterations.
- Unrolling can increase code size.

To maintain the advantages of loop unrolling while overcoming these limitations, IA-64 provides architectural support for software pipelining. Software pipelining enables the compiler to interleave the execution of several loop iterations without having to unroll a loop. Software pipelining in IA-64 is performed using:

- Loop-branch instructions.
- `LC` and `EC` application registers.
- Rotating registers and loop stage predicates.
- Branch hints that can assign a special prediction mechanism to important branches.

In addition to software pipelined *while* and *counted* loops, IA-64 provides particular support for simple counted loops using the `br.cloop` instruction. The `cloop` branch instruction uses the 64-bit Loop Count (`LC`) application register rather than a qualifying predicate to determine the branch exit condition.

For a complete discussion of software pipelining support in IA-64, see Chapter 12, “Software Pipelining and Loop Support”.

9.7.3 Rotating Registers

Rotating registers enable succinct implementation of software pipelining with predication. Rotating registers are rotated by one register position each time one of the special loop branches is executed. Thus, after one rotation, the content of register `X` will be found in register `X+1` and the value of the highest numbered rotating register will be found in `r32`. The size of the rotating region of general registers can be any multiple of 8 and is selected by a field in the `alloc` instruction. The predicate and floating-point registers can also be rotated but the number of rotating registers is not programmable: predicate registers `p16` through `p63` are rotated, and floating-point registers `f32` through `f127` are rotated.

9.8 Summary

IA-64 provides features that reduce the effects of traditional microarchitectural performance barriers by enabling:

- Improved ILP with a large number of registers and software scheduling of instruction groups and bundles.
- Better branch handling through predication.
- Reduced overhead for procedure calls through the register stack mechanism.
- Streamlined loop handling through hardware support of software pipelined loops.
- Support for hiding memory latency using speculation.

Memory Reference 10

10.1 Overview

Memory latency is a major factor in determining the performance of integer applications. In order to help reduce the effects of memory latency, IA-64 explicitly supports software pipelining, large register files, and compiler-controlled speculation. This chapter discusses features and optimizations related to compiler-controlled speculation. See Chapter 12, "Software Pipelining and Loop Support" for a complete description of how to use software pipelining.

The early sections of this chapter review non-speculative load and store in IA-64 and general concepts and terminology related to data dependences. The concept of speculation is then introduced, followed by discussions and examples of how speculation is used in IA-64. The remainder of this chapter describes several important optimizations related to memory access and instruction scheduling.

10.2 Non-Speculative Memory References

IA-64 supports non-speculative loads and stores, as well as explicit memory hint instructions.

10.2.1 Stores to Memory

Integer store instructions in IA-64 can write either 1, 2, 4, or 8 bytes and 4, 8, or 10 bytes for floating-point stores. For example, a `st4` instruction will write the first four bytes of a register to memory.

Although IA-64 uses a little endian memory byte order by default, software can change the byte order by setting the big endian (be) bit of the user mask (UM).

10.2.2 Loads from Memory

Integer load instructions in IA-64 can read either 1, 2, 4, or 8 bytes from memory depending on the type of load issued. Loads of 1, 2, or 4 bytes of data are zero-extended to 64-bits prior to being written into their target registers.

Although loads are provided for various data types, the basic IA-64 data type is the quadword (8 bytes). Apart from a few exceptions, all integer operations are on quadword data. This can be particularly important when dealing with signed integers and 32-bit addresses, or any addresses that are shorter than 64 bits.

10.2.3 Data Prefetch Hint

The `lfetch` instruction requests that lines be moved between different levels of the memory hierarchy. Like all hint instructions in IA-64, `lfetch` has no effect on program correctness, and any microarchitecture implementation of IA-64 may choose to ignore it.

10.3 Instruction Dependences

Data and control dependences are fundamental factors in optimization and instruction scheduling. Such dependences can prevent a compiler from scheduling instructions in an order that would yield shorter critical paths and better resource usage since they restrict the placement of instructions relative to other instructions on which they are dependent.

In general, memory references are the major source of control and data dependences that cannot be broken due to getting a wrong answer (if a data dependence is broken) or raising a fault that should not be raised (if a control dependence is broken). This section describes:

- Background material on memory reference dependences
- Descriptions of how dependences constrain code scheduling on traditional architectures

Section 10.4 describes IA-64 memory reference features that increase the number of dependences that can be removed by a compiler.

10.3.1 Control Dependences

An instruction is *control dependent* on a branch if the direction taken by the branch affects whether the instruction is executed. In the code below, the load instruction is control dependent on the branch:

```
(p1)br.cond some_label
    ld8 r4=[r5]
```

The following sections provide overviews of control dependences and their effects on optimization.

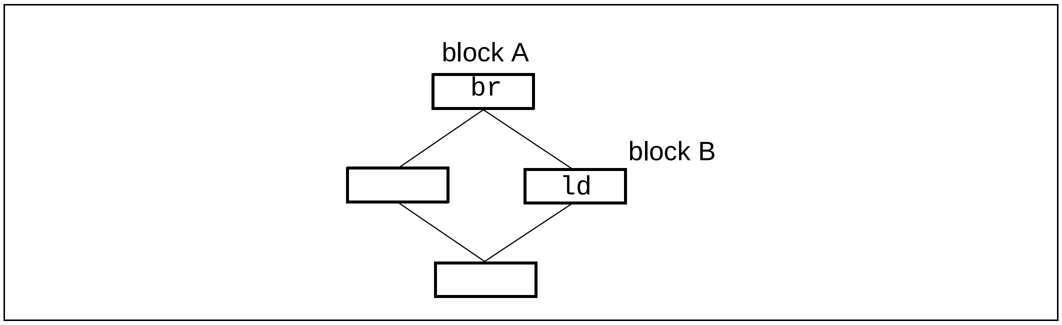
10.3.1.1 Instruction Scheduling and Control Dependencies

The code below contains a control dependence at the branch instruction:

```
    add r7=r6,1             // Cycle 0
    add r13=r25,r27
    cmp.eq p1,p2=r12,r23
(p1)br.cond  some_label ;;

    ld4 r2=[r3] ;;          // Cycle 1
    sub r4=r2,r11           // Cycle 3
```

A compiler cannot safely move the load instruction before the branch unless it can guarantee that the moved load will not cause a fatal program fault or otherwise corrupt program state. Since the load cannot be moved upward, the schedule cannot be improved using normal code motion.

Thus, the branch creates a barrier to instructions whose execution depends upon it. In the figure below, the load in block B cannot be moved up because of a conditional branch at the end of block A.



10.3.2 Data Dependences

A data dependence exists between an instruction that accesses a register or memory location and another instruction that alters the same register or location.

10.3.2.1 Basics of Data Dependence

The following basic terms describe data dependences between instructions:

Write-after-write (WAW) A dependence between two instructions that write to the same register or memory location.

Write-after-read (WAR) A dependence between two instructions in which an instruction reads a register or memory location that a subsequent instruction writes.

Read-after-write (RAW) A dependence between two instructions in which an instruction writes to a register or memory location that is read by a subsequent instruction.

Ambiguous memory dependences
Dependences between a load and a store, or between two stores where it cannot be determined if the involved instructions access overlapping memory locations. Ambiguous memory references include possible WAW, WAR, or RAW dependences.

Independent memory references
References by two or more memory instructions that are known not to have conflicting memory accesses.

10.3.2.2 Data Dependence in IA-64

The IA-64 architecture requires the programmer to insert stops between RAW and WAW *register* dependences to ensure correct code results. For example, in the code below, the add instruction computes a value in r4 needed by the sub instruction:

```
add r4=r5,r6 ;;// Instruction group 1
sub r7=r4,r9   // Instruction group 2
```

The stop after the add instruction terminates one instruction group so that the sub instruction can legally read r4.

On the other hand, IA-64 implementations are architecturally required to observe *memory*-based dependences within an instruction group. In a single instruction group, a program can contain memory-based data dependent instructions and hardware will produce the same results as if the instructions were executed sequentially and in program order. The pseudo-code below demonstrates a memory dependence that will be observed by hardware:

```
mov r16=1
mov r17=2 ;;
st8 [r15]=r16
st8 [r14]=r17 ;;
```

If the address in `r14` is equal to the address in `r15`, uni-processor hardware guarantees that the memory location will contain the value in `r17` (2). The following RAW dependence is also legal in the same instruction group even if software is unable to determine if `r1` and `r2` overlap:

```
st8 [r1]=x
ld4 y=[r2]
```

10.3.2.3 Instruction Scheduling and Data Dependencies

The dependence rules are sufficient to generate correct code, but to generate efficient code, the compiler must take into account the latencies of instructions. For example, the generic implementation has a two cycle latency to the first level data cache. In the code below, the stop maintains correct ordering, but a use of `r2` is scheduled only one cycle after its load:

```
add r7=r6,1            // Cycle 0
add r13=r25,r27
cmp.eq p1,p2=r12,r23 ;;

add r11=r13,r29        // Cycle 1
ld4 r2=[r3] ;;

sub r4=r2,r11          // Cycle 3
```

Since the latency of a load is two cycles, the `sub` instruction will stall until cycle three. To avoid a stall, the compiler can move the load earlier in the schedule so that the machine can perform useful work each cycle:

```
ld4 r2=[r3]            // Cycle 0
add r7=r6,1
add r13=r25,r27
cmp.eq p1,p2=r12,r23 ;;

add r11=r13,r29 ;;     // Cycle 1

sub r4=r2,r11          // Cycle 2
```

In this code, there are enough independent instructions to move the load earlier in the schedule to make better use of the functional units and reduce execution time by one cycle.

Now suppose that the original code sequence contained an ambiguous memory dependence between a store instruction and the load instruction:

```
add r7=r6,1             // Cycle 0
add r13=r25,r27
cmp.ne p1,p2=r12,r23 ;;

st4 [r29]=r13           // Cycle 1
ld4 r2=[r3] ;;

sub r4=r2,r11           // Cycle 3
```

In this case, the load cannot be moved past the store due to the memory dependence. Stores will cause data dependences if they cannot be disambiguated from loads or other stores.

In the absence of other architectural support, stores can prevent moving loads and their dependent instructions: The following C language statements could not be reordered unless `ptr1` and `ptr2` were statically known to point to independent memory locations:

```
*ptr1 = 6;
x = *ptr2;
```

10.4 Using IA-64 Speculation to Overcome Dependences

Both data and control dependences constrain optimization of program code. IA-64 provides support for two basic techniques used to overcome dependences:

Data speculation — Allows a load and possibly its uses to be moved across ambiguous memory writes.

Control speculation — Allows a load and possibly its uses to be moved across a branch on which the load is control dependent.

These techniques are used to hide load latencies and reduce execution time.

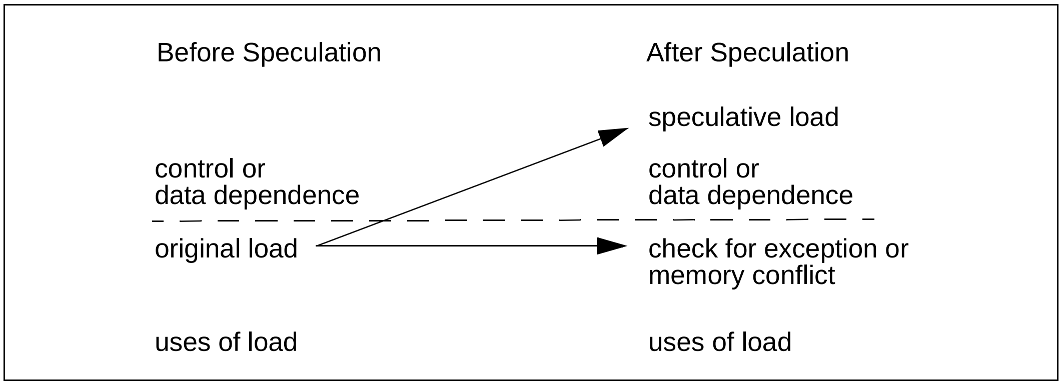
10.4.1 IA-64 Speculation Model

The limitations imposed by dependences on instruction scheduling can be solved by separating the loading of data from the exception handling or the acknowledgment of data conflicts. IA-64 supports special speculative versions of instructions to accomplish this:

- Control speculative load instructions defer exceptions
- Data speculative load instructions save address information
- Special check instructions check for exceptions or data conflicts.

An IA-64 speculative load can be moved above a dependence barrier (shown as a dashed line) as shown in the figure below.

The check detects a deferred exception or a conflict with an intervening store and provides a mechanism to recover from failed speculation. With this support, speculative loads and their uses can be scheduled earlier than non-speculative instructions. As a result, the memory latencies of these loads can be hidden more easily than for non-speculative loads.



10.4.2 Using IA-64 Data Speculation

Data speculation in IA-64 uses a special load instruction (`ld.a`) called an *advanced load* instruction and an associated check instruction (`chk.a` or `ld.c`) to validate data-speculated results.

When the `ld.a` instruction is executed, an entry is allocated in a hardware structure called the Advanced Load Address Table (ALAT). The ALAT is indexed by physical register number and records the load address, the type of the load, and the size of the load.

A check instruction must be executed before the result of an advanced load can be used by any non-speculative instruction. The check instruction must specify the same register number as the corresponding advanced load.

When a check instruction is executed, the ALAT is searched for an entry with the same target physical register number and type. If an entry is found, execution continues normally with the next instruction.

If no matching entry is found, the speculative results need to be recomputed:

- Use a `chk.a` if a load and some of its uses are speculated. The `chk.a` jumps to compiler-generated recovery code to re-execute the load and dependent instructions.
- Use a `ld.c` if no uses of the load are speculated. The `ld.c` reissues the load.

Entries are removed from the ALAT due to:

- Stores that write to addresses overlapping with ALAT entries
- Other advanced loads that target the same physical registers as ALAT entries
- Implementation-defined hardware or operating system conditions needed to maintain correctness
- Limitations of the capacity, associativity, and matching algorithm used for a given implementation of the ALAT

10.4.2.1 Advanced Load Example

Advanced loads can reduce the critical path of a sequence of instructions. In the code below, a load and store may access conflicting memory addresses:

```
st8 [r4]=r12       // Cycle 0: ambiguous store
ld8 r6=[r8] ;;     // Cycle 0: load to advance
add r5=r6,r7 ;;    // Cycle 2
st8 [r18]=r5       // Cycle 3
```

On the generic machine model, the code above would execute in four cycles, but it can be rewritten using an advanced load and check:

```
ld8.a r6=[r8]      // Cycle -2 or earlier

// Other instructions

st8 [r4]=r12       // Cycle 0: ambiguous store
ld8.c r6=[r8]      // Cycle 0: check load
add r5=r6,r7 ;;    // Cycle 0
st8 [r18]=r5       // Cycle 1
```

The original load has been turned into a check load, and an advanced load has been scheduled above the ambiguous store. If the speculation succeeds, the execution time of the remaining non-speculative code is reduced because the latency of the advanced load is hidden.

10.4.2.2 Recovery Code Example

Consider again the non-speculative code from the last section:

```
st8 [r4]=r12       // Cycle 0: ambiguous store
ld8 r6=[r8] ;;     // Cycle 0: load to advance
add r5=r6,r7 ;;    // Cycle 2
st8 [r18]=r5       // Cycle 3
```

The compiler could move up not only the load, but also one or more of its uses. This transformation uses a `chk.a` rather than a `ld.c` instruction to validate the advanced load. Using the same example code sequence but now advancing the `add` as well as the `ld8` results in:

```
    ld8.a   r6=[r8] ;; // Cycle -3

    // other instructions

    add     r5=r6,r7   // Cycle -1: add that uses r6

    // Other instructions

    st8     [r4]=r12   // Cycle 0
    chk.a   r6,recover // Cycle 0: check
back: // Return point from jump to recover
    st8     [r18]=r5   // Cycle 0
```

Recovery code must also be generated:

```
recover:
    ld8     r6=[r8] ;; // Reload r6 from [r8]
    add     r5=r6,r7   // Re-execute the add
    br      back       // Jump back to main code
```

If the speculation fails, the check instruction branches to the label `recover` where the speculated code is re-executed. If the speculation succeeds, execution time of the transformed code is three cycles less than the original code.

10.4.2.3 Terminology Review

Terms related to speculation, such as *advanced loads* and *check loads*, have well-defined meanings in IA-64. The terms below were introduced in the preceding sections:

Data speculative load
A speculative load that is statically scheduled prior to one or more stores upon which it may be dependent. The data speculative load instruction is `ld.a`.

Advanced load
A data speculative load.

Check load
An instruction that checks whether a corresponding advanced load needs to be re-executed and does so if required. The check load instruction is `ld.c`.

Advanced load check
An instruction that takes a register number and an offset to a set of compiler-generated instructions to re-execute speculated instructions when necessary. The advanced load check instruction is `chk.a`.

Recovery code
Program code that is branched to by a speculation check. Recovery code repeats a load and chain of dependent instructions to recover from a speculation failure.

10.4.3 Using Control Speculation in IA-64

The check to determine if control speculation was successful is similar to that for data speculation.

10.4.3.1 The NAT Bit

The Not A Thing (NaT) bit is an extra bit on each of the general registers. A register NaT bit indicates whether the content of a register is valid. If the NaT bit is set to one, the register contains a deferred exception token due to an earlier speculation fault. In a floating-point register, the presence of a special value called the NaTVal signals a deferred exception.

During a control speculative load, the NaT bit on the destination register of the load may be set if an exception occurs and it is deferred. The exact set of events and exceptions that cause an exception to be deferred (thus causing the NaT bit to be set), depends in part upon operating system policy. When a speculative instruction reads a source register that has its NaT bit set, NaT bits of the target registers of that instruction are also set. That is, NaT bits are propagated through dependent computations.

10.4.3.2 Control Speculation Example

When a control speculative load is scheduled, the compiler must insert a speculative check, `chk.s`, along all paths on which results of the speculative load are consumed. If a non-speculative instruction (other than a `chk.s`) reads a register with its NaT bit set, a NaT consumption fault occurs, and the operating system will terminate the program.

The code sequence below illustrates a basic use of IA-64 control speculation:

```
(p1)br.cond some_label// Cycle 0
    ld8 r1=[r5] ;;    // Cycle 1
    add r2=r1,r3      // Cycle 3
```

This code can be rewritten using a control speculative load and check. The check can be placed in the same basic block as the original load:

```
    ld8.s r1=[r5] ;;  // Cycle -2

    // Other instructions

(p1)br.cond some_label// Cycle 0
    chk.s r1,recovery // Cycle 0
    add  r2=r1,r3     // Cycle 0
```

Until a speculation check is reached dynamically, the results of the control speculative chain of instructions cannot be stored to memory or otherwise accessed non-speculatively without the possibility of a fault. If a speculation check is executed and the NaT bit on the checked register is set, the processor will branch to recovery code pointed to by the check instruction.

It is also possible to test for the presence of set NaT bits and NaTVals using the test NaT (`tnat`) and floating-point class (`fclass`) instructions.

Although every speculative computation needs to be checked, this does not mean that every speculative load requires its own `chk.s`. Speculative checks can be optimized by taking advantage of the propagation of NaT bits through registers as described in Section 10.5.6.

10.4.3.3 Spills, Fills and the UNAT Register

Saving and restoring of registers that may have set NaT bits is enabled by `st8.spill` and `ld8.fill` instructions and the User NaT Collection application register (UNAT).

The “spill general register and NaT” instruction, `st8.spill`, saves eight bytes of a general register to memory and writes its NaT bit into the UNAT. Bits 8:3 of the memory address of the store determine which UNAT bit is written with the register NaT value. The “fill general register” instruction, `ld8.fill`, reads eight bytes from memory into a general register and sets the register NaT bit according to the value in the UNAT. Software is responsible for saving and restoring the UNAT contents to ensure correct spilling and filling of NaT bits.

The corresponding floating-point instructions, `stf.spill` and `ldf.fill`, save and restore floating-point registers in floating-point register format without surfacing exceptions due to NaTVals.

10.4.3.4 Terminology Review

The terms below are related to control speculation:

Control speculative load — A speculative load that is scheduled prior to an earlier controlling branch. References to “speculative loads” without qualifiers generally refer to control speculative loads and not data speculative loads. Loads using the `ld.s` instruction are control speculative loads.

Speculation check — An instruction that checks whether a speculative instruction has deferred an exception. Speculation check instructions include labels that point to compiler-generated recovery code. The speculation check instruction is `chk.s`.

Recovery code — Code executed to recover from a speculation failure. Control speculative recovery code is analogous to data speculative recovery code.

10.4.4 Combining Data and Control Speculation

A load that is both data and control speculative is called a *speculative advanced load*. The `ld.sa` instruction performs all the operations of both a speculative load and an advanced load. An ALAT entry will not be allocated if this type of load generates a deferred exception token, so an advanced load check instruction (`chk.a`) is sufficient to check for both interference from subsequent stores and for deferred exceptions.

10.5 Optimization of Memory References

Speculation can increase parallelism and help to hide latency by enabling more code motion than can be performed on traditional architectures. Speculation can increase the application of traditional loop optimizations such as invariant code motion and common subexpression elimination. IA-64 also offers post-increment loads and stores that improve instruction throughput without increasing code size.

Memory reference optimization should take several factors into account including:

- Difference between the execution costs of speculative and non-speculative code
- Code size
- Interference probabilities and properties of the ALAT (for data speculation)

The remainder of this chapter discusses these factors and optimizations relating to memory accesses.

10.5.1 Speculation Considerations

The use of data speculation requires more attention than the use of control speculation. In part this is due to the fact that one control speculative load cannot inadvertently cause another control speculative load to fail. Such an effect is possible with data speculative loads since the ALAT has limited capacity and the replacement policy of ALAT entries is implementation dependent. For example, if an advanced load is issued and there are no unused ALAT entries, the hardware may choose to invalidate an existing entry to make room for a new one.

Moreover, exceptions associated with control speculative calculations are uncommon in correct code since they are related to events such as page faults and TLB misses. However, excessive control speculation can be expensive as associated instructions fill issue slots.

Although the static critical path of a program may be reduced by the use of data speculation, the following factors contribute to the benefit/dynamic cost of data speculation:

- The probability that an intervening store will interfere with an advanced load
- The cost of recovering from a failed advanced load
- The specific microarchitectural implementation of the ALAT: its size, associativity, and matching algorithm.

Determining interference probabilities can be difficult, but dynamic memory profiling can help to predict how often ambiguous loads and stores will conflict.

When using advanced loads, there should be case-by-case consideration as to whether advancing only a load and using a `ld.c` might be preferable to advancing both a load and its uses, which would require the use of the potentially more expensive `chk.a`.

Even when recovery code is not executed, its presence extends the lifetimes of registers used in data and control speculation, thus increasing register pressure and possibly the cost of register movement by the Register Stack Engine (RSE). See Section 10.5.3 for information on considerations for recovery code placement.

10.5.2 Data Interference

Data references with *low* interference probabilities and *high* path probabilities can make the best use of data speculation. In the pseudo-code below, assume the probabilities that the stores to `*p1` and `*p2` conflict with `var` are independent.

```
*p1 =       /* Prob interference = 0.30 */
. . .
*p2 =       /* Prob interference = 0.40 */
. . .
    = var   /* Load to be advanced */
```

If the compiler advances the load from var above the stores to pointers `p1` and `p2`, then:

```
Prob that stores to p1 or p2 interfere with var
    = 1.0 - (Prob p1 will not interfere with var *
            Prob p2 will not interfere with var)
    = 1.0 - (0.70 * 0.60)
    = 0.58
```

Given the interference probabilities above, there is a 58% probability at least one of `p1` and `p2` will interfere with a load from `var` if it is advanced above both of them. A compiler can use traditional heuristics concerning data interference and interprocedural memory access information to estimate these probabilities.

When advancing loads past function calls, the following should be considered:

- If a called function has many stores in it, it is more likely that actual or aliased ALAT conflicts will occur.
- If other advanced loads are executed during the function call, it is possible that their physical register numbers will either be identical or conflict with ALAT entries allocated from calls in parent functions.

- If it is unknown whether a large number of advanced loads will be executed by the called routines, then the possibility that the capacity of that ALAT may be exceeded must be considered.

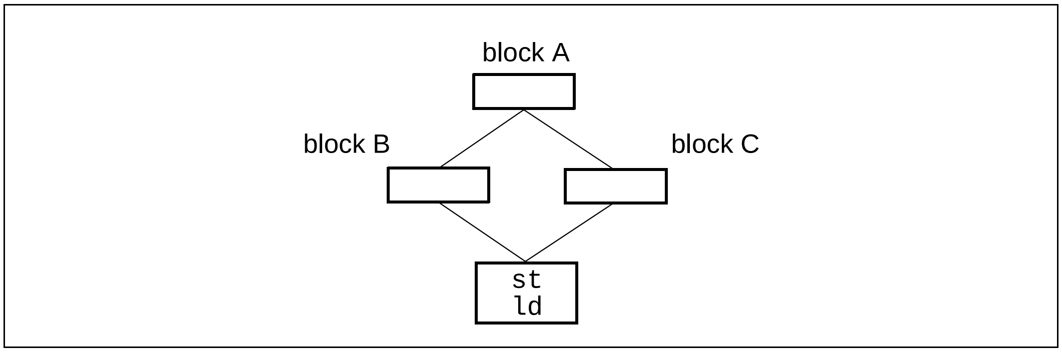
10.5.3 Optimizing Code Size

Part of the decision of when to speculate should involve consideration of any possible increases in code size. *Such consideration is not particular to speculation, but to any transformations that cause code to be duplicated, such as loop unrolling, procedure inlining, or tail duplication.* Techniques to minimize code growth are discussed later in this section.

In general, control speculation increases the dynamic code size of a program since some of the speculated instructions are executed and their results are never used. Recovery code associated with control speculation primarily contributes to the static size of the binary since it is likely to be placed out-of-line and not brought into cache until a speculative computation fails (uncommon for control speculation).

Data speculation has a similar effect on code size except that it is less likely to compute values that are never used since most non-control speculative data speculative loads will have their results checked. Also, since control speculative loads only fail in uncommon situations such as deferred data related faults (depending on operating system configuration), while data speculative loads can fail due to ALAT conflicts, actual memory conflicts, or aliasing in the ALAT, the decision as to where to place recovery code for advanced loads is more difficult than for control speculation and should be based on the expected conflict rate for each load.

As a general rule, efficient compilers will attempt to minimize code growth related to speculation. As an example, moving a load above the join of two paths may require duplication of speculative code on every path. The flow graph depicted below and the explanation shows how this could arise.



If the compiler or programmer advanced the load up to block B from its original non-speculative position, all speculative code would need to be duplicated in both blocks B and C. This duplicated code might be able to occupy NOP slots that already exist. But if space for the code is not already available, it might be preferable to advance the load to block A since only one copy would be required in this case.

10.5.4 Using Post-Increment Loads and Stores

Post-increment loads and stores can improve performance by combining two operations in a single instruction. Although the text in this section mentions only post-increment loads, most of the information applies to stores as well.

Post-increment loads are issued on M-units and can increment their address register by either an immediate value or by the contents of a general register. The following pseudo-code that performs two loads:

```
ld8 r2=[r1]
add r1=1,r1 ;;
ld8 r3=[r1]
```

can be rewritten using a post-increment load:

```
ld8 r2=[r1],1 ;;
ld8 r3=[r1]
```

Post-increment loads may not offer direct savings in dependence path height, but they are important when calculating addresses that feed subsequent loads:

- A post-increment load avoids code size expansion by combining two instructions into one.
- Adds can be issued on either I-units or M-units. When a program combines an add with a load, an I-unit or M-unit resource remains available that otherwise would have been consumed. Thus, throughput of dependent adds and loads can be doubled by using post-increment loads.

A disadvantage of post-increment loads is that they create new dependences between post-increment loads and the operations that use the post-increment values. In some cases, the compiler may wish to separate post-increment loads into their component instructions to improve the overall schedule. Alternatively, the compiler could wait until after instruction scheduling and then opportunistically find places where post-increment loads could be substituted for separate load and add instructions.

10.5.5 Loop Optimization

In cyclic code, speculation can extend the use of classical loop optimizations like invariant code motion. Examine this pseudo-code:

```
while (cond) {
    c = a + b; // Probably loop invariant
    *ptr++ = c;// May point to a or b
}
```

The variables `a` and `b` are probably loop invariant; however, the compiler must assume the stores to `*ptr` will overwrite the values of `a` and `b` unless analysis can guarantee that this can never happen. The use of advanced loads and checks allows code that is likely to be invariant to be removed from a loop, even when a pointer cannot be disambiguated:

```
    ld4.a r1 = [&a]
    ld4.a r2 = [&b]
    add r3 = r1,r2// Move computation out of loop
    while (cond) {
        chk.a.nc r1, recover1
L1:     chk.a.nc r2, recover2
L2:     *p++ = r3
    }
```

At the end of the module:

```
recover1:      // Recover from failed load of a
    ld4.a r1 = [&a]
    add  r3 = r1, r2
    br.sptk L1 // Unconditional branch

recover2:      // Recover from failed load of b
    ld4.a r2 = [&b]
    add  r3 = r1, r2
    br.sptk L2 // Unconditional branch
```

Using speculation in this loop hides the latency of the calculation of `c` whenever the speculated code is successful.

Since checks have both a clear (clr) and no clear (nc) form, the programmer must decide which to use. This example shows that when checks are moved out of loops, the no clear version should be used. This is because the clear (clr) version will cause the corresponding ALAT entry to be removed (which would cause the next check to that register to fail).

10.5.6 Minimizing Check Code

Checks of speculative loads can sometimes be combined to reduce code size. The propagation of NaT bits and NaTVals via speculative instructions can permit a single check of a speculative result to replace multiple intermediate checks. The code below demonstrates this optimization potential:

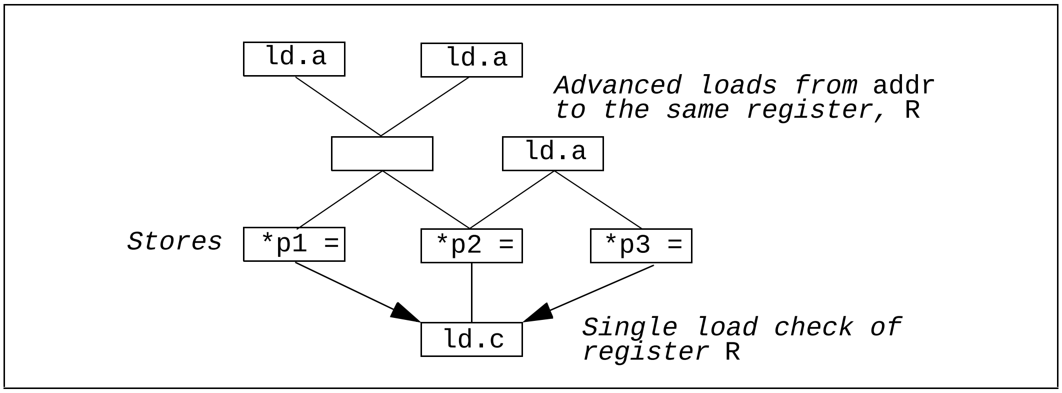
```
    ld4.s r1=[r10] // Speculatively load to r1
    ld4.s r2=[r20] // Speculatively load to r2
    add  r3=r1,r2;;// Add two speculative values

    // Other instructions

    chk.s r3,imm21 // Check for NaT bit in r3
    st4 [r30]=r1   // Store r1
    st4 [r40]=r2   // Store r2
    st4 [r50]=r3   // Store r3
```

Only the result register, `r3`, needs to be checked before stores of any of `r1`, `r2`, or `r3`. If a NaT bit were set at the time of the control speculative loads of `r1` or `r2`, the NaT bit would have been propagated to `r3` from `r1` or `r2` via the `add` instruction.

Another way to reduce the amount of check code is to use control flow analysis to avoid issuing extra `ld.c` or `ld.a` instructions. For example, the compiler can schedule a single check where it is known to be reached by all copies of the advanced load. The portion of a flow graph shown in the figure below demonstrates where this technique might be applied.

A single check in the lowermost block shown for all of the advanced loads is correct if both of these conditions are met:

- The lowermost block post-dominates all of the blocks with advanced loads from location `addr.`
- The lowermost block precedes any uses of the advanced loads from `addr.`



10.6 Summary

The examples in this chapter show where IA-64 can take advantage of existing techniques like dynamic profiling and disambiguation. Special IA-64 support allows implementation of speculation in common scenarios in which it would normally not be allowed. Speculation, in turn, increases ILP by making greater code motion possible, thus enhancing traditional optimizations such as those involving loops.

Even though IA-64's speculation model can be applied in many different situations, careful cost and benefit analysis is needed to insure best performance.

Predication, Control Flow, and Instruction Stream 11

11.1 Overview

This chapter is divided into three sections that describe optimizations related to predication, control flow, and branch hints as follows:

- The **predication** section describes if-conversion, predicate usage, and code scheduling to reduce the affects of branching.
- The **control flow optimization** section describes optimizations that collapse and converge control flow by using parallel compares, multiway branches, and multiple register writers under predicate.
- The **branch and prefetch hints** section describes how hints are used to improve branch and prefetch performance.

11.2 Predication

Predication allows the compiler to convert control dependences into data dependences. This section describes several sources of branch-related performance issues, followed by a summary of IA-64's predication mechanism, followed by a series of descriptions of optimizations and techniques based on predication.

11.2.1 Performance Costs of Branches

Branches can decrease application performance by consuming hardware resources for prediction at execution time and by restricting instruction scheduling freedom during compilation.

11.2.1.1 Prediction Resources

Branch prediction resources include branch target buffers, branch prediction tables, and the logic used to control these resources. The number of branches that can accurately be predicted is limited by the size of the buffers on the processor, and such buffers tend to be small relative to the total number of branches executed in a program.

This limitation means that branch intensive code may have a large portion of its execution time spent due to contention for prediction resources. Furthermore, even though the size of the predictors is a primary factor in determining branch prediction performance, some branches are best predicted with different types of predictors. For example, some branches are best predicted statically while others are more suitably predicted dynamically. Of those predicted dynamically, some are of greater importance than others, such as loop branches.

Since the cost of a misprediction is generally proportional to pipeline length, good branch prediction is essential for processors with long instruction pipelines. Thus, optimizing the use of prediction resources can significantly improve the overall performance of an application.

Suppose, for instance, that the conditional in the code below is mispredicted 30% of the time and branch mispredictions incur a ten cycle penalty. On average, the mispredicted branch will add three cycles to each execution of the code sequence (30% * 10 cycles):

```
if (r1)
    r2 = r3 + r4;
else
    r7 = r6 - r5;
```

Equivalent IA-64 code that has not been optimized is shown below. It requires five instructions including two branches and executes in two cycles, not including potential misprediction or taken-branch penalty cycles:

```
     cmp.eq     p1,p2=r1,r0     // Cycle 0
(p1) br.cond    else_clause     // Cycle 0
     add        r2=r3,r4        // Cycle 1
     br         end_if          // Cycle 1
else_clause:
     sub        r7=r6,r5        // Cycle 1
end_if:
```

Using the information above, this code will take five cycles to execute on average even thought the critical path is only two cycles long (2 cycles + (30% * 10 cycles) = 5). If the branch misprediction penalty could be eliminated (either by reducing contention for resources or by removing the branch itself), performance of the code sequence would improve by a factor of two.

11.2.1.2 Instruction Scheduling

Branches limit the ability of the compiler to move instructions that alter memory state or that can raise exceptions, because instructions in a program are control dependent on all lexically enclosing branches. In addition to the control dependences, compound conditionals can take several cycles to compute and may themselves require intermediate branches in languages like C that require short-circuit evaluation.

Control speculation is the primary mechanism used to perform global code motion for IA-64 compilers. However, when an instruction does not have a speculative form or the instruction could potentially corrupt memory state, control speculation may be insufficient to allow code motion. Thus, techniques that allow greater freedom in code motion or eliminate branches can improve the compiler’s ability to schedule instructions.

11.2.2 Predication in IA-64

Now that the performance implications of branching have been described, this section overviews predication – the primary IA-64 mechanism used by optimizations described in this section.

Almost all IA-64 instructions can be tagged with a guarding predicate.   If the value of the guarding predicate is false at execution time, then the predicated instruction’s architectural updates are suppressed, and the instruction behaves like a nop. If the predicate is true, then the instruction behaves as if it were unpredicated. There are a small number of instructions such as unconditional compares and floating-point square-root and reciprocal approximate instructions whose qualifying predicate do not operate as described above. See *Part I: IA-64 Application Architecture Guide* for additional information.

The following sequence shows a set of predicated instructions:

```
(p1) add   r1=r2,r3
(p2) ld8   r5=[r7]
(p3) chk.s r4,recovery
```

To set the value of a predict register, IA-64 provides compare and test instructions such as those as shown below.

```
    cmp.eq  p1,p2=r5,r6
    tbit    p3,p4=r6,5
```

Additionally, a predicate almost always requires a stop to separate its producing instruction and its use:

```
    cmp.eq  p1,p2=r1,r2 ;;
(p1)add     r1=r2,r3
```

The only exception to this rule involves an integer compare or test instruction that sets a predicate that is used as the condition for a subsequent branch instruction:

```
    cmp.eq   p1,p2=r1,r2   // No stop required
(p1)br.cond some_target
```

11.2.3 Optimizing Program Performance Using Predication

This section describes predication-related optimizations, their use, and basic performance analysis techniques. Following are descriptions of optimizations including if-conversion, misprediction elimination, off-path predication, upward code motion, and downward code motion.

11.2.3.1 Applying If-Conversion

One of the most important optimizations enabled by predication is the complete removal of branches from some program sequences. Without predication, the pseudo-code below would require a branch instruction to conditionally jump around the if-block code:

```
    if (r4) {
        add  r1=r2,r3
        ld8  r6=[r5]
    }
```

Using predication, the sequence can be written without a branch:

```
    cmp.ne  p1,p0=r4,0 ;;// Set predicate reg
(p1)add     r1=r2,r3
(p1)ld8     r6=[r5]
```

The process of predicating instructions in conditional blocks and removing branches is referred to as *if-conversion*. Once if-conversion has been performed, instructions can be scheduled more freely because there are fewer branches to limit code motion, and there are fewer branches competing for issue slots.

In addition to removing branches, this transformation will make dynamic instruction fetching more efficient since there are fewer possibilities for control flow changes. Under more complex circumstances, several branches can be removed. The following C code sequence:

```
if (r1)
    r2 = r3 + r4;
else
    r7 = r6 - r5;
```

can be rewritten in IA-64 assembler without branches as:

```
     cmp.ne p1,p2 = r1,0 ;;
(p1) add    r2 = r3,r4
(p2) sub    r7 = r6,r5
```

Since instructions from opposite sides of the conditional are predicated with complementary predicates they are guaranteed not to conflict, hence the compiler has more freedom when scheduling to make the best use of hardware resources. The compiler could also try to schedule these statements with earlier or later code since several branches and labels have been removed as part of if-conversion.

Since the branches have been removed, no branch misprediction is possible and there will be no pipeline bubbles due to taken branches. Such effects are significant in many large applications, and these transformations can greatly reduce branch-induced stalls or flushes in the pipeline.

Thus, comparing the cost of the code above with the non-predicated version above shows that:

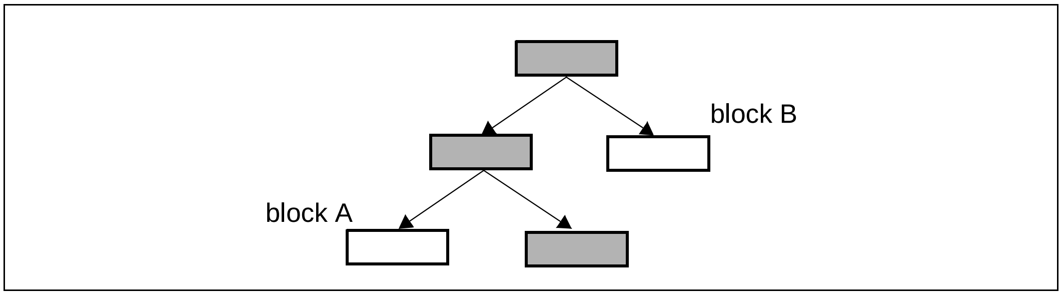
- Non-predicated code consumes: 2 cycles + (30% * 10 cycles) = 5 cycles
- Predicated code consumes: 2 cycles

In this case, predication saves an average of three cycles.

11.2.3.2 Off-Path Predication

If a compiler has dynamic profile information, it is possible to form an instruction schedule based on the control flow path that is most likely to execute – this path is called the main trace. In some cases, execution paths not on the main trace are still executed frequently, and thus it may be beneficial to use predication to minimize their critical paths as well.

The main trace of a flow graph is highlighted in the picture below. Although blocks A and B are not on the main trace, suppose they are executed a significant number of times.



If some of the instructions in block A or block B can be included in the main trace without increasing its critical path, then techniques of upward code motion can be applied to reduce the critical path through blocks A and B when they are taken. An example of how to use predication to implement upward code motion is given in the next section.

11.2.3.3 Upward Code Motion

When traditional control speculation is inadequate, it may still be possible to predicate an instruction and move it up or down in the schedule to reduce dependence height. This is possible because predicating an instruction replaces a control dependence with a data dependence. If the data dependence is less constraining than the control dependence, such a transformation may improve the instruction schedule.

Given the IA-64 assembly sequence below, the store instruction cannot be moved above the enclosing conditional instruction because it could cause an address fault or other exception, depending upon the branch direction:

```
(p1)br.cond some_label    // Cycle 0
    st4     [r34] = r23   // Cycle 1
    ld4     r5 = [r56]    // Cycle 1
    ld4     r6 = [r57]    // Cycle 2:no cycle 1 M's
```

One reason why it might be desirable to move the store instruction up is to allow loads below it to move up.

Note: Ambiguous stores are barriers beyond which normal loads cannot move. In this case, moving the store also frees up an M-unit slot. To rewrite the code so that the store comes before the branch, `p2` has been assigned the complement of `p1`:

```
(p2)st4     [r34] = r23   // Cycle 0
(p2)ld4     r5 = [r56]    // Cycle 0
(p1)br.cond some_label    // Cycle 0
    ld4     r6 = [r57]    // Cycle 1
```

Since the store is now predicated, no faults or exceptions are possible when the branch is taken, and memory state is only updated if and when the original home block of the store is entered. Once the store is moved, it is also possible to move the load instruction without having to use advanced or speculative loads (as long as `r5` is not live on the taken branch path).

11.2.3.4 Downward Code Motion

As with upward code motion, downward code motion is normally difficult in the presence of stores. The next example shows how code can be moved downward past a label, a transformation that is often unsafe without predication:

```
    ld8     r56 = [r45] ;; // Cycle 0: load
    st4     [r23] = r56 ;; // Cycle 2: store
label_A:
    add ...                // Cycle 3
    add ...
    add ...
    add ... ;;
```

In the code above, suppose the latency between the load and the store is two clocks. Assuming the load instruction cannot be moved upward due to other dependences, the only way to schedule the instructions so that the load latency is covered is to move the store downward past the label.

The following code demonstrates the overall idea of using predicates to enable downward code motion. In actual compiler-generated code, the predicates that are explicitly computed in this example might already be available in predicate registers and not require extra instructions.

```
    // Point which “dominates” label_A
    cmp.ne p1,p0 = r0,r0   // Initialize p1 to false

    // Other instructions

    cmp.eq  p1,p0 = r0,r0  // Initialize p1 to true
    ld8     r56=[r45] ;;   // Cycle 0
label_A:
    add     ...            // Cycle 1
    add     ...
    add     ...
    add     ... ;;
(p1)st4     [r23]=r56      // Cycle 2
```

Here, downward code motion saves one cycle. There are examples of more sophisticated situations involving cyclic scheduling, other store-constrained code motion, or pulling code from outside loops into them, but they are not described here.

11.2.3.5 Cache Pollution Reduction

Loads and stores with predicates that are false at runtime are generally likely not to cause any cache lines to be removed, replaced, or brought in. Also, no extra instructions or recovery code are required as would be necessary for IA-64 control or data speculation. Therefore, when the use of predication yields the same critical path length as IA-64 data or control speculation, it is almost always preferable to use predication.

11.2.4 Predication Considerations

Even though predication can have a variety of beneficial effects, there are several cases where the use of predication should be carefully considered. Such cases are usually associated with execution paths that have unbalanced total latencies or over-usage of a particular resource such as those associated with memory operations.

11.2.4.1 Unbalanced Execution Paths

The simple conditional below has an unbalanced flow-dependence height. Suppose that non-predicated assembly for this sequence takes two clocks for the if-block and approximately 18 clocks if we assume a `setf` takes 8 clocks, a `getf` takes 2 clocks, and an `xma` takes 6 clocks:

```
    if (r4)    // 2 clocks
        r3 = r2 + r1;
    else       // 18 clocks
        r3 = r2 * r1;
    f (r3);    // An integer use of r3
```

If-converted IA-64 code is shown below. The cycle numbers shown depend upon the values of `p1` and `p2` and assume the latencies shown:

```
    // Issue cycle if p2 is:TrueFalse
    cmp.ne  p1,p2=r4,r0;;  //  0   0
(p1)add     r3=r2,r1       //  1   1
(p2)setf    f1=r1          //  1   1
(p2)setf    f2=r2 ;;       //  1   1
(p2)xma.l   f3=f1,f2,f0 ;; //  9   2
```

```
(p2)getf    r3=f3 ;;        //  15  3
(p2)use of  r3              //  17  4
```

This code takes 18 cycles to complete if p2 is true and five cycles if p2 is false. When analyzing such cases, consider execution weights, branch misprediction probabilities, and prediction costs along each path.

In the three scenarios presented below, assume a branch misprediction costs ten cycles. No instruction cache or taken-branch penalties are considered.

11.2.4.2 Case 1

Suppose the if-clause is executed 50% of the time and the branch is never mispredicted. The average number of clocks for:

- Unpredicated code is: (2 cycles * 50%) + (18 cycles * 50%) = 10 clocks
- Predicated code is: (5 cycles * 50%) + (18 cycles * 50%) = 11.5 clocks

In this case, if-conversion would *increase* the cost of executing the code.

11.2.4.3 Case 2

Suppose the if-clause is executed 70% of the time and the branch mispredicts 10% if the time with mispredicts costing 10 clocks. The average number of clocks for:

- Unpredicated code is:
  (2 cycles * 70%) + (18 cycles * 30%) + (10 cycles * 10%) = 7.8 clocks
- Predicated code is:
  (5 cycles * 70%) + (18 cycles * 30%) = 8.9 clocks

In this case, if-conversion still would *increase* the cost of executing the code.

11.2.4.4 Case 3

Suppose the if-clause is executed 30% of the time and the branch mispredicts 30% of the time. The average number of clocks for:

- Unpredicated code is:
  (2 cycles * 30%) + (18 cycles * 70%) + (10 cycles * 30%) = 16.2 clocks
- Predicated code is:
  (5 cycles * 30%) + (18 cycles * 70%) = 14.1 clocks

In this case, if-conversion would *decrease* the execution cost by more than two clocks, on average.

11.2.4.5 Overlapping Resource Usage

Before performing if-conversion, the programmer must consider the execution resources consumed by predicated blocks in addition to considering flow-dependence height. The *resource availability height* of a set of instructions is the minimum number of cycles taken considering only the execution resources required to execute them.

The code below is derived from an if-then-else statement. Given the generic machine model that has only two load/store (M) units. If a compiler predicates and combines these two blocks, then the resource availability height through the block will be four clocks since that is the minimum amount of time necessary to issue eight memory operations:

```
then_clause:
    ld  r1=[r21]    // Cycle 0
    ld  r2=[r22]    // Cycle 0
    st  [r32]=r3    // Cycle 1
    st  [r33]=r4 ;;// Cycle 1
    br  end_if
else_clause:
    ld  r3=[r23]    // Cycle 0
    ld  r4=[r24]    // Cycle 0
    st  [r34]=r5    // Cycle 1
    st  [r35]=r6 ;;// Cycle 1
end_if:
```

As with the example in the previous section, assuming various misprediction rates and taken branch penalties changes the decision as to when to predicate and when not to predicate. One case is illustrated here:

11.2.4.6 Case 1

Suppose the branch condition mispredicts 10% of the time and that the predicated code takes four clocks to execute. The average number of clocks for:

- Non-predicated code is: (10 cycles * 10%) + 2 cycles = 3 cycles
- Predicated code is: 4 cycles

Predicating this code would *increase* execution time even though the flow dependence heights of the branch paths are equal.

11.2.5 Guidelines for Removing Branches

The following if-conversion guidelines apply to cases where only local behavior of the code and its execution profile are known:

1. The flow dependence and resource availability heights of both paths must be considered when deciding whether to predicate or not.
2. If if-conversion increases the length of *any control path* through the original code sequence, careful analysis using profile or misprediction data must be performed to ensure that execution time of the converted code is equivalent to or better than unpredicated code.
3. If if-conversion removes a branch that is mispredicted a significant percentage of the time, the transformation frequently pays off even if the blocks are significantly unbalanced since mispredictions are very expensive.
4. If the flow-dependence heights of the paths being if-converted are nearly equal and there are sufficient resources to execute both streams simultaneously, if-conversion is often advantageous.

Although these guidelines are useful for optimizing segments of code, the behavior of some programs is limited by non-local effects such as overall branch behavior, sensitivity to code size, percentage of time spent servicing branch mispredictions, etc. In these situations, the decision to use if-convert or perform other speculative transformation becomes more involved.

11.3 Control Flow Optimizations

A common occurrence in programs is for several control flows to converge at one point or for multiple control flows to start from one point. In the first case, multiple flows of control are often computing the value of the same variable or register and the join point represents the point at which the program needs to select the correct value before proceeding. In the second case, multiple flows may begin at a point where several independent paths are taken based on a set of conditions.

In addition to these multiway joins and branches, the computation of complex compound conditions normally requires a tree-like computation to reduce several conditions into one. IA-64 provides special instructions that allow such conditions to be computed in fewer tree levels.

A third control-flow related optimization uses predication to improve instruction fetching by if-conversion to generate straight-line sequences that can be efficiently fetched. The use and optimization of these cases is described in the remainder of this section.

11.3.1 Reducing Critical Path with Parallel Compares

The computation of the compound branch condition shown below requires several instructions on processors without special instructions:

```
    if ( rA || rB || rC || rD ) {
        /* If-block instructions */
    }
    /* after if-block */
```

The pseudo-code below, shows one possible solution uses a sequence of branches:

```
    cmp.ne  p1,p0 = rA,0
    cmp.ne  p2,p0 = rB,0
(p1)br.cond if_block
(p2)br.cond if_block
    cmp.ne  p3,p0 = rC,0
    cmp.ne  p4,p0 = rD,0
(p3)br.cond if_block
(p4)br.cond if_block
    // after if-block
```

On many IA-64 implementations, this sequence is likely to require at least two cycles to execute if all the conditions are false, plus the possibility of more cycles due to one or more branch mispredictions. Another possible sequence computes an or-tree reduction:

```
    or      r1 = rA,rB
    or      r2 = rC,rD ;;
    or      r3 = r1,r2 ;;
    cmp.ne  p1,p2 = r3,0
(p1)br      if_block
```

This solution requires three cycles to compute the branch condition which can then be used to branch to the if-block.

Note: It is also possible to predicate the if-block using `p1` to avoid branch mispredictions.

To reduce the cost of compound conditionals, IA-64 has special *parallel compare* instructions to optimize expressions that have and and or operations. These compare instructions are special in that multiple and/or compare instructions are allowed to target the same predicate within a single instruction group. This feature allows the possibility that a compound conditional can be resolved in a single cycle.

For this usage model to work properly, IA-64 requires that the programmer ensure that during any given execution of the code, that all instructions that target a given predicate register must either:

- write the same value (0 or 1) or
- do not write the target register at all.

This usage model means that sometimes a parallel compare may not update the value of its target registers and thus, unlike normal compares, the predicates used in parallel compares must be initialized prior to the parallel compare. Please see *Part I: IA-64 Application Architecture Guide* for full information on the operation of parallel compares.

Initialization code must be placed in an instruction group prior to the parallel compare. However, since the initialization code has no dependences on prior values, it can generally be scheduled without contributing to the critical path of the code.

The instructions below shows how to generate code for the example above using parallel compares:

```
    cmp.ne      p1,p0 = r0,r0 ;; // initialize p1 to 0
    cmp.ne.or   p1,p0 = rA,r0
    cmp.ne.or   p1,p0 = rB,r0
    cmp.ne.or   p1,p0 = rC,r0
    cmp.ne.or   p1,p0 = rD,r0
(p1)br.cond     if_block
```

It is also possible to use p1 to predicate the if-block in-line to avoid a possible misprediction. More complex conditional expressions can also be generated with parallel compares:

```
    if ((rA < 0) && (rB == -15) && (rC > 0))
        /* If-block instructions */
```

The assembly pseudo-code below shows a possible sequence for the C code above:

```
    cmp.eq       p1,p0=r0,r0;; // initialize p1 to 1
    cmp.ne.and   p1,p0=rB,-15
    cmp.ge.and   p1,p0=rA,r0
    cmp.le.and   p1,p0=rC,r0
```

When used correctly, and and or compares write both target predicates with the same value or do not write the target predicate at all. Another variation on parallel compare usage is where both the if and else part of a complex conditional are needed:

```
    if ( rA == 0 || rB == 10 )
        r1 = r2 + r3;
    else
        r4 = r5 -  r6;
```

Parallel compares have an andcm variant that computes both the predicate and its complement simultaneously.

```
    cmp.ne         p1,p2 = r0,r0 ;; // initialize p1,p2
    cmp.eq.or.andcm p1,p2 = rA,r0
    cmp.eq.or.andcm p1,p2 = rB,10 ;;
(p1)add            r1=r2,r3
(p2)sub            r4=r5,r6
```

Clearly, these instructions can be used in other combinations to create more complex conditions.

11.3.2 Reducing Critical Path with Multiway Branches

While there are no special instructions to support branches with multiple conditions and multiple targets, IA-64 has implicit support by allowing multiple consecutive B-syllables within an instruction group.

An example uses a basic block with four possible successors. The following IA-64 multi-target branch code uses a BBB bundle template and can branch to either block B, block C, block D, or fall through to block A:

```
label_AA:
        ... // Instructions in block AA
{ .bbb
(p1)br.cond label_B
(p2)br.cond label_C
(p3)br.cond label_D
}
            // Fall through to A
label_A:
        ... // Instructions in block A
```

The ordering of branches is important for program correctness unless all branches are mutually exclusive, in which case the compiler can choose any ordering desired.

11.3.3 Selecting Multiple Values for One Variable or Register with Predication

A common occurrence in programs is for a set of paths that compute different values for the same variable to join and then continue. A variant of this is when separate paths need to compute separate results but could otherwise use the same registers since the paths are known to be complementary. The use of predication can optimize these cases.

11.3.3.1 Selecting One of Several Values

When several control paths that each compute a different value of a single variable meet, a sequence of conditionals is usually required to select which value will be used to update the variable. The use of predication can efficiently implement this code without branches:

```
    switch (rW)
    case 1:
        rA = rB + rC;
        break;
    case 2:
        rA = rE + rF;
        break;
    case 3:
        rA = rH - rI;
        break;
```

The entire switch-block above can be executed in a single cycle using predication if all of the predicates have been computed earlier. Assume that if `rW` equals 1, 2, or 3, then one of `p1`, `p2`, or `p3` is true, respectively:

```
(p1)add  rA=rB,rC
(p2)add  rA=rE,rF
(p3)sub  rA=rH,rI ;;
```

Without this predication capability, numerous branches or conditional move operations would be needed to collapse these values.

IA-64 allows multiple instructions to target the same register in the same clock provided that only one of the instructions writing the target register is predicated true in that clock. Similar capabilities exist for writing predicate registers, as discussed in Section 11.3.1.

11.3.3.2 Reducing Register Usage

In some instances it is possible to use the same register for two separate computations in the presence of predication. This technique is similar to the technique for allowing multiple writers to store a value into the same register, although it is a register allocation optimization rather than a critical path issue.

After if-conversion, it is particularly common for sequences of instructions to be predicated with complementary predicates. The contrived sequence below shows instructions predicated by `p1` and `p2`, which are known by the compiler to be complementary:

```
(p1)add  r1=r2,r3
(p2)sub  r5=r4,r56
(p1)ld8  r7=[r2]
(p2)ld8  r9=[r6] ;;
(p1)a use of r1
(p2)a use of r5
(p1)a use of r7
(p2)a use of r9
```

Assuming registers `r1`, `r5`, `r7`, and `r9` are used for compiler temporaries, each of which is live only until its next use, the preceding code segment can be rewritten as:

```
(p1)add  r1=r2,r3
(p2)sub  r1=r4,r56 // Reuse r1
(p1)ld8  r7=[r2]
(p2)ld8  r7=[r6] ;;// Reuse r7
(p1)a use of r1
(p2)a use of r1
(p1)a use of r7
(p2)a use of r7
```

The new sequence uses two fewer registers. With the 128 registers that IA-64 provides this may not seem essential, but reducing register use can still reduce program and register stack engine spills and fills that can be common in codes with high instruction-level parallelism.

11.3.4 Improving Instruction Stream Fetching

Instructions flow through the pipeline most efficiently when they are executed in large blocks with no taken branches. Whenever the instruction pointer needs to be changed, the hardware may have to insert bubbles into the pipeline either while the target prediction is taking place or because the target address is not computed until later in the pipeline.

By using predication to reduce the number of control flow changes, the fetching efficiency will generally improve. The only case where predication is likely to reduce instruction cache efficiency is when there is a large increase in the number of instructions fetched which are subsequently predicated off. Such a situation uses instruction cache space for
instructions that compute no useful results.

11.3.4.1 Instruction Stream Alignment

For many processors, when a program branches to a new location, instruction fetching is performed on instruction cache lines. If the target of the branch does not start on a cache line boundary, then fetching from that target will likely not retrieve an entire cache line. This problem can be avoided if a programmer aligns instruction groups that cross more than one bundle so that the instruction groups do not span cache line boundaries. However, padding all labels would cause an unacceptable increase in code size. A more practical approach aligns only tops of loops and commonly entered basic blocks when the first instruction group extends across more than one bundle. That is, if both of the following conditions are true at some label L, then padding previous instruction groups so that `L` is aligned on a cache line boundary is recommended:

- The label is commonly branched to from out-of-line. Examples include tops of loops and commonly executed else clauses.
- The instruction group starting at label `L` extends across more than one bundle.

To illustrate, assume code at label `L` in the segment below is not cache-aligned and that a cache boundary occurs between the two bundles. If a program were to branch to `L`, then execution may split issue after the third add instruction even though there are no resource oversubscriptions or stops:

```
L:
{ .mii
    add     r1=r2,r3
    add     r4=r5,r6
    add     r7=r8,r9
}
{ .mfb
    ld8     r14=[r56] ;;
    nop.f
    nop.b
}
```

On the other hand, if `L` were aligned on an even-numbered bundle, then all four instructions at `L` could issue in one cycle.

11.4 Branch and Prefetch Hints

Branch and prefetch hints are architecturally defined to allow the compiler or hand coder to provide extra information to the hardware. Compared to hardware, the compiler has more time, looks at a wider instruction window (including the source), and performs more analysis. Transfer of this knowledge to the processor can help to reduce penalties associated with icache accesses and branch prediction.

Two types of branch-related hints are defined by the IA-64 architecture: branch prediction hints and instruction prefetch hints. Branch prediction hints let the compiler recommend the resources (if any) that should be used to dynamically predict specific branches. With prefetch hints, the compiler can indicate the areas of the code that should be prefetched to reduce demand Icache misses.

Hints can be specified as completers on branch (`br`) and move to branch register (abbreviated mov2br in this text since the actual mnemonic is `mov br=xx`). The hints on branch instructions are the easiest to use since the instruction already exists and the hint completer just has to be specified. mov2br instructions are used for indirect branches. The exact interpretation of these hints is implementation specific although the general behavior of hints is expected to be similar between processor generations.

It is also possible to re-write the hint fields on branches later using a binary rewriting tools. This can occur statically or at execution time based on profile data without changing the correctness of the program. This technique allows IA-64 static hints to be tailored for usage patterns that may not be fully known at compilation time or when the binaries are first distributed.

11.5 Summary

This chapter has presented a wide variety of topics related to optimizing control flow including predication, branch architecture, multiway branches, parallel compares, instruction stream alignment, and branch hints. Although such topics could have been presented in separate chapters, the interplay between the features is best understood by their effects on each other.

Predication and its interplay on scheduling region formation is central to IA-64 performance. Unfortunately, discussion of compiler algorithms of this nature are far beyond the scope of this document.

Software Pipelining and Loop Support 12

12.1 Overview

IA-64 provides extensive support for software-pipelined loops, including register rotation, special loop branches, and application registers. When combined with predication and support for speculation, these features help to reduce code expansion, path length, and branch mispredictions for loops that can be software pipelined.

The beginning of this chapter reviews basic loop terminology and instructions, and describes the problems that arise when optimizing loops in the absence of architectural support. The IA-64 specific loop support features are then introduced. The remainder of this chapter describes the programming and optimization of various type of loops using the IA-64 features.

12.2 Loop Terminology and Basic Loop Support

Loops can be categorized into two types: counted and while. In counted loops, the loop condition is based on the value of a loop counter and the trip count can be computed prior to starting the loop. In while loops, the loop condition is a more general calculation (not a simple count) and the trip count is unknown. Both types are directly supported in IA-64.

IA-64 improves the performance of conventional counted loops by providing a special counted loop branch (the `br.cloop` instruction) and the Loop Count application register (`LC`). The `br.cloop` instruction does not have a branch predicate. Instead, the branching decision is based on the value of the `LC` register. If the `LC` register is greater than zero, it is decremented and the `br.cloop` branch is taken.

12.3 Optimization of Loops

In many loops, there are not enough independent instructions within a single iteration to hide execution latency and make full use of the functional units. For example, in the loop body below, there is very little ILP:

```
L1: ld4    r4 = [r5],4 ;; // Cycle 0  load postinc 4
    add    r7 = r4,r9 ;;  // Cycle 2
    st4    [r6] = r7,4    // Cycle 3  store postinc 4
    br.cloop L1 ;;        // Cycle 3
```

In this code, all the instructions from iteration X are executed before iteration X+1 is started. Assuming that the store from iteration X and the load from iteration X+1 are independent memory references, utilization of the functional units could be improved by moving independent instructions from iteration X+1 to iteration X, effectively overlapping iteration X with iteration X+1.

This section describes two general methods for overlapping loop iterations, both of which result in code expansion on traditional architectures. The code expansion problem is addressed by IA-64 loop support features that are explored later in this chapter. The loop above will be used as a running example in the next few sections.

12.3.1 Loop Unrolling

Loop unrolling is a technique that seeks to increase the available instruction level parallelism by making and scheduling multiple copies of the loop body together. The registers in each copy of the loop body are given different names to avoid unnecessary WAW and WAR data dependences. The code below shows the loop from our example on page 12-1 after unrolling twice (total of two copies of the original loop body) and instruction scheduling, assuming two memory ports and a two cycle latency for loads. For simplicity, assume that the loop trip count is a constant N that is a multiple of two, so that no exit branch is required after the first copy of the loop body:

```
L1: ld4     r4 = [r5],4 ;;     // Cycle 0
    ld4     r14 = [r5],4 ;;    // Cycle 1
    add     r7 = r4,r9 ;;      // Cycle 2
    add     r17 = r14,r9       // Cycle 3
    st4     [r6] = r7,4 ;;     // Cycle 3
    st4     [r6] = r17,4       // Cycle 4
    br.cloop L1 ;;             // Cycle 4
```

The above code does not expose as much ILP as possible. The two loads are serialized because they both use and update `r5`. Similarly the two stores both use and update `r6`. A variable which is incremented (or decremented) once each iteration by the same amount is called an induction variable. The single induction variable `r5` (and similarly `r6`) can be expanded into two registers as shown in the code below:

```
    add     r15 = 4,r5
    add     r16 = 4,r6 ;;
L1: ld4     r4 = [r5],8        // Cycle 0
    ld4     r14 = [r15],8 ;;   // Cycle 0
    add     r7 = r4,r9         // Cycle 2
    add     r17 = r14,r9 ;;    // Cycle 2
    st4     [r6] r7,8          // Cycle 3
    st4     [r16] = r17,8      // Cycle 3
    br.cloop L1 ;;             // Cycle 3
```

Compared to the original loop on page 12-1, twice as many functional units are utilized and the code size is twice as large. However, no instructions are issued in cycle 1 and the functional units are still under utilized in the remaining cycles. The utilization can be increased by unrolling the loop more times, but at the cost of further code expansion. The loop below is unrolled four times (assuming the trip count is multiple of four):

```
    add     r15 = 4,r5
    add     r25 = 8,r5
    add     r35 = 12,r5
    add     r16 = 4,r6
    add     r26 = 8,r6
    add     r36 = 12,r6 ;;
L1: ld4     r4 = [r5],16       // Cycle 0
    ld4     r14 = [r15],16 ;;  // Cycle 0
    ld4     r24 = [r25],16     // Cycle 1
    ld4     r34 = [r35],16 ;;  // Cycle 1
    add     r7 = r4,r9          // Cycle 2
    add     r17 = r14,r9 ;;     // Cycle 2
    st4     [r6] = r7,16        // Cycle 3
    st4     [r16] = r17,16      // Cycle 3
    add     r27 = r24,r9        // Cycle 3
    add     r37 = r34,r9 ;;     // Cycle 3
    st4     [r26] = r27,16      // Cycle 4
    st4     [r36] = r37,16      // Cycle 4
    br.cloop L1 ;;              // Cycle 4
```

The two memory ports are now utilized in every cycle except cycle 2. Four iterations are now executed in five cycles verses the two iterations in four cycles for the previous version of the loop.

12.3.2 Software Pipelining

Software pipelining is a technique that seeks to overlap loop iterations in a manner that is analogous to hardware pipelining of a functional unit. Each iteration is partitioned into stages with zero or more instructions in each stage. A conceptual view of a single pipelined iteration of the loop from page 12-1 in which each stage is one cycle long is shown below:

```
stage 1:ld4  r4 = [r5],4
stage 2:---                 // empty stage
stage 3:add  r7 = r4,r9
stage 4:st4  [r6] = r7,4
```

The following is a conceptual view of five pipelined iterations:

```
     1   2   3   4   5            Cycle
-------------------------------------------------------
    ld4                           X
        ld4                      X+1
    add     ld4                  X+2
    st4 add     ld4              X+3
        st4 add     ld4          X+4
            st4 add              X+5
                st4 add          X+6
                    st4          X+7
```

The number of cycles between the start of successive iterations is called the initiation interval (II). In the above example, the II is one. Each stage of a pipelined iteration is II cycles long. Most of the examples in this chapter utilize modulo scheduling, which is a particular form of software pipelining in which the II is a constant and every iteration of the loop has the same schedule. It is likely that software pipelining algorithms other than modulo scheduling could benefit from the IA-64 loop support features. Therefore the examples in this chapter are discussed in terms of software pipelining rather than modulo scheduling.

Software pipelined loops have three phases: prolog, kernel, and epilog, as shown below:

```
     1   2   3   4   5             Phase
-------------------------------------------------------
    ld4
        ld4                        Prolog
    add     ld4
-------------------------------------------------------
    st4 add     ld4                Kernel
        st4 add     ld4
```

```
------------------------------------------------------------------------
               st4 add
                    st4 add                Epilog
                         st4
```

During the prolog phase, a new loop iteration is started every II cycles (every cycle for the above example) to fill the pipeline. During the first cycle of the prolog, stage 1 of the first iteration executes. During the second cycle, stage 1 of the second iteration and stage 2 of the first iteration execute, etc. By the start of the kernel phase, the pipeline is full. Stage 1 of the fourth iteration, stage 2 of the third iteration, stage 3 of the second iteration, and stage 4 of the first iteration execute. During the kernel phase, a new loop iteration is started, and another is completed every II cycles. During the epilog phase, no new iterations are started, but the iterations already in progress are completed, draining the pipeline. In the above example, iterations 3-5 are completed during the epilog phase.

The software pipeline is coded as a loop that is very different from the original source code loop. To avoid confusion when discussing loops and loop iterations, we use the term *source loop* and *source iteration* to refer back to the original source code loop, and the term *kernel loop* and *kernel iteration* to refer to the loop that implements the software pipeline.

In the above example, the load from the second source iteration is issued before result of the first load is consumed. Thus, in many cases, loads from successive iterations of the loop must target different registers to avoid overwriting existing live values. In traditional architectures, this requires unrolling of the kernel loop and software renaming of the registers, resulting in code expansion. Furthermore, in traditional architectures, separate blocks of code are generated for the prolog, kernel, and epilog phases, resulting in additional code expansion.

12.4 IA-64 Loop Support Features

The code expansion that results from loop optimizations (such as software pipelining and loop unrolling) on traditional architectures can increase the number of instruction cache misses, thus reducing overall performance. The IA-64 loop support features allow some loops to be software pipelined without code expansion. Register rotation provides a renaming mechanism that reduces the need for loop unrolling and software renaming of registers. Special software pipelined loop branches support register rotation and, combined with predication, reduce the need to generate separate blocks of code for the prolog and epilog phases.

12.4.1 Register Rotation

Register rotation renames registers by adding the register number to the value of a rotating register base (rrb) register contained in the CFM. The rrb register is decremented when certain special software pipelined loop branches are executed at the end of each kernel iteration. Decrementing the rrb register makes the value in register X appear to move to register X+1. If X is the highest numbered rotating register, it's value wraps to the lowest numbered rotating register.

A fixed-sized area of the predicate and floating-point register files (`p16`-`p63` and `f32`-`f127`), and a programmable-sized area of the general register file are defined to rotate. The size of the rotating area in the general register file is determined by an immediate in the `alloc` instruction and must be either zero or a multiple of 8, up to a maximum of 96 registers. The lowest numbered rotating register in the general register file is `r32`. An rrb register is provided for each of the three rotating register files: `CFM.rrb.gr` for the general registers; `CFM.rrb.fr` for the floating-point registers; `CFM.rrb.pr` for the predicate registers. The software pipelined loop branches decrement all the rrb registers simultaneously.

Below is an example of register rotation. The swp_branch pseudo-instruction represents a software pipelined loop branch:

```
L1: ld4     r35 = [r4],4       // post increment by 4
    st4     [r5] = r37,4       // post increment by 4
    swp_branch  L1 ;;
```

The value that the load writes to r35 is read by the store two kernel iterations (and two rotations) later as r37. In the meantime, two more instances of the load are executed. Because of register rotation, those instances write their result to different registers and do not modify the value needed by the store.

The rotation of predicate registers serves two purposes. The first is to avoid overwriting a predicate value that is still needed. The second purpose is to control the filling and draining of the pipeline. To do this, a programmer assigns a predicate to each stage of the software pipeline to control the execution of the instructions in that stage. This predicate is called the *stage predicate*. For counted loops, p16 is architecturally defined to be the predicate for the first stage, p17 is defined to be the predicate for the second stage, etc. A conceptual view of a pipelined source iteration of the example counted loop on page 12-1 is shown below. Each stage is one cycle long and the stage predicates are shown:

```
stage 1:(p16)  ld4 r4 = [r5],4
stage 2:(p17)  ---            // empty stage
stage 3:(p18)  add r7 = r4,r9
stage 4:(p19)  st4 [r6] = r7,4
```

A register rotation takes place at the end of each stage (when the software-pipelined loop branch is executed in the kernel loop). Thus a 1 written to p16 enables the first stage and then is rotated to p17 at the end of the first stage to enable the second stage for the same source iteration. Each 1 written to p16 sequentially enables all the stages for a new source iteration. This behavior is used to enable or disable the execution of the stages of the pipelined loop during the prolog, kernel, and epilog phases as described in the next section.

12.4.2 Note on Initializing Rotating Predicates

In this chapter, the instruction mov pr.rot = immed is used to initialize rotating predicates. This instruction ignores the value of CFM.rrb.pr. Thus, the examples in this chapter are written assuming that CFM.rrb.pr is always zero prior to the initialization of predicate registers using mov pr.rot = immed.

12.4.3 Software-pipelined Loop Branches

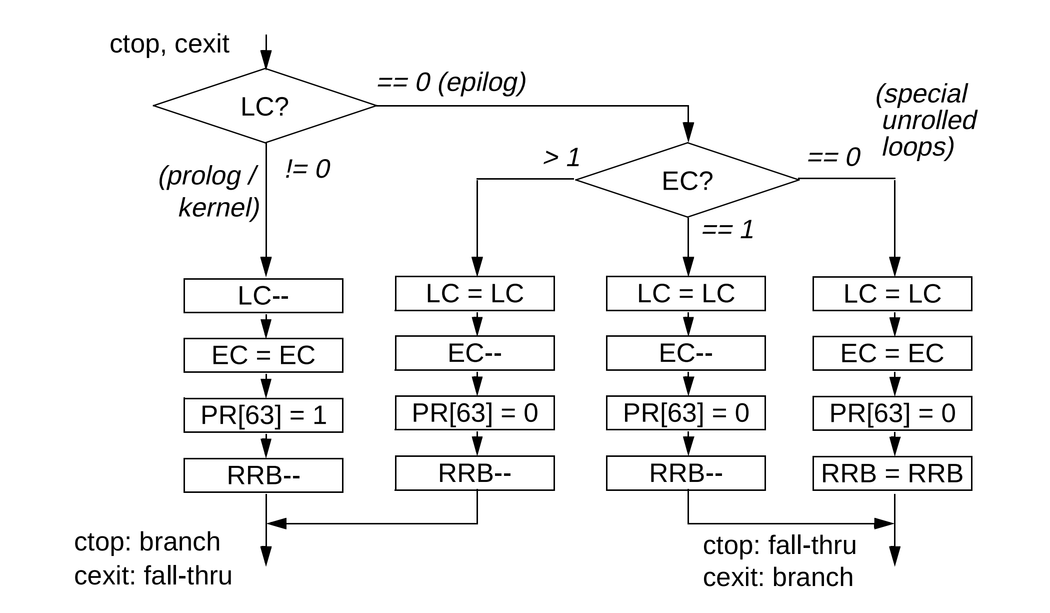
The special software-pipelined loop branches allow the compiler to generate very compact code for software-pipelined loops by supporting register rotation and by controlling the filling and draining of the software pipeline during the prolog and epilog phases. Generally speaking, each time a software-pipelined loop branch is executed, the following actions take place:

1. A decision is made on whether or not to continue kernel loop execution.
2. p16 is set to a value to control execution of the stages of the software pipeline (p63 is written by the branch, and after rotation this value will be in p16).
3. The registers are rotated (rrb registers are decremented).
4. The Loop Count (LC) and/or the Epilog Count (EC) application registers are selectively decremented.

There are two types of software-pipelined loop branches: counted and while.

12.4.3.1 Counted Loop Branches

The figure below shows a flowchart for the counted loop type:



During the prolog and kernel phase, a decision to continue kernel loop execution means that a new source iteration is started. Register rotation must occur so that the new source iteration does not overwrite registers that are in use by prior source iterations that are still in the pipeline. p16 is set to 1 to enable the stages of the new source iteration. LC is decremented to update the count of remaining source iterations. EC is not modified.

During the epilog phase, the decision to continue loop execution means that the software pipeline has not yet been fully drained and execution of the source iterations in progress must continue. Register rotation must continue because the remaining source iterations are still writing results and the consumers of the results expect rotation to occur. p16 is now set to 0 because there are no more new source iterations and the instructions that correspond to non-existent source iterations must be disabled. EC contains the count of the remaining execution stages for the last source iteration and is decremented during the epilog. For most loops, when a software pipelined loop branch is executed with EC equal to 1, it indicates that the pipeline has been drained and a decision is made to exit the loop. The special case in which a software-pipelined loop branch is executed with EC equal to 0 can occur in unrolled software-pipelined loops if the target of the cexit branch is set to the next sequential bundle.

There are two types of software-pipelined loop branches for counted loops. br.ctop is taken when a decision to continue kernel loop execution is made, and is not taken otherwise. It is used when the loop execution decision is located at the bottom of the loop. br.cexit is not taken when a decision to continue kernel loop execution is made, and is taken otherwise. It is used when the loop execution decision is located somewhere other than the bottom of the loop.

12.4.3.2 Counted Loop Example

A conceptual view of a pipelined iteration of the example counted loop on page 12-1 with II equal to one is shown below:

```
stage 1:(p16)  ld4 r4 = [r5],4
stage 2:(p17)  ---            // empty stage
stage 3:(p18)  add r7 = r4,r9
stage 4:(p19)  st4 [r6] = r7,4
```

To generate an efficient pipeline, the compiler must take into account the latencies of instructions and the available functional units. For this example, the load latency is two and the load and add are scheduled two cycles apart. The pipeline below is coded assuming there are two memory ports and the loop count is 200.

Note: Rotating GRs have now been included in the code (the code directly preceding did not). Also, induction variables that are post incremented must be allocated to the static portion of the register file:

```
     mov lc = 199             // LC =loop count - 1
     mov ec = 4               // EC =epilog stages + 1
     mov pr.rot = 1<<16 ;;    // PR16 = 1, rest = 0
L1:
(p16)ld4 r32  = [r5],4        // Cycle 0
(p18)add r35  = r34,r9        // Cycle 0
(p19)st4 [r6] = r36,4         // Cycle 0
     br.ctop L1 ;;            // Cycle 0
```

The memory ports are fully utilized. The table below shows a trace of the execution of this loop:

Cycle	Port/Instructions				State before br.ctop					
	M	I	M	B	p16	p17	p18	p19	LC	EC
0	ld4			br.ctop	1	0	0	0	199	4
1	ld4			br.ctop	1	1	0	0	198	4
2	ld4	add		br.ctop	1	1	1	0	197	4
3	ld4	add	st4	br.ctop	1	1	1	1	196	4
…	…	…	…	…	…	…	…	…	…	…
100	ld4	add	st4	br.ctop	1	1	1	1	99	4
…	…	…	…	…	…	…	…	…	…	…
199	ld4	add	st4	br.ctop	1	1	1	1	0	4
200		add	st4	br.ctop	0	1	1	1	0	3
201		add	st4	br.ctop	0	0	1	1	0	2
202			st4	br.ctop	0	0	0	1	0	1
...					0	0	0	0	0	0

In cycle 3, the kernel phase is entered and the fourth iteration of the kernel loop executes the `ld4`, `add`, and `st4` from the fourth, second, and first source iterations respectively. By cycle 200, all 200 loads have been executed, and the epilog phase is entered. When the `br.ctop` is executed in cycle 202, `EC` is equal to 1. `EC` is decremented, the registers are rotated one last time, and execution falls out of the kernel loop.

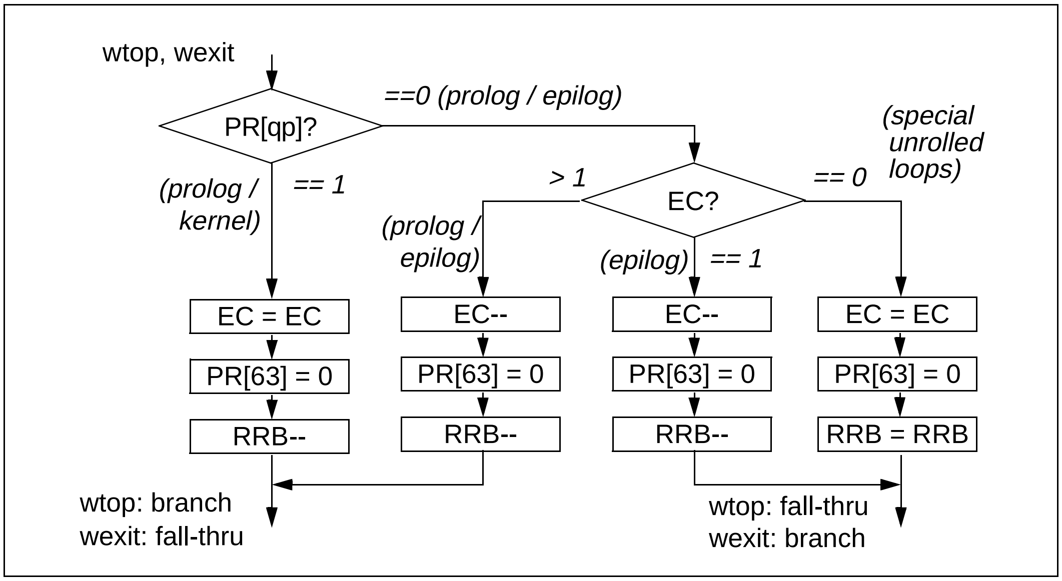
Note: After this final rotation, `EC` and the stage predicates (`p16` – `p19`) are 0.

It is desirable to allocate variables that are loop variant to the rotating portion of the register file whenever possible to preserve space in the static portion for loop invariant variables.

Note: Induction variables that are post incremented must be allocated to the static portion of the register file.

12.4.3.3 While Loop Branches

The figure below shows the flowchart for the while loop type branch:



There are a few differences in the operation of the while loop branch compared to the counted loop branch. The while loop branch does not access `LC` — a branch predicate determines the behavior of this branch instead. During the kernel and epilog phases, the branch predicate is one and zero respectively. During the prolog phase, the branch predicate may be either zero or one depending on the scheme used to program the while loop.   Also, `p16` is always set to zero after rotation. The reasons for these differences are related to the nature of while loops and will be explained in more depth with an example in a later section.

12.4.4 Terminology Review

The terms below were introduced in the preceding sections:

Initiation Interval (II) The number of cycles between the start of successive source iterations in a software pipelined loop. Each stage of the pipeline is II cycles long.

Prolog The first phase of a software-pipelined loop, in which the pipeline is filled.

Kernel The second phase of a software-pipelined loop, in which the pipeline is full.

Epilog The third phase of a software-pipelined loop, in which the pipeline is drained.

Source iteration An iteration of the original source code loop.

Kernel iteration An iteration of the loop that implements the software pipeline.

Register Rotation A form of register renaming that is visible to software. Registers are renamed with respect to a rotating register base that is decremented.

Induction Variable Value that is incremented (or decremented) once per source iteration by the same amount.

12.5 Optimization of Loops in IA-64

Register rotation, predication, and the software pipelined loop branches allow the generation of compact, yet highly parallel code. Speculation can further increase loop performance by removing dependence barriers that limit the throughput of software pipelined loops. Register rotation removes the requirement that kernel loops be unrolled to allow software renaming of the registers. However in some cases performance can be increased by unrolling the source loop prior to software pipelining, or by generating explicit prolog and/or epilog blocks. The remainder of this chapter discusses loop optimizations.

12.5.1 While Loops

The programming scheme for while loops depends upon the structure of the loop. This section discusses do-while loops, in which the loop condition is computed at the bottom of the loop. Optimizing compilers often transform while loops (where the condition is computed at the top of the loop) into do-while loops by moving the condition computation to the bottom of the loop and placing a copy of the condition computation prior to the loop to reduce the number of branches in the loop. The remainder of this section refers to such loops simply as while loops. Below is a simple while loop:

```
L1: ld4     r4 = [r5],4 ;;     // Cycle 0
    st4     [r6] = r4,4        // Cycle 2
    cmp.ne  p1,p0 = r4,r0      // Cycle 2
(p1)br      L1 ;;              // Cycle 2
```

A conceptual view of a pipelined iteration of this loop with II equal to one is shown below:

```
stage 1:    ld4  r4   = [r5],4
stage 2:    ---                // empty stage
stage 3:    st4  [r6]= r4,4
            cmp.ne.unc   p1,p0 = r4,r0
    (p1)    br   L1
```

The following is a conceptual view of four overlapped source iterations assuming the load and store are independent memory references. The store, compare, and branch instructions in stage two are represented by the pseudo-instruction `scb`:

```
     1   2   3   4     Cycle
-------------------------------------------------------
    ld4                  X
        ld4.s           X+1
    scb     ld4.s       X+2
        scb     ld4.s   X+3
            scb         X+4
                scb     X+5
```

Notice that the load for the second source iteration is executed before the compare and branch of the first source iteration. That is, the load (and the update of `r5`) is speculative. The loop condition is not computed until cycle X+2, but in order to maximize the use of resources, it is desirable to start the second source iteration at cycle X+1. Without the support for control speculation in IA-64, the second source iteration could not be started until cycle X+3.

The computation of the loop condition for while loops is very different from that of counted loops. In counted loops, it is possible to compute the loop condition in one cycle using a counted loop branch. This is what a `br.ctop` instruction does when it sets `p16`. In while loops, a compare must compute the loop condition and set the stage predicates. The stages prior to the one containing the compare are called the *speculative stages* of the pipeline, because it is not possible for the compare to completely control the execution of these stages. Therefore, the stage predicate set by the compare is used (after rotation) to control the first non-speculative stage of the pipeline.

The pipelined version of the while loop on page 12-9 is shown below. A check for the speculative load is included:

```
      mov   ec = 2
      mov   pr.rot = 1 << 16 ;;     // PR16 = 1, rest = 0
L1:
      ld4.s r32 = [r5],4            // Cycle 0
(p18) chk.s r34, recovery           // Cycle 0
(p18) cmp.ne p17,p0 = r34,r0        // Cycle 0
(p18) st4    [r6] = r34,4           // Cycle 0
(p17) br.wtop.sptk L1 ;;            // Cycle 0
L2:
```

To explain why the kernel loop is programmed the way it is, it is helpful to examine a trace of the execution of the loop (assume there are 200 source iterations) shown in the table below.

There is no stage predicate assigned to the load because it is speculative. The compare sets `p17`. This is the branch predicate for the current iteration and, after rotation, the stage predicate for the first non-speculative stage (stage three) of the next source iteration. During the prolog, the compare cannot produce its first valid result until cycle two. The initialization of the predicates provides a pipeline that disables the compare until the first source iteration reaches stage two in cycle two. At that point the compare starts generating stage predicates to control the non-speculative stages of the pipeline. Notice that the compare is conditional. If it were unconditional, it would always write a zero to `p17` and the pipeline would not get started correctly.

Cycle	Port/Instructions					State before br.wtop			
	M	I	I	M	B	p16	p17	p18	EC
0	ld4.s				br.wtop	1	0	0	2
1	ld4.s				br.wtop	0	1	0	1
2	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
3	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
…	…	…		…	…	…	…	…	…
100	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
…	…	…		…	…	…	…	…	…
199	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
200	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
201	ld4.s	cmp	chk	st4	br.wtop	0	0	1	1
						0	0	0	0

The executions of `br.wtop` in the first two cycles of the prolog do not correspond to any of the source iterations. Their purpose is simply to continue the kernel loop until the first valid loop condition can be produced. In cycle one, the branch predicate `p17` is one. For this programming scheme, the branch predicate of the `br.wtop` is always a one during the last speculative stage of the first source iteration. During all the prior stages, the branch predicate is zero. If the branch predicate is zero, the `br.wtop` continues the kernel loop only if `EC` is greater than one. It also decrements `EC`. Thus `EC` must be initialized to (# epilog stages + # speculative pipeline stages). In the above example, this is 0 + 2 = 2.

In cycle 201, the compare for the 200th source iteration is executed. Since this is the final source iteration, the result of the compare is a zero and `p17` is unmodified. The zero that was rotated into `p17` from `p16` causes the `br.wtop` to fall through to the loop exit. `EC` is decremented and the registers are rotated one last time.

In the above example, there are no epilog stages. As soon as the branch predicate becomes zero, the kernel loop is exited.

12.5.2 Loops with Predicated Instructions

Instructions that already have predicates in the source loop are not assigned stage predicates. They continue to be controlled by compare instructions in the loop body. For example, the following loop contains predicated instructions:

```
L1: ldfs     f4 = [r5],4
    ldfs     f9 = [r8],4 ;;
    fcmp.ge.unc p1,p2 = f4,f9 ;;
(p1)stfs     [r9] = f4, 4
(p2)stfs     [r9] = f9, 4
    br.cloop L1 ;;
```

Below is a possible pipeline with an II of 2, assuming a floating-point load latency of 9 cycles:

```
stage 1:(p16)  ldfs f4 = [r5],4
        (p16)  ldfs f9 = [r8],4 ;;
               ---               // empty cycle
stage 2-4:     ---               // empty stages
stage 5:       ---               // empty cycle
        (p20)  fcmp.ge.unc p1,p2 = f4,f9 ;;
stage 6:           ---           // empty cycle
        (p1)   stfs [r9] = f4, 4
        (p2)   stfs [r9] = f9, 4
```

The following is the code to implement the pipeline:

```
        mov    lc = 199           // LC = loop count - 1
        mov    ec = 6             // EC = epilog stages + 1
        mov    pr.rot=1<<16;;     // PR16 = 1, rest = 0
L1:
(p16)   ldfs   f32 = [r5],4
(p16)   ldfs   f38 = [r8],4 ;;
(p32)   stfs   [r9] = f37, 4
(p20)   fcmp.ge.unc p31,p32 = f36,f42
(p33)   stfs   [r9] = f43, 4
L2:     br.ctop.sptk L1 ;;
```

12.5.3 Multiple-Exit Loops

All of the example loops discussed so far have a single exit at the bottom of the loop. The loop below contains multiple exits — an exit at the bottom associated with the loop closing branch, and an early exit in the middle:

```
L1:     ld4    r4 = [r5],4 ;;
        ld4    r9 = [r4] ;;
        cmp.eq.unc p1,p0 = r9,r7
(p1)    br.cond  exit             // early exit
        add    r8 = -1,r8 ;;
        cmp.ge.unc p3,p0 = r8,r0
(p3)    br.cond L1 ;;
```

Loops with multiple exits require special care to ensure that the pipeline is correctly drained when the early exit is taken.There are two ways to generate a pipelined version of the above loop: 1) convert it to a single exit loop or 2) pipeline it with the multiple exits explicitly present.

12.5.3.1 Converting Multiple Exit Loops to Single Exit Loops

The first is to transform the multiple exit loop into a single exit loop. In the source loop, execution of the add, the second compare and the second branch is guarded by the first branch. The loop can be transformed into a single exit loop by using predicates to guard the execution of these instructions and moving the early exit branch out of the loop as shown below:

```
L1:     ld4    r4 = [r5],4 ;;
        ld4    r9 = [r4] ;;
        cmp.eq.unc p1,p2 = r9,r7
        add    r8 = -1,r8 ;;
(p2)    cmp.ge.unc p3,p0 = r8,r0
```

```
(p3)    br.cond L1 ;;
(p1)    br.cond exit       // early exit if p1 is 1
```

The computation of p3 determines if either exit of the source loop would have been taken. If p3 is zero, the loop is exited and p1 is used to determine which exit was actually taken. The add is executed speculatively (it is not guarded by p2) to keep the dependence from the cmp.eq to the add from limiting the II. It is assumed that either r8 is not live out at the early exit or that compensation code is added at the target of the early exit. The pipeline for this loop is shown below with the stage predicate assignments but no other rotating register allocation. The compare and the branch at the end of stage 4 are not assigned stage predicates because they already have qualifying predicates in the source loop:

```
stage 1:   ld4.s r4 = [r5],4 ;;   // II = 2
           ---                    // empty cycle
stage 2:   ---                    // empty cycle
           ld4.s r9 = [r4] ;;
stage 3:   ---                    // empty stage
stage 4:
    (p19)  add  r8 = -1,r8
    (p19)  cmp.eq.unc p1,p2 = r9,r7 ;;
    (p2)   cmp.ge.unc p3,p0 = r8,r0
    (p3)   br.cond L1 ;;
```

The code to implement this pipeline is shown below complete with the chk instruction:

```
        mov     ec = 3
        mov     pr.rot = 1 << 16 ;;    // PR16 = 1, rest = 0
L1:     ld4.s   r32 = [r5],4           // Cycle 0
(p19)   chk.s   r36, recovery          // Cycle 0
(p19)   add     r8  = -1,r8            // Cycle 0
(p19)   cmp.eq.unc p31,p32 = r36,r7 ;;// Cycle 0
        ld4.s   r34 = [r33]            // Cycle 1
(p32)   cmp.ge  p18,p0 = r8,r0         // Cycle 1
L2:
(p18)   br.wtop.sptk L1 ;;             // Cycle 1
(p32)   br.cond exit                   // early exit if p32 is 1
```

Note: When the loop is exited, one final rotation occurs, rotating the value in p31 to p32. Thus, p32 is used as the branch predicate for the early exit branch.

12.5.3.2 Pipelining with Explicit Multiple Exits

The second approach is to combine the last three instructions in the loop into a br.cloop instruction and then pipeline the loop. The pipeline using this approach is shown below:

```
stage 1:ld4.s r4 = [r5],4 ;;      // II = 1
stage 4:ld4.s r9 = [r4] ;;
stage 6:cmp.eq.unc p1,p0 = r9,r7
    (p1)br.cond  exit
        br.cloop L1 ;;
```

There are five speculative stages in this pipeline because a non-speculative decision to initiate another loop iteration cannot be made until the br.cond and br.cloop are executed in stage 6. The code to implement this pipeline is shown below assuming a trip count of 200:

```
        mov     lc = 204
        mov     ec = 1
        mov     pr.rot = 1 << 16 ;;    // PR16 = 1, rest = 0
L1:
        ld4.s   r32 = [r5],4           // Cycle 0
(p21)   chk.s   r38, recovery          // Cycle 0
(p21)   cmp.eq.unc p1,p0 = r38,r7      // Cycle 0
        ld4.s   r36 = [r35]            // Cycle 0
(p1)    br.cond exit                   // Cycle 0
L2:     br.ctop.sptk L1;               // Cycle 0
```

When the kernel loop is exited at either the br.cond or the br.ctop, the last source iteration is complete. Thus, EC is initialized to 1 and there is no explicit epilog block generated for the early exit.   The LC register is initialized to five more than 199 because there are five speculative stages. The purpose of the first five executions of br.ctop is simply to keep the loop going until the first valid branch predicate is generated for the br.cond. During each of these executions, LC is decremented, so five must be added to the LC initialization amount to compensate.

A smaller II is achieved with the second approach. This pipelined code will also work if LC is initialized to 199 and EC is initialized to 6. However, if the early exit is taken, LC will have been decremented too many times and will need to be adjusted if it is used at the target of the early exit. If there is any epilog when the early exit is taken, that epilog must be explicit.

12.5.4 Software Pipelining Considerations

There may be instances where it may not be desirable to pipeline a loop. Software pipelining increases the throughput of iterations, but may increase the time required to complete a single iteration. As a result, loops with very small trip counts may experience decreased performance when pipelined. For example, consider the following loop:

```
L1: ld4     r4 = [r5],4            // Cycle 0
    ld4     r7 = [r8],4 ;;         // Cycle 0
    st4     [r6] = r4,4            // Cycle 2
    st4     [r9] = r7,4            // Cycle 2
    br.cloop L1 ;;                 // Cycle 2
```

The following is a possible pipeline with an II of 2:

```
stage 1:ld4  r4 = [r5],4           // Cycle 0
        ld4  r7 = [r8],4 ;;        // Cycle 0
        ---                        // empty cycle
stage 2:---                        // empty cycle
        st4 [r6] = r4,4            // Cycle 3
        st4 [r9] = r7,4 ;;         // Cycle 3
```

In the source loop, one iteration is completed every three cycles. In the software pipelined loop, it takes four cycles to complete the first iteration. Thereafter, iterations are completed every two cycles. If the trip count is two, the execution time of both versions of the loop is the same, six cycles. If the average trip count of the loop is less than two, the software pipelined version of the loop is slower than the source loop.

In addition, it may not be desirable to pipeline a floating-point loop that contains a function call. The number of floating-point registers used by the loop is not known until after the loop is pipelined. After pipelining, it may be difficult to find empty slots for the instructions needed to save and restore the caller-saved floating-point registers across the function call.

12.5.5 Software Pipelining and Advanced Loads

Advanced loads allow some code that is likely to be invariant to be removed from loops, thus reducing the resource requirements of the loop. Use of advanced loads also can reduce the critical path through the iterations, allowing a smaller II to be achieved. See Chapter 10, "Memory Reference" for more information on advanced loads. However, caution must be exercised when using advanced loads with register rotation. For this discussion, we assume an ALAT with 32 entries.

12.5.5.1 Capacity Limitations

An advanced load with a destination that is a rotating register targets a different physical register and allocates a new ALAT entry for each kernel iteration. For example, the simple loop below replaces 32 ALAT entries in 32 iterations:

```
L1: (p16)   ld4.a r32 = [r8]
    (p47)   ld4.c r63 = [r8]
            br.ctop L1 ;;
```

To avoid unnecessary ALAT misses, the check load or advanced load check must be executed before a later advanced load causes a replacement of the entry being checked. In the simple loop above, the unnecessary ALAT misses do not occur because the check load is done within 31 iterations of the advanced load. In the example below, an ALAT miss is encountered for every check load because the advanced load replaces an entry just before the corresponding check load is executed:

```
L1: (p16)   ld4.a r32 = [r8]
    (p48)   ld4.c r64 = [r8]
            br.ctop L1 ;;
```

12.5.5.2 Conflicts in the ALAT

Using an advanced load to remove a likely invariant load from a loop while advancing another load inside the loop results in poor performance if the latter load targets a rotating register. The advanced load that targets the rotating register will eventually invalidate the ALAT entry for the loop invariant load. Thereafter, every execution of the check load for the loop invariant load will cause an ALAT miss.

When more than one advanced load in the loop targets a rotating register, the registers must be assigned and the register lifetimes controlled so that the check load for a particular advanced load X is executed before any of the other advanced loads can invalidate the entry allocated by load X. For example, the following loop successfully targets rotating registers with two advanced loads without any ALAT misses because the two advanced load – check load pairs never create more than 32 simultaneously live ALAT entries:

```
L1: (p16)   ld4.a r32 = [r8]
    (p31)   ld4.c r47 = [r8]
    (p16)   ld4.a r48 = [r9]
    (p31)   ld4.c r63 = [r9]
            br.ctop L1 ;;
```

When the code cannot be arranged to avoid ALAT misses, it may be best to assign static registers to the destinations of the advanced loads and unroll the loop to explicitly rename the destinations of the advanced loads where necessary. The following example shows how to unroll the loop to avoid the use of rotating registers. The loop has an II equal to 1 and the check load is executed one cycle (and one rotation) after the advanced load:

```
L1: (p16)  ld4.a r33 = [r8]
    (p17)  ld4.c r34 = [r8]
           br.ctop L1 ;;
```

Static registers can be assigned to the destinations of the loads if the loop is unrolled twice:

```
L1: (p16)  ld4.a r3 = [r8]
    (p17)  ld4.c r4 = [r8]
           br.cexit L2 ;;
    (p16)  ld4.a r4 = [r8]
    (p17)  ld4.c r3 = [r8]
           br.ctop L1 ;;
L2:        //
```

Rotating registers could still be used for the values that are not generated by advanced loads. The effect of this unrolling on instruction cache performance must be considered as part of the cost of advancing a load.

12.5.6 Loop Unrolling Prior to Software Pipelining

In some cases, higher performance can be achieved by unrolling the loop prior to software pipelining. Loops that are resource constrained can be improved by unrolling such that the limiting resource is more fully utilized. In the following example if we assume the target processor has only two memory units, the loop performance is bound by the number of memory units:

```
L1: ld4  r4 = [r5],4              // Cycle 0
    ld4  r9 = [r8],4 ;;           // Cycle 0
    add  r7 = r4,r9 ;;            // Cycle 2
    st4  [r6] = r7,4              // Cycle 3
    br.cloop L1 ;;                // Cycle 3
```

A pipelined version of this loop must have an II of at least two because there are three memory instructions, but only two memory units. If the loop is unrolled twice prior to software pipelining and assuming the store is independent of the loads, an II of 3 can be achieved for the new loop. This is an effective II of 1.5 for the original source loop. Below is a possible pipeline for the unrolled loop:

```
stage 1:(p16)  ld4     r4 = [r5],8        // odd iteration
        (p16)  ld4     r9 = [r8],8 ;;     // odd iteration
stage 2:(p16)  ld4     r14 = [r15],8      // even iteration
        (p16)  ld4     r19 = [r18],8 ;;   // even iteration
            // ---     empty cycle
stage 3:(p18)  add     r7 = r4,r9         // odd iteration
        (p17)  add     r17 = r14,r19;;    // even iteration
stage 4:    // ---      empty cycle
        (p19)  st4     [r6]  = r7,8       // odd iteration
        (p18)  st4     [r16] = r17,8 ;;   // even iteration
```

The unrolled loop contains two copies of the source loop body, one that corresponds to the odd source iterations and one that corresponds to the even source iterations. The assignment of stage predicates must take this into account. Recall that each 1 written to p16 sequentially enables all the stages for a new source iteration. During stage one of the above pipeline, the stage predicate for the odd iteration is in p16. The stage predicate for the even iteration does not exist yet. During stage two of the above pipeline, the stage predicate for the odd iteration is in p17 and the new stage predicate for the even iteration is in p16. Thus within the same pipeline stage, if the stage predicate for the odd iteration is in predicate register X, the stage predicate for the even iteration is in predicate register X-1. The pseudo-code to implement this pipeline assuming an unknown trip count is shown below:

```
       add     r15 = r5,4
       add     r18 = r8,4
       mov     lc = r2            // LC = loop count - 1
       mov     ec = 4             // EC = epilog stages + 1
       mov     pr.rot=1<<16;;     // PR16 = 1, rest = 0
L1:
(p16)  ld4     r33 = [r5],8       // Cycle 0 odd iteration
(p18)  add     r39 = r35,r38      // Cycle 0 odd iteration
(p17)  add     r38 = r34,r37      // Cycle 0 even iteration
(p16)  ld4     r36 = [r8],8       // Cycle 0 odd iteration
       br.cexit.spnt L3 ;;        // Cycle 0
(p16)  ld4     r33 = [r15],8      // Cycle 1 even iteration
(p16)  ld4     r36 = [r18],8 ;;   // Cycle 1 even iteration
(p19)  st4     [r6]  = r40,8      // Cycle 2 odd iteration
(p18)  st4     [r16] = r39,8      // Cycle 2 even iteration
L2:    br.ctop.sptk L1 ;;         // Cycle 2
L3:
```

Notice that the stages are not equal in length. Stages 1 and 3 are one cycle each, and stages 2 and 4 are two cycles each. Also, the length of the epilog phase varies with the trip count. If the trip count is odd, the number of epilog stages is three, starting after the br.cexit and ending at the br.ctop. If the trip count is even, the number of epilog stages is two, starting after the br.ctop and ending at the br.ctop. The EC must be set to account for the maximum number of epilog stages. Thus for this example, EC is initialized to four. When the trip count is even, one extra epilog stage is executed and br.exit L3 is taken. All of the stage predicates used during the extra epilog stages are equal to 0, so nothing is executed.

The extra epilog stage for even trip counts can be eliminated by setting the target of the br.cexit branch to the next sequential bundle and initializing EC to three as shown below:

```
       add     r15 = r5,4
       add     r18 = r8,4
       mov     lc = r2            // LC = loop count - 1
       mov     ec = 3             // EC = epilog stages + 1
       mov     pr.rot=1<<16;;     // PR16 = 1, rest = 0
L1:
(p16)  ld4     r33 = [r5],8       // Cycle 0 odd iteration
(p18)  add     r39 = r35,r38      // Cycle 0 odd iteration
(p17)  add     r38 = r34,r37      // Cycle 0 even iteration
(p16)  ld4     r36 = [r8],8       // Cycle 0 odd iteration
       br.cexit.spnt L4 ;;        // Cycle 0
L4:
(p16)  ld4     r33 = [r15],8      // Cycle 1 even iteration
(p16)  ld4     r36 = [r18],8 ;;   // Cycle 1 even iteration
(p19)  st4     [r6]  = r40,8      // Cycle 2 odd iteration
```

```
(p18)   st4     [r16] = r39,8       // Cycle 2 even iteration
L2:     br.ctop.sptk L1 ;;          // Cycle 2
L3:
```

If the loop trip count is even, two epilog stages are executed and the kernel loop is exited at the `br.ctop`. If the trip count is odd, the first two epilog stages are executed and then the `br.cexit` branch is taken. Because the target of the `br.cexit` branch is the next sequential bundle (L4), a third epilog stage is executed before the kernel loop is exited at the `br.ctop`. This optimization saves one stage at the end of the loop when the trip count is even, and is beneficial for short trip count loops.

Although unrolling can be beneficial, there are a few issues to consider before trying to unroll and software pipeline. Unrolling reduces the trip count of the loop that is given to the pipeliner, and thus may make pipelining of the loop undesirable since low trip count loops sometimes run faster unpipelined. Unrolling also increases the code size, which may adversely affect instruction cache performance. Unrolling is most beneficial for small loops because the potential performance degradation due to under utilized resources is greater and the effect of unrolling on the instruction cache performance is smaller compared to large loops.

12.5.7 Implementing Reductions

In the following example, a sum of products is accumulated in register f7:

```
    mov     f7 = 0 ;;           // initialize sum
L1: ldfs    f4 = [r5],4
    ldfs    f9 = [r8],4 ;;
    fma     f7 = f4,f9,f7 ;;    // accumulate
    br.cloop L1 ;;
```

The performance is bound by the latency of the `fma` instruction which we assume is 5 cycles for these examples. A pipelined version of this loop must have an II of at least five because the `fma` latency is five. By making use of register rotation, the loop can be transformed into the one below.

Note: The loop has not yet been pipelined. The register rotation and special loop branches are being used to enable an optimization prior to software pipelining.

```
    mov     lc = 199            // LC = loop count - 1
    mov     ec = 1              // Not pipelined, so no epilog
    mov     f33 = 0             // initialize 5 sums
    mov     f34 = 0
    mov     f35 = 0
    mov     f36 = 0
    mov     f37 = 0 ;;
L1: ldfs    f4 = [r5],4
    ldfs    f9 = [r8],4 ;;
    fma     f32 = f4,f9,f37 ;; // accumulate
    br.ctop L1 ;;

    fadd    f10 = f33,f34       // add sums
    fadd    f11 = f35,f36 ;;
    fadd    f12 = f10,f11 ;;
    fadd    f7 = f12,f37
```

This loop maintains five independent sums in registers f33-f37. The fma instruction in iteration X produces a result that is used by the fma instruction in iteration X+5. Iterations X through X+4 are independent, allowing an II of one to be achieved. The code for a pipelined version of the loop assuming two memory ports and a nine cycle latency for a floating-point load is shown below:

```
    mov     lc = 199           // LC = loop count - 1
    mov     ec = 10            // EC = epilog stages + 1
    mov     pr.rot=1<<16       // PR16 = 1, rest = 0
    mov     f33 = 0            // initialize sums
    mov     f34 = 0
    mov     f35 = 0
    mov     f36 = 0
    mov     f37 = 0
L1:
(p16)ldfs   f50 = [r5],4       // Cycle 0
(p16)ldfs   f60 = [r8],4       // Cycle 0
(p25)fma    f41 = f59,f69,f46  // Cycle 0
    br.ctop.sptk L1 ;;         // Cycle 0
    fadd    f10 = f42,f43      // add sums
    fadd    f11 = f44,f45 ;;
    fadd    f12 = f10,f11 ;;
    fadd    f7 = f12,f46
```

12.5.8 Explicit Prolog and Epilog

In some cases, an explicit prolog is necessary for code correctness. This can occur in cases where a speculative instruction generates a value that is live across source iterations. Consider the following loop:

```
    ld4     r3 = [r5] ;;
L1: ld4     r6 = [r8],4        // Cycle 0
    ld4     r5 = [r9],4 ;;     // Cycle 0
    add     r7 = r3,r6 ;;      // Cycle 2
    ld4     r3 = [r5]          // Cycle 3
    and     r10 = 3,r7;;       // Cycle 3
    cmp.ne p1,p0=r10,r11       // Cycle 4
(p1)br.cond L1 ;;              // Cycle 4
```

The following is a possible pipeline for the loop:

```
stage 1:       ld4.s   r6 = [r8],4    // II = 2
               ld4.s   r5 = [r9],4 ;;
               ---                    // empty cycle
stage 2:       ---                    // empty cycle
               ld4.s   r36 = [r5]
               add     r7 = r37,r6 ;;
stage 3:(p18)  and     r10 = 3,r7 ;;
       (p18)   cmp.ne  p1,p0 = r10,r11
       (p1)    br.wtop L1 ;;
```

Note: In the code above, the ld4 and the add instructions in stage 2 have been reordered. Register rotation has been used to eliminate the WAR register dependence from the add to the ld4. The first two stages are speculative. The code to implement the pipeline is shown below:

```
    ld4     r36 = [r5]
    mov     ec = 2
    mov     pr.rot = 1 << 16 ;;// PR16 = 1, rest = 0
L1: ld4.s   r32 = [r8],4       // Cycle 0
    ld4.s   r34 = [r9],4       // Cycle 0
(p18)and    r40 = 3,r39 ;;     // Cycle 0
    ld4.s   r36 = [r35]        // Cycle 1
    add     r38 = r37,r33      // Cycle 1
(p18)chk.s r40, recovery       // Cycle 1
(p18)cmp.ne  p17,p0 = r40,r11  // Cycle 1
(p17)br.wtop L1 ;;             // Cycle 1
```

The problem with this pipelined loop is that the value written to `r36` prior to the loop is overwritten before it is used by the `add`. The value is overwritten by the load into `r36` in the first kernel iteration. This load is in the second stage of the pipeline, but cannot be controlled during the first kernel iteration because it is speculative and does not have a stage predicate. This problem can be solved by peeling off one iteration of the kernel and excluding from that copy any instructions that are not in the first stage of the pipeline as shown below.

Note: The destination register numbers for the instructions in the explicit prolog have been increased by one. This is to account for the fact that there is no rotation at the end of the peeled kernel iteration.

```
    ld4     r37 = [r5]
    mov     ec = 1
    mov     pr.rot = 1<<17 ;;  // PR17 = 1, rest = 0
    ld4     r33 = [r8],4
    ld4     r35 = [r9],4
L1: ld4.s   r32 = [r8],4       // Cycle 0
    ld4.s   r34 = [r9],4       // Cycle 0
(p18)and    r40 = 3,r39;;      // Cycle 0
    ld4.s   r36 = [r35]        // Cycle 1
    add     r38 = r37,r33      // Cycle 1
(p18)chk.s r40, recovery       // Cycle 1
(p18)cmp.ne  p17,p0 = r40,r11  // Cycle 1
(p17)br.wtop L1 ;;             // Cycle 1
```

In some cases, higher performance can be achieved by generating separate blocks of code for all or part of the prolog and/or epilog phase. It is clear from the execution trace of the pipelined counted loop from page 12-7 that the functional units are under-utilized during the prolog and epilog phases. Part of the prolog and epilog could be peeled off and merged with the code preceding and following the loop. The following is a pipelined version of that counted loop with an explicit prolog and epilog:

```
    mov     lc = 196
    mov     ec = 1
prolog:
    ld4     r35 = [r5],4 ;;    // Cycle 0
    ld4     r34 = [r5],4 ;;    // Cycle 1
    ld4     r33 = [r5],4       // Cycle 2
    add     r36 = r35,r9 ;;    // Cycle 2
L1:
    ld4     r32 = [r5],4
    add     r35 = r34,r9
    st4     [r6] = r36,4
L2: br.ctop L1 ;;
epilog:
    add     r35  = r34,r9      // Cycle 0
    st4     [r6] = r36,4 ;;    // Cycle 0
    add     r34 = r33,r9       // Cycle 1
    st4     [r6] = r35,4 ;;    // Cycle 1
    st4     [r6] = r34,4       // Cycle 2
```

The entire prolog (first three iterations of the kernel loop) and epilog (last three iterations) have been peeled off. No attempt has been made to reschedule the peeled instructions. The stage predicates have been removed from the instructions since they are not required for controlling the prolog and epilog phases. Removing them from the prolog makes the prolog instructions independent of the rotating predicates and eliminates the need for software-pipelined loop branches between prolog stages. Thus the entire prolog is independent of the initialization of `LC` and `EC` that precede it. The register numbers in the prolog and epilog have been adjusted to account for the lack of rotation between stages during those phases.

Note: This code assumes that the trip count of the source loop is at least four. If the minimum trip count is unknown at compile time, then a runtime check of the trip count must be added before the prolog. If the trip count is less than four, then control branches to a copy of the original loop.

If this pipelined loop is nested inside an outer loop, there exists a further optimization opportunity. The outer loop could be rotated such that the kernel loop is at the top followed by the epilog for the current outer loop iteration and the prolog for the next outer loop iteration. A copy of the prolog would also be added prior to the outer loop.

Note: From the earlier trace of the counted loop execution, the functional unit usage of the prolog and epilog are complimentary such that they could be very nicely overlapped.

The drawback of creating an explicit prolog or epilog is code expansion.

12.5.9 Redundant Load Elimination in Loops

Unrolling of a loop is sometimes necessary to remove copy operations created by loop optimizations. The following is an example of redundant load elimination. In the code below, each iteration loads two values, one of which has already been loaded by the previous source iteration:

```
    add     r8 = r5,4 ;;
L1: ld4     r4 = [r5],4        // a[i]
    ld4     r9 = [r8],4 ;;     // a[i+1]
    add     r7 = r4,r9 ;;
    st4     [r6] = r7,4
    br.cloop L1 ;;
```

The redundant load can be eliminated by adding a copy of the first load prior to the loop and changing the load to a copy (`mov`):

```
    add     r8 = r5,4
    ld4     r9 = [r5],4;;      // a[i]
L1: mov     r4 = r9            // a[i] = previous a[i+1]
    ld4     r9 = [r8],4 ;;     // a[i+1]
    add     r7 = r4,r9 ;;
    st4     [r6] = r7,4
    br.cloop L1 ;;
```

In traditional architectures, the mov instruction can only be removed by unrolling the loop twice. One instruction is removed from the loop at the cost of two times code expansion. The IA-64 register rotation feature can be used to eliminate the mov instruction without unrolling the loop:

```
    add     r8 = r5,4
    ld4     r33 = [r5],4;;     // a[i]
L1: ld4     r32 = [r8],4 ;;    // a[i+1]
    add     r7 = r33,r32 ;;
    st4     [r6] = r7,4
    br.ctop L1 ;;
```

12.6 Summary

The examples in this chapter show how IA-64 features can be used to optimize loops without the code expansion required with traditional architectures. Register rotation, predication, and the software-pipelined loop branches all contribute to this capability.   Control speculation increases the overlap of the iterations of while loops. Data speculation increases the overlap of iterations of loops that have loads and stores that cannot be disambiguated.

Floating-point Applications 13

13.1 Overview

The IA-64 floating-point architecture is fully ANSI/IEEE-754 standard compliant. IA-64 provides performance enhancing features such as the fused multiply accumulate instruction, the large floating-point register file (with static and rotating sections), the extended range register file data representation, the multiple independent floating-point status fields, and the high bandwidth memory access instructions that enable the creation of compact, high performance, floating-point application code.

The beginning of this chapter reviews some specific performance limitations that are common in floating-point intensive application codes. Later, IA-64 features that address these limitations are presented with illustrative code examples. The remainder of this chapter highlights the optimization of some commonly used kernels using the IA-64 features.

13.2 FP Application Performance Limiters

Floating-point applications are characterized by a predominance of loops. Some loops compute complex calculations on regularly structured data, others simply copy data from one place to another, while others perform gather/scatter-type operations that simultaneously compute and rearrange data. The following sections describe code characteristics that limit performance and how they affect these different kinds of loops.

13.2.1 Execution Latency

Loops often contain recurrence relationships. Consider the tri-diagonal elimination kernel from the Livermore Fortran Kernel suite.

```
 DO 5 i = 2, N
5   X[i] = Z[i] * (Y[i] - X[i-1])
```

The dependence between `X[i]` and `X[i-1]` limits the iteration time of the loop to be the sum of the latency of the subtract and the multiply. The available parallelism can be increased by unrolling the loop and can be exploited by replicating computation, however the fundamental limitation of the data dependence remains.

Sometimes, even if the loop is vectorizable and can be software pipelined, the iteration time of the loop is limited by the execution latency of the hardware that executes the code. A simple vector divide (shown below) is a typical example:

```
 DO 1 I = 1, N
1   X[i] = Y[i] / Z[i]
```

Since typical modern microprocessors contain a non-pipelined floating-point unit, the iteration time of the loop is the latency of the divide which can be tens of clocks.

13.2.2 Execution Bandwidth

When sufficient ILP exists and can be exploited, the performance limitation is the availability of the execution resources – or the execution bandwidth of the machine. Consider the dense matrix multiply kernel from the BLAS3 library.

```
    DO 1 i = 1, N
        DO 1 j = 1, P
            DO 1 k = 1, M
   1            C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

Common techniques of loop interchange, loop unrolling, and unroll-and-jam, can be used to increase the available ILP in the inner loop. When this is done, the inner loop contains an abundance of independent floating-point computations with a relatively small number of memory operations. The performance constraint is then largely the floating-point execution bandwidth of the machine (assuming sufficient registers are available to hold the accumulators – `C[i,j]` and the intermediate computations).

13.2.3 Memory Latency

While cycle time disparity between the processor and memory creates a general memory latency problem for most codes, there are a few special conditions in floating-point codes that exacerbate its impact.

One such condition is the use of indirect addressing. Gather/scatter codes in general and sparse matrix vector multiply code (below) in particular are good examples.

```
    DO 1 ROW = 1, N
        R[ROW] = 0.0d0
        DO 1 I = ROWEND(ROW-1)+1, ROWEND(ROW)
   1        R[ROW] = R[ROW] + A[I] * X[COL[I]]
```

The memory latency of the access of `COL[I]` is exposed, since it is used to index into the vector `X`. The access of the element of `X`, the computation of the product, and the summation of the product on `R[ROW]` are all dependent on the memory latency of the access of `COL[I]`.

Another common condition in floating-point codes where memory latency impact is exacerbated is the presence of ambiguous memory dependences. Consider the incomplete Cholesky conjugate gradient excerpt kernel, again from the Livermore Fortran Kernel suite.

```
    II      = n
    IPNTP   = 0
222 IPNT    = IPNTP
    IPNTP   = IPNTP + II
    II      = II/2
    I       = IPNTP + 1
cdir$ ivdep
    DO 2 K = IPNT+2, IPNTP, 2
       I    = I+1
   2   X[I]= X[K] - V[K] * X[K-1] - V[K-1] * X[K+1]
    IF (II .GT. 1) GO TO 222
```

The DO-loop involves an update of X at the index I using X at the indices K, K+1, K-1. Since it is difficult for the compiler to establish whether these indices overlap, the loads of X[K], X[K+1] or X[K-1] for the next iteration cannot be scheduled until the store of X[I] of the current iteration. This exposes the memory latency of access of these operands.

13.2.4 Memory Bandwidth

Floating-point loops are often limited by the rate at which the machine can deliver the operands of the computation. The DAXPY kernel from the BLAS1 library is a typical example:

```
     DO 1 I = 1, N
    1   Y[I] = Y[I] + A * X[I]
```

The computation requires loading two operands (X[I] and Y[I]) and storing one result (Y[I]) for each floating-point multiply and add operation. If the data arrays (X and Y) are not in cache, then the performance of this loop on most modern microprocessors would be limited by the available memory bandwidth on the machine.

13.3 IA-64 Floating-point Features

This section highlights IA-64 features that reduce the impact of the performance limiters described in Section 13.2 using illustrative examples.

13.3.1 Large and Wide Floating-point Register Set

As machine cycle times are reduced, the latency in cycles of the execution units generally increases. As latency increases, register pressure due to multiple operations in-flight also increases. Furthermore as multiple execution units are added, the register pressure increases similarly since even more instructions can be in-flight at any one time.

IA-64 provides 128 directly addressable floating-point registers to enable data reuse and to reduce the number of load/store operations required due to an insufficient number of registers. This reduction in the number of loads and stores can increase performance by changing a computation from being memory operation (MOP) limited to being floating-point operation (FLOP) limited. Consider the dense matrix multiply code below:

```
     DO 1 i = 1, N
         DO 1 j = 1, P
             DO 1 k = 1, M
    1            C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

In the inner loop (k), 2 loads are required for every multiply and add operation. The MOP:FLOP ratio is therefore 1:1.

```
L1: ldfd    f5  = [r5], 8      // Load A[i,k]
    ldfd    f6  = [r6], 8      // Load B[k,j]
    fma.d.s0 f7= f5, f6, f7   // *,+ to C[i,j]
    br.cloop L1
```

Here, three registers are required to hold the operands (`f5`, `f6`) and the accumulator (`f7`). By recognizing the reuse of `A[i,k]` for different `B[k,j]` as `j` is varied, and the reuse of `B[k,j]` for different `A[i,k]` as `i` is varied, the computation can be restructured as:

```
    DO 1 i = 1, N
        DO 1 j = 1, P
            DO 1 k = 1, M
                C[i  ,j  ] = C[i  ,j  ]
                            + A[i  ,k]*B[k,j  ]
                C[i+1,j  ] = C[i+1,j  ]
                            + A[i+1,k]*B[k,j  ]
                C[i  ,j+1] = C[i  ,j+1]
                            + A[i  ,k]*B[k,j+1]
   1            C[i+1,j+1] = C[i+1,j+1]
                            + A[i+1,k]*B[k,j+1]
```

Now, for every 4 loads, 4 multiplies and adds can be performed, thus changing the MOP:FLOP ratio to 1:2. However, 8 registers are now required: 4 for the accumulators and 4 for the operands.

```
    add     r6 = r5, 8
    add     r8 = r7, 8
L1: ldfd    f5 = [r5], 16          // Load A[i,k]
    ldfd    f6 = [r6], 16          // Load A[i+1,k]
    ldfd    f7 = [r7], 16          // Load B[k,j]
    ldfd    f8 = [r8], 16          // Load B[k,j+1]
    fma.d.s0 f9 = f5, f7, f9       // *,+ on C[i,j]
    fma.d.s0 f10 = f5, f8, f10     // *,+ on C[i,j+1]
    fma.d.s0 f11 f6, f7, f11       // *,+ on C[i+1,j]
    fma.d.s0 f12 = f6, f7, f12     // *,+ on C[i+1,j+1]
    br.cloop L1
```

With 128 available registers, the outer loops of `i` and `j` could be unrolled by 8 each so that 64 multiplies and adds can be performed by loading just 16 operands.

The floating-point register file is divided into two regions: a static region (`f0-f31`) and a rotating region (`f32-f127`). The register rotation provides the automatic register renaming required to create compact kernel-only software-pipelined code. Register rotation also enables scheduling software pipelined code with an initiation interval that is less than the longest latency operation. For e.g., consider the simple vector add loop shown below:

```
    DO 1 i = 1, N
   1    A[i] = B[i] + C[i]
```

The basic inner loop is:

```
L1: ldf     f5  = [r5], 8          // Load B[i]
    ldf     f6  = [r6], 8          // Load C[i]
    fadd    f7  = f5, f6           // Add operands
    stf     [r7]= f7, 8            // Store A[i]
    br.cloop L1
```

If we suppose the minimum floating-point load latency is 9 clocks, and 2 memory operations can be issued per clock, the above loop has to be unrolled by at least six if there is no register rotation.

```
           add  r8  = r7, 8
L1: (p18)  stf  [r7] = f25, 16    // Cycle 17,26 ...
    (p18)  stf     [r8] = f26, 16    // Cycle 17,26 ...
    (p17)  fadd    f25  = f5, f15    // Cycle 8,17,26 ...
    (p16)  ldf     f5 = [r5], 8      // Cycle 0,9,18 ...
    (p16)  ldf     f15 = [r6], 8     // Cycle 0,9,18 ...
    (p17)  fadd    f26  = f6, f16 ;; // Cycle 9,18,27 ...
    (p16)  ldf     f6  = [r5], 8     // Cycle 1,10,19 ...
    (p16)  ldf     f16  = [r6], 8    // Cycle 1,10,19 ...
    (p18)  stf     [r7] = f27, 16    // Cycle 20,29 ...
    (p18)  stf     [r8] = f28, 16    // Cycle 20,29 ...
    (p17)  fadd    f27 = f7, f17 ;;  // Cycle 11,20 ...
    (p16)  ldf     f7  = [r5], 8     // Cycle 3,12,21 ...
    (p16)  ldf     f17  = [r6], 8    // Cycle 3,12,21 ...
    (p17)  fadd    f28  = f8, f18 ;; // Cycle 12,21 ...
    (p16)  ldf     f8   = [r5], 8    // Cycle 4,13,22 ...
    (p16)  ldf     f18  = [r6], 8    // Cycle 4,13,22 ...
    (p18)  stf     [r7] = f29, 16    // Cycle 23,32 ...
    (p18)  stf     [r8] = f30, 16    // Cycle 23,32 ...
    (p16)  fadd    f29  = f9, f19 ;; // Cycle 14,23 ...
    (p16)  ldf     f9 = [r5], 8      // Cycle 6,15,24 ...
    (p16)  ldf     f19  = [r6], 8    // Cycle 6,15,24 ...
    (p16)  fadd    f30  = f10, f20 ;; // Cycle 15,24 ...
    (p16)  ldf     f10  = [r5], 8    // Cycle 7,16,25 ...
    (p16)  ldf     f20  = [r6], 8    // Cycle 7,16,25 ...
           br.ctop L1 ;;
```

However, with register rotation, the same loop can be scheduled with an initiation interval of just 2 clocks without unrolling (and 1.5 clocks if unrolled by 2):

```
L1: (p24)  stf     [r7] = f57, 8     // Cycle 15,17 ...
    (p21)  fadd    f57  = f37, f47   // Cycle 9,11,13 ...
    (p16)  ldf     f32  = [r5], 8    // Cycle 0,2,4,6 ...
    (p16)  ldf     f42  = [r6], 8    // Cycle 0,2,4,6 ...
           br.ctop L1 ;;
```

It is thus often advantageous to modulo schedule and then unroll (if required). Please see Chapter 12 on software pipelining for details on how to rewrite loops using this transformation.

13.3.1.1 Notes on FP Precision

The floating-point registers are 82 bits wide with 17 bits for exponent range, 64 bits for significand precision and 1 sign bit. During computation, the result range and precision is determined by the computational model chosen by the user. The computational model is indicated either statically in the instruction encoding, or dynamically via the precision control (PC) and widest-range-exponent (WRE) bits in the floating-point status register. Using an appropriate computational model, the user can minimize the error accumulation in the computation. In the above matrix multiply example, if the multiply and add computations are performed in full register file range and precision, the results (in accumulators) can hold 64 bits of precision and up to 17 bits of range for inputs that might be single precision numbers. With the rounding performed at the 64th precision bit (instead of the 24th for single precision) a smaller error is accumulated with each multiply and add. Furthermore, with 17 bits of range (instead of 8 bits for single precision) large positive and negative products can be added to the accumulator without overflow or underflow. In addition to providing more accurate results the extra range and precision can often enhance the performance of iterative computations that are required to be performed until convergence (as indicated by an error bound) is reached.

13.3.2 Multiply-Add Instruction

IA-64 defines the fused multiply-add (`fma`) as the basic floating-point computation, since it forms the core of many computations (linear algebra, series expansion, etc.) and its latency in hardware is typically less than the sum of the latencies of an individual multiply operation (with rounding) implementation and an individual add operation (with rounding) implementation.

In computational loops that have a loop carried dependence and whose speed is often determined by the latency of the floating-point computation rather than the peak computational rate, the multiply-add operation can often be used advantageously. Consider the Livermore FORTRAN Kernel 9 – General Linear Recurrence Equations:

```
   DO 191 k= 1,n
       B5(k+KB5I)= SA(k) + STB5 * SB(k)
       STB5= B5(k+KB5I) - STB5
191CONTINUE
```

Since there is a true data dependence between the two statements on variable `B5(k+KB5I))` and a loop-carried dependence on variable `STB5`, the loop number of clocks per iteration is entirely determined by the latency of the floating-point operations. In the absence of an `fma` type operation, and assuming that the individual multiply and add latencies are 5 clocks each and the loads are 8 cycles, the loop would be:

```
L1: (p16)   ldf     f32 = [r5], 8         // Load SA(k)
    (p16)   ldf     f42  = [r6], 8        // Load SB(k)
    (p17)   fmul    f5  = f7, f43;;       // tmp,Clk 0,15 ...
    (p17)   fadd    f6  = f33, f5 ;;      // B5,Clk 5,20 ...
    (p17)   stf     [r7] = f6, 8          // Store B5
    (p17)   fsub    f7  = f6, f7          // STB5,Clk 10,25 ..
            br.ctop L1 ;;
```

With an `fma`, the overall latency of the chain of operations decreases and assuming a 5 cycle `fma`, the loop iteration speed is now 10 clocks (as opposed to 15 clocks above).

```
L1: (p16)   ldf     f32  = [r5], 8        // Load SA(k)
    (p16)   ldf     f42 = [r6], 8         // Load SB(k)
    (p17)   fma     f6  = f7, f43, f33;;  // B5,Clk 0,10 ...
    (p17)   stf     [r7] = f6, 8          // Store B5
    (p17)   fsub    f7  = f6, f7          // STB5,Clk 5,15 ..
            br.ctop L1 ;;
```

The fused multiply-add operation also offers the advantage of a single rounding error for the pair of computations which is valuable when trying to compute small differences of large numbers.

13.3.3 Software Divide/Square-root Sequence

To perform division or square root operations on IA-64, a software based sequence of operations is used. The sequence consists of obtaining an initial guess (using `frcpa`/`frsqrta` instruction) and then refining the guess by performing Newton-Raphson iterations until the error is sufficiently small so that it may not affect the rounding of the result. Examples of double precision divide and square root sequences, optimized for latency and throughput, are provided below.

Note: For reduced precision, square and divide sequences can be completed with even fewer instructions.

13.3.3.1 Double Precision – Divide

Divide (Max Throughput) (10 Instructions, 8 Groups)	Divide (Min Latency) (13 Instructions, 7 Groups)
frcpa.s0 f8,p6 = f6,f7 ;; (p6) fnma.s1 f9 = f7,f8,f1 ;; (p6) fma.s1 f8 = f9,f8,f8 (p6) fma.s1 f9 = f9,f9,f0 ;; (p6) fma.s1 f8 = f9 ,f8,f8 (p6) fma.s1 f9 = f9,f9,f0 ;; (p6) fma.s1 f8 = f9,f8,f8 ;; (p6) fma.d.s1 f9 = f6,f8,f0 ;; (p6) fnma.d.s1 f6 = f7,f9,f6 ;; (p6) fma.d.s0 f8 = f6,f8,f9	frcpa.s0 f8,p6 = f6,f7 ;; (p6) fma.s1 f9 = f6,f8,f0 (p6) fnma.s1 f10 = f7,f8,f1 ;; (p6) fma.s1 f9 = f10,f9,f9 (p6) fma.s1 f11 = f10,f10,f0 (p6) fma.s1 f8 = f10,f8,f8 ;; (p6) fma.s1 f9 = f11,f9,f9 (p6) fma.s1 f10 = f11,f11,f0 (p6) fma.s1 f8 = f11,f8,f8 ;; (p6) fma.d.s1 f9 = f10,f9,f9 (p6) fma.s1 f8 = f10,f8,f8 ;; (p6) fnma.d.s1 f6 = f7,f9,f6 ;; (p6) fma.d.s0 f8 = f6,f8,f9

13.3.3.2 Double Precision – Square-root

Square-root (Max Throughput) (15 Instructions, 13 Groups)	Square-root (Min Latency) (17 Instructions, 12 Groups)
frsqrta.s0 f8,p6 = f6 (p6) fma.s1 f10 = f7,f6,f0 ;; (p6) fma.s1 f9 = f8,f8,f0 ;; (p6) fnma.s1 f9 = f9,f10,f7 ;; (p6) fma.s1 f8 = f9,f8,f8 ;; (p6) fma.s1 f9 = f8,f10,f0 ;; (p6) fnma.s1 f9 = f9,f8,f7 ;; (p6) fma.s1 f8 = f9,f8,f8 ;; (p6) fma.s1 f9 = f8,f10,f0 ;; (p6) fnma.s1 f9 = f9,f8,f7 ;; (p6) fma.s1 f8 = f9,f8,f8 ;; (p6) fma.d.s1 f9 = f6,f8,f0 (p6) fma.s1 f8 = f7,f8,f0 ;; (p6) fnma.s1 f6 = f9,f9,f6 ;; (p6) fma.d.s0 f8 = f6,f8,f9	frsqrta.s0 f8,p6 = f6 (p6) fma.s1 f9 = f7,f6,f0 ;; (p6) fma.s1 f10 = f8,f8,f0 ;; (p6) fnma.s1 f10 = f10,f9,f7 ;; (p6) fma.s1 f8 = f10,f8,f8 ;; (p6) fma.s1 f10 = f8,f9,f0 ;; (p6) fnma.s1 f10 = f10,f8,f7 ;; (p6) fma.s1 f8 = f10,f8,f8 ;; (p6) fma.s1 f10 = f6,f8,f0 (p6) fma.s1 f9 = f8,f9,f0 (p6) fma.s1 f11 = f7,f8,f0 ;; (p6) fnma.s1 f12 = f10,f10,f6 (p6) fnma.s1 f7 = f9,f8,f7 ;; (p6) fma.s1 f8 = f12,f11,f10 (p6) fma.s1 f7 = f7,f11,f11 ;; (p6) fnma.s1 f6 = f8,f8,f6 ;; (p6) fma.d.s0 f8 = f6,f7,f8

The first instruction (`frcpa`) provides an approximation (good to 8 bits) of the reciprocal of `f7` and sets the predicate (`p6`) to 1, if the ratio `f6`/`f7` can be obtained using the prescribed Newton-Raphson iterations. If, however, the ratio `f6`/`f7` is special (finite/0, finite/infinite, etc) the final result of `f6`/`f7` is provided in `f8` and the predicate (`p6`) is cleared. For certain boundary conditions (when the operand values (`f6` and `f7`) are well outside the single precision, double precision and even double-extended precision ranges) frcpa will cause a software assist fault and the software handler will produce the ratio `f6`/`f7` and return it in `f8` and clear the predicate (`p6`).

The multiple status fields provided in the FPSR are used in these sequences. S0 is the main (architectural) status field and it is written to by the first operation (`frcpa`) to signal any faults (V, Z, D), and by the last operation to signal any traps. The conditions of all intermediate operations are ignored by writing them to S1. Thus these sequences not only obtain the correct IEEE 754 specified result (in `f8`) but the flags are also set (in S0) as per the standard's requirements. If the divide is part of a speculative chain of operations that is using S2 as its status field, then S0 should be replaced with S2 in these sequences. S1 can still be used by the intermediate operations of all the divide sequences (i.e., those that target S0, S2, or S3) since those flags are all discarded.

When divides and square-root appear in vectorizable loops, it is often very advantageous to have these operations be performed in software rather than hardware. In software, these operations can be pipelined and the overall throughput be improved, whereas in hardware these operations are typically not pipelineable.

Another significant advantage of the software based divide/square-root computations is that the accuracy of the result can be controlled by the user and can be traded off for speed. This trade-off is often used in graphics codes where the divide accuracy of about 14-bits suffices and the sequence can be shorter than that used for single or double precision.

13.3.4 Computational Models

IA-64 offers complete user control of the computational model. The user can select the result's precision and range, the rounding mode, and the IEEE trap response. Appropriately selecting the computational model can result in code that has greater accuracy, is higher performance, or both.

The register file format is uniform for the 3 memory data types – single, double and double-extended. Since all the computations are performed on registers (regardless of the data type of its content) operands of different types can be easily combined. Also since the conversion from the memory type to the register file format is done on loads automatically no extra operations are required to perform the format conversion.

The C syntax semantics are also easily emulated. Loads convert all input operands into the register file format automatically. Data operands of different types, now residing in register file format can be operated upon and all intermediate results coerced to double precision by statically indicating the result precision in the instruction encoding. The computation leading to the final result can specify the result precision and range. (statically in the instruction encoding for single and double precision, and dynamically in the status field bits for double-extended precision). Compliance to the IA32 FP computational style (range=extended, precision=single/double/extended) can also achieved using the status field bits.

13.3.5 Multiple Status Fields

The FPSR is divided into 1 main (architectural) status field and 3 additional identical status fields. These additional status fields could be used to performance advantage.

First, divide and square-root sequences (described in Section 13.3.3) contain operations that might cause intermediate results to overflow/underflow or be inexact even if the final result may not. In order to maintain correct IEEE flag status the status flags of these computations need to be discarded. One of these additional status fields (typically status field 1) can be used to discard these flags.

Second, speculating floating-point operations requires maintaining the status flags of the speculated operations distinct from the architectural status flags until the speculated operations are committed to architectural state (if they ever are). One of these additional status fields (typically status fields 2 and even 3) can be used for this purpose.

Consider the Livermore FORTRAN kernel 16 – Monte Carlo Search

```
  DO 470 k= 1,n
      k2= k2+1
      j4= j2+k+k
      j5= ZONE(j4)
      IF( j5-n      ) 420,475,450
415   IF( j5-n+II   ) 430,425,425
420   IF( j5-n+LB   ) 435,415,415
425   IF( PLAN(j5)-R) 445,480,440
430   IF( PLAN(j5)-S) 445,480,440
435   IF( PLAN(j5)-T) 445,480,440
440   IF( ZONE(j4-1)) 455,485,470
445   IF( ZONE(j4-1)) 470,485,455
450   k3= k3+1
      IF( D(j5)-(D(j5-1)*(T-D(j5-2))**2
   ,      +(S-D(j5-3))**2
   ,         +(R-D(j5-4))**2)) 445,480,440
455   m= m+1
      IF( m-ZONE(1) ) 465,465,460
460   m= 1
465   IF( i1-m) 410,480,410
470   CONTINUE
475   CONTINUE
480   CONTINUE
485   CONTINUE
```

Profiling indicates that the conditional after statement 450 is most frequently executed. It is therefore advantageous to speculatively execute the computation in the conditional while the conditionals in 415...445 are being evaluated. In the event that any of the conditionals in 415...445 cause the control to be moved on beyond 450.

The availability of multiple additional status fields can allow a user to maintain multiple computational environments and to dynamically select among them on an operation by operation basis. A common use is in the implementation of interval arithmetic code where each primitive operation is required to be computed in two different rounding modes to determine the interval of the result.

13.3.6 Other Features

IA-64 offers a number of other architectural constructs to enhance the performance of different computational situations.

13.3.6.1 Operand Screening Support

Operand screening is often a required or useful step prior to a computation. The operand may be screened to ensure that it is in a valid range (e.g. finite positive or zero input to square-root, non-zero divisor) or it may be screened to take an early out – the result of the computation is predetermined or could be computed more efficiently in another way. The `fclass` instruction can be used to classify the input operand to either be or not be a part of a set of classes. Consider the following code used for screening invalid operands for square-root computation:

```
IF (A .LT. 0.0D0 .OR. A .GT. MAXREAL) THEN
   WRITE (*, “INVALID INPUT OPERAND”)
ELSE
   WRITE (*, “SQUARE-ROOT = “, SQRT(A))
ENDIF
```

The above conditional can be determined by a single fclass instruction as indicated below:

```
fclass [fill in the details here]
```

The resultant complimentary predicates can be used to control the `THEN` and `ELSE` statements individually.

13.3.6.2 Min/Max/AMin/AMax

IA-64 provides direct instruction level support for the FORTRAN intrinsic `MIN(a,b)` or the equivalent C idiom: `a<b ? a : b` and the FORTRAN intrinsic `MAX(a,b)` or the equivalent C idiom: `a<b ? b : a`. These instructions can enhance performance by avoiding the function call overhead in FORTRAN, and by reducing the critical path in C. The instruction is designed not to be commutative so that by appropriately selecting the input operand order, the user can either ignore or catch NaNs.

Consider the problem of finding the minimum value in an array (similar to the Livermore FORTRAN kernel 24)

```
   XMIN = X(1)
   DO 24  k= 2,n
24    IF(X(k) .LT. XMIN)  XMIN = X(k)
```

Since NaNs are unordered, comparison with NaNs (including LT) will return false. Hence if the above code is implemented as:

```
    ldf     f5  = r5, 8 ;;
L1: ldf     f6  = r5, 8
    fmin    f5  = f6, f5
    br.cloop L1 ;;
```

If the value in the array `X` (loaded in `f6`) is a NaN, the new minimum value (in `f5`) will remain unchanged, since the NaN will fail the `.LT.` comparison and `fmin` will return the second argument – in this case the old minimum value in `f5`.

However, if the code is implemented as:

```
    ldf     f5  = r5, 8 ;;
L1: ldf     f6  = r5, 8
    fmin    f5  = f5, f6
    br.cloop L1 ;;
```

Now, if the value in the array `X` (loaded in `f6`) is a NaN, the new minimum value (in `f5`) will be set to the NaN, since the NaN will fail the `.LT.` comparison and fmin will return the second argument – in this case the NaN in `f6`.

`famin`/`famax` perform the comparison on the absolute value of the input operands (i.e. they ignore the sign bit) but otherwise operate in the same (non-commutative) way as the `fmin`/`fmax` instructions.

13.3.6.3 Integer/Floating-point Conversion

Unsigned integers are converted to their equivalently valued floating-point representations by simply moving the integer to the significand field of the floating-point register using the setf.sig instruction. The resulting floating-point value would be in its unnormal representation (unless the unsigned integer was greater than 263).

Conversions from signed integers to floating-point and from floating-point to signed or unsigned integers are accomplished by `fcvtxf` and `fcvtfx`/`fcvtfxu` instructions respectively. However, since signed integers are converted directly to their canonical floating-point representations, they do not need to be normalized after conversion.

13.3.6.4 FP Subfield Handling

It is sometimes useful to assemble a floating-point value from its constituent fields. Multiplication and division of floating-point values by powers of two, for example, can be easily accomplished by appropriately adjusting the exponent. IA-64 provides instructions that allow moving floating-point fields between the integer and floating-point register files. Division of a floating-point number by 2.0 is accomplished as follows:

```
    getf.exp    r5  = f5            // Move S+Exp to int
    add         r5  = r5, -1        // Sub 1 from Exp
    setf.exp    f6  = r5            // Move S+Exp to FP
    fmerge.se   f5  = f6, f5        // Merge S+E w/ Mant
```

Floating-point values can also be constructed from fields from different floating-point registers.

13.3.7 Memory Access Control

Recognizing the trend of growing memory access latency, and the implementation costs of high bandwidth, IA-64 incorporates many architectural features to help manage the memory hierarchy and increase performance. As described in Section 13.2, memory latency and bandwidth are significant performance limiters in floating-point applications. IA-64 offers features to address both these limitations.

In order to enhance the core bandwidth to the floating-point register file, IA-64 defines load-pair instructions. In order to mitigate the memory latency, IA-64 defines explicit and implicit data prefetch instructions. In order to maximize the utilization of caches, IA-64 defines locality attributes as part of memory access instructions to help control the allocation (and de-allocation) of data in the caches. For instances where the instruction bandwidth may become a performance limiter, IA-64 defines machine hints to trigger relevant instruction prefetches.

13.3.7.1 Load-pair Instructions

The floating-point load pair instructions enable loading two contiguous values in memory to two independent floating-point registers. The target registers are required to be odd and even physical registers so that the machine can utilize just one access port to accomplish the register update.

Note: The odd/even pair restriction is on physical register numbers, not logical register numbers. A programming violation of this rule will cause an illegal operation fault).

For example, suppose a machine provides sufficient bandwidth from L1 to sustain 2 load-pairs every cycle. Then loops that require up to 2 data elements (of 8 bytes each) per floating-point instruction can run at peak speeds when the data is resident in L1. A common example of such a case is a simple double precision dot product – DDOT:

```
   DO 1 I = 1, N
 1    C = C + A(I) * B(I)
```

The inner loop consists of two loads (for `A` and `B`) and an (to accumulate the product on C). The loop would run at the latency of the fma due to the recurrence on C. In order to break the recurrence on C, the loop is typically unrolled and multiple partial accumulators are used.

```
   DO 1 I = 1, N, 8
      C1 = C1 + A[I] * B[I]
      C2 = C2 + A[I+1] * B[I+1]
      C3 = C3 + A[I+2] * B[I+2]
      C3 = C3 + A[I+3] * B[I+3]
      C3 = C3 + A[I+4] * B[I+4]
      C3 = C3 + A[I+5] * B[I+5]
      C3 = C3 + A[I+6] * B[I+6]
 1    C4 = C4 + A[I+7] * B[I+7]
   C = C1 + C2 + C3 + C4
```

If normal (non-double pair) loads are used, the inner loop would consist of 16 loads and 8 fmas. If we assume the machine has two memory ports, this loop would be limited by the availability of M slots and run at a peak rate of 1 clock per iteration. However, if this loop is rewritten using 8 load-pairs (for `A[I]`, `A[I+1]` and `B[I]`, `B[I+1]` and `A[I+2]`, `A[I+3]` and `B[I+2]`, `B[I+3]` and so on) and 8 fmas this loop could run at a peak rate of 2 iterations per clock (or just 0.5 clocks per iteration) with just two M-units.

13.3.7.2 Data Prefetch

`lfetch` allows the advance prefetching of a line (defined as 32 bytes or more) of data into the cache from memory. Allocation hints can be used to indicate the nature of the locality of the subsequent accesses on that data and to indicate as to which level of cache that data needs to be promoted to.

While regular loads can also be used to achieve the effect of data prefetching, (if the load target is never used) lfetches can more effectively reduce the memory latency without using floating-point registers as targets of the data being prefetched. Furthermore `lfetch` allows prefetching the data to different levels of caches.

13.3.7.3 Allocation Control

Since data accesses have different locality attributes (temporal/non-temporal, spatial/non-spatial), IA-64 allows annotating the data accesses (loads/stores) to reflect these attributes. Based on these annotations, the implementation can better manage the storage of the data in the caches.

Temporal and Non-temporal hints are defined. These attributes are applicable to the various cache levels. (Only two cache levels are architecturally identified). The non temporal hint is best used for data that typically has no reuse with respect to that level of cache. The temporal hint is used for all other data (that has reuse).

13.4 Summary

This chapter describes the limiting factors for many scientific and floating-point applications: memory latency and bandwidth, functional unit latency, and number of available functional units. It also describes the important features of IA-64 floating-point support beyond the software-pipelining support described in Chapter 12, "Software Pipelining and Loop Support" that help to overcome some of these performance limiters. Architectural support for speculation, rounding, and precision control are also described.

Examples in the chapter include how to implement floating-point division and square root, common scientific computations such as reductions, use of features such as the `fma` instruction, and various Livermore kernels.

Part III: Appendices

Instruction Sequencing Considerations A

Instruction execution consists of four phases:

1. Read the instruction from memory (*fetch*).
2. Read architectural state, if necessary (*read*).
3. Perform the specified operation (*execute*).
4. Update architectural state, if necessary (*update*).

An **instruction group** is a sequence of instructions starting at a given bundle address and slot number and including all instructions at sequentially increasing slot numbers and bundle addresses up to the first stop or taken branch. For the instructions in an instruction group to have well-defined behavior, they must meet the ordering and dependency requirements described below.

If the instructions in instruction groups meet the resource-dependency requirements, then the behavior of a program will be as though each individual instruction is sequenced through these phases in the order listed above. The order of a phase of a given instruction relative to any phase of a previous instruction is prescribed by the instruction sequencing rules below.

- There is no a priori relationship between the *fetch* of an instruction and the *read*, *execute*, or *update* of any dynamically previous instruction. The `sync.i` and `srlz.i` instructions can be used to enforce a sequential relationship between the *fetch* of all succeeding instructions and the *update* of all previous instructions.
- Between instruction groups, every instruction in a given instruction group will behave as though its read occurred after the update of all the instructions from the previous instruction group. All instructions are assumed to have unit latency. Instructions on opposing sides of a stop are architecturally considered to be separated by at least one unit of latency.

  Some system state updates require more stringent requirements than those described here.
- Within an instruction group, every instruction will behave as though its read of the memory and ALAT state occurred after the update of the memory and ALAT state of all prior instructions in that instruction group.
- Within an instruction group, every instruction will behave as though its read of the register state occurred before the update of the register state by any instruction (prior or later) in that instruction group, except as noted in the Register dependencies and Memory dependencies described below.

The ordering rules above form the context for register dependency restrictions, memory dependency restrictions and the order of exception reporting. These dependency restrictions apply only between instructions whose resource reads and writes are not dynamically disabled by predication.

- Register dependencies: Within an instruction group, read-after-write (RAW) and write-after-write (WAW) register dependencies are not allowed (except as noted in "RAW Ordering Exceptions" on page A-2 and "WAW Ordering Exceptions" on page A-3). Write-after-read (WAR) register dependencies are allowed (except as noted in "WAR Ordering Exceptions" on page A-4).

These dependency restrictions apply to both explicit register accesses (from the instruction's operands) and implicit register accesses (such as application and control registers implicitly accessed by certain instructions). Predicate register PR0 is excluded from these register dependency restrictions, since writes to PR0 are ignored and reads always return 1 (one).

- Memory dependencies: Within an instruction group, RAW, WAW, and WAR memory dependencies and ALAT dependencies are allowed. A load will observe the results of the most recent store to the same memory address. In the event that multiple stores to the same address are present in the same instruction group, memory will contain the result of the latest store after execution of the instruction group. A store following a load to the same address will not affect the data loaded by the load. Advanced loads, check loads, advanced load checks, stores, and memory semaphore instructions implicitly access the ALAT. RAW, WAW, and WAR ALAT dependencies are allowed within an instruction group and behave as described for memory dependencies.

The net effect of the dependency restrictions stated above is that a processor may execute all (or any subset) of the instructions within a legal instruction group concurrently or serially with the end result being identical. If these dependency restrictions are not met, the behavior of the program is undefined.

The instruction sequencing resulting from the rules stated above is termed sequential execution.

The ordering rules and the dependency restrictions allow the processor to dynamically re-order instructions, execute instructions with non-unit latency, or even concurrently execute instructions on opposing sides of a stop or taken branch, provided that correct sequencing is enforced and the appearance of sequential execution is presented to the programmer.

IP is a special resource in that reads and writes of IP behave as though the instruction stream was being executed serially, rather than in parallel. RAW dependencies on IP are allowed, and the reader gets the IP of the bundle in which it is contained. So, each bundle being executed in parallel logically reads IP, increments it and writes it back. WAW is also allowed.

Ignored ARs are not exceptional for dependency checking purposes. RAW and WAW dependencies to ignored ARs are not allowed.

A.1 RAW Ordering Exceptions

There are four exceptions to the rule prohibiting RAW register dependencies within an instruction group. These exceptions are the `alloc` instruction, check load instructions, instructions that affect branching, and the `ld8.fill` and `st8.spill` instructions.

- The `alloc` instruction implicitly writes the Current Frame Marker (CFM) which is implicitly read by all instructions accessing the stacked subset of the general register file. Instructions that access the stacked subset of the general register file may appear in the same instruction group as alloc and will see the stack frame specified by the `alloc`.

Note: Some instructions have RAW or WAW dependences on resources other than CFM affected by `alloc` and are thus not allowed in the same instruction group after an `alloc`: `flushrs`, move from AR[BSPSTORE], move from AR[RNAT], `br.cexit`, `br.ctop`, `br.wexit`, `br.wtop`, `br.call`, `br.ia`, `br.ret`, `clrrrb`. Also note that `alloc` is required to be the first instruction in an instruction group.

- A check load instruction may or may not perform a load since it is dependent upon its corresponding advanced load. If the check load misses the ALAT it will execute a load from memory. A check load and a subsequent instruction that reads the target of the check load may exist in the same instruction group. The dependent instruction will get the new value loaded by the check load.

- A branch may read branch registers and may implicitly read predicate registers, the LC, EC, and PFS application registers, as well as CFM. Except for LC, EC and predicate registers, writes to any of these registers by a non-branch instruction will be visible to a subsequent branch in the same instruction group. Writes to predicate registers by any non-floating-point instruction will be visible to a subsequent branch in the same instruction group. RAW register dependencies within the same instruction group are not allowed for LC and EC. Dynamic RAW dependencies where the predicate writer is a floating-point instruction and the reader is a branch are also not allowed within the same instruction group. Branches `br.cond`, `br.call`, `br.ret` and `br.ia` work like other instructions for the purposes of register dependency; i.e., if their qualifying predicate is 0, they are not considered readers or writers of other resources. Branches `br.cloop`, `br.cexit`, `br.ctop`, `br.wexit`, and `br.wtop` are exceptional in that they are always readers or writers of their resources, regardless of the value of their qualifying predicate.
- The `ld8.fill` and `st8.spill` instructions implicitly access the User NaT Collection application register (UNAT). For these instructions the restriction on dynamic RAW register dependencies with respect to UNAT applies at the bit level. These instructions may appear in the same instruction group provided they do not access the same bit of UNAT. RAW UNAT dependencies between `ld8.fill` or `st8.spill` instructions and mov ar= or mov =ar instructions accessing UNAT must not occur within the same instruction group.

For the purposes of resource dependencies, CFM is treated as a single resource.

A.2 WAW Ordering Exceptions

There are three exceptions to the rule prohibiting WAW register dependencies within an instruction group. The exceptions are compare-type instructions, floating-point instructions, and the `st8.spill` instruction.

- The set of compare-type instructions includes: `cmp`, `cmp4`, `tbit`, `tnat`, `fcmp`, `frsqrta`, `frcpa`, and `fclass`. Compare-type instructions in the same instruction group may target the same predicate register provided:
    - — The compare-type instructions are either all AND-type compares or all OR-type compares (AND-type compares correspond to ".and" and ".andcm" completers; OR-type compares correspond to ".or" and ".orcm" completers), or
    - — The compare-type instructions all target PR 0. All WAW dependencies for PR 0 are allowed; the compares can be of any types and can be of differing types.

All other WAW dependencies within an instruction group are disallowed, including dynamic WAW register dependencies with move to PR instructions that access the same predicate registers as another writer.

Note: The move to PR instruction only writes those PRs indicated by its mask, but the move from PR instruction always reads all the predicate registers.

- Floating-point instructions implicitly write the Floating-point Status Register (FPSR) and the Processor Status Register (PSR). Multiple floating-point instructions may appear in the same instruction group since the restriction on WAW register dependencies with respect to the FPSR and PSR do not apply. The state of FPSR and PSR after executing the instruction group will be the logical OR of all writes.

- The st8.spill instruction implicitly writes the UNAT register. For this instruction the restriction on WAW register dependencies with respect to UNAT applies at the bit level. Multiple st8.spill instructions may appear in the same instruction group provided they do not write the same bit of UNAT. WAW register dependencies between st8.spill instructions and mov ar= instructions targeting UNAT must not occur within the same instruction group.

WAW dependencies to ignored ARs are not allowed.

A.3 WAR Ordering Exceptions

WAR dependence between the reading of PR63 by a branch instruction and the subsequent writing of PR63 by a loop closing branch (br.ctop, br.cexit, br.wtop, or br.wexit) in the same instruction group is not allowed. Otherwise, WAR dependencies are allowed.

IA-64 Pseudo-Code Functions B

Table B-1 contains all pseudo-code functions used in Chapter 7, "IA-64 Instruction Reference".

Table B-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_write(rtype, raddr, paddr, size)	Allocates a new ALAT entry using the register type specified by `rtype`, the register address specified by `raddr`, the physical memory address specified by `paddr`, and the access size specified by `size`. No value is returned. This function guarantees that only one ALAT entry exists for a given `raddr`. If a `ld.c.nc`, `ldf.c.nc`, or `ldfp.c.nc` instruction's `raddr` matches an existing ALAT entry's register tag, but the instruction's `size` and/or `paddr` are different than that of the existing entry's; then this function may either preserve the existing entry, or invalidate it and write a new entry with the instruction's specified `size` and `paddr`.
check_target_register(r1)	If r1 targets an out-of-frame stacked register (as defined by CFM), an illegal operation fault is delivered, and this function does not return.
check_target_register_sof(r1, newsof)	If r1 targets an out-of-frame stacked register (as defined by the newsof parameter), an illegal operation fault is delivered, and this function does not return.
concatenate2(x1, x2)	Concatenates the lower 32 bits of the 2 arguments, and returns the 64-bit result.
concatenate4(x1, x2, x3, x4)	Concatenates the lower 16 bits of the 4 arguments, and returns the 64-bit result.

Table B-1. Pseudo-Code Functions (Cont'd)

Function	Operation
concatenate8(x1, x2, x3, x4, x5, x6, x7, x8)	Concatenates the lower 8 bits of the 8 arguments, and returns the 64-bit result.
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(fr2, fr3, frel, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the fcmp instruction.
fcvt_fx_exception_fault_check(fr2, trunc, sf *tmp_fp_env)	Checks for all floating-point faulting conditions for the fcvt.fx and fcvt.fx.trunc instructions. It propagates NaNs, and NaTVals.
fcvt_fxu_exception_fault_check(fr2 , trunc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the fcvt.fxu and fcvt.fxu.trunc instructions. It propagates NaNs, and NaTVals.
fma_exception_fault_check(fr2, fr3, fr4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the fma instruction. It propagates NaNs, NaTVals, and special IEEE results.
fminmax_exception_fault_check(fr2, fr3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `famax`, `famin`, `fmax`, and `fmin` instructions.
fms_fnma_exception_fault_check(fr2, fr3, fr4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fms` and `fnma` instructions. It propagates NaNs, NaTVals, 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_is_nan_or_inf(freg)	Returns true if the floating-point exception_fault_check functions returned a IEEE fault disabled default result or a propagated NaN.
fp_equal(fr1, fr2)	IEEE standard equality relationship test.
fp_ieee_recip(num, den)	Returns the true quotient for special sets of operands, or an approximation to the reciprocal of the divisor to be used in the software divide algorithm.
fp_ieee_recip_sqrt(root)	Returns the true square root result for special operands, or an approximation to the reciprocal square root to be used in the software square root algorithm.
fp_is_nan(freg)	Returns true when floating register contains a NaN.
fp_is_natval(freg)	Returns true when floating register contains a NaTVal
fp_is_normal(freg)	Returns true when floating register contains a normal number.
fp_is_pos_inf(freg)	Returns true when floating register contains a positive infinity.
fp_is_qnan(freg)	Returns true when floating register contains a quiet NaN.
fp_is_snan(freg)	Returns true when floating register contains a signalling NaN.
fp_is_unorm(freg)	Returns true when floating register contains an unnormalized number.
fp_is_unsupported(freg)	Returns true when floating register contains an unsupported format.
fp_less_than(fr1, fr2)	IEEE standard less-than relationship test.
fp_lesser_or_equal(fr1, fr2)	IEEE standard less-than or equal-to relationship test
fp_normalize(fr1)	Normalizes an unnormalized fp value. This function flushes to zero any unnormal values which can not be represented in the register file

Table B-1. Pseudo-Code Functions (Cont'd)

Function	Operation
fp_raise_fault(tmp_fp_env)	Checks the local instruction state for any faulting conditions which require an interruption to be raised.
fp_raise_traps(tmp_fp_env)	Checks the local instruction state for any trapping conditions which require an interruption to be raised.
fp_reg_bank_conflict(f1, f2)	Returns true if the two specified FRs are in the same bank.
fp_reg_disabled(f1, f2, f3, f4)	Check for possible disabled floating-point register faults.
fp_reg_read(freg)	Reads the FR and gives canonical double-extended denormals (and pseudo-denormals) their true mathematical exponent. Other classes of operands are unaltered.
fp_unordered(fr1, fr2)	IEEE standard unordered relationship
fp_fr_to_mem_format(freg, size)	Converts a floating-point value in register format to floating-point memory format. It assumes that the floating-point value in the register has been previously rounded to the correct precision which corresponds with the `size` parameter.
frcpa_exception_fault_check(fr2, fr3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `frcpa` instruction. It propagates NaNs, NaTVals, and special IEEE results.
frsqrta_exception_fault_check(fr3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `frsqrta` instruction. It propagates NaNs, NaTVals, and special IEEE results
ignored_field_mask(regclass, reg, value)	Boolean function that returns value with bits cleared to 0 corresponding to ignored bits for the specified register and register type.
instruction_serialize()	Ensures all prior register updates with side-effects are observed before subsequent instruction and data memory references are performed. Also ensures prior SYNC.i operations have been observed by the instruction cache.
instruction_synchronize	Synchronizes the instruction and data stream for Flush Cache operations. This function ensures that when prior FC operations are observed by the local data cache they are observed by the local instruction cache, and when prior FC operations are observed by another processor's data cache they are observed within the same processor's instruction cache.
is_finite(freg)	Returns true when floating register contains a finite number.
is_ignored_reg(regnum)	Boolean function that returns true if `regnum` is an ignored application register, otherwise false.
is_inf(freg)	Returns true when floating register contains an infinite number.
is_kernel_reg(ar_addr)	Returns a one if `ar_addr` is the address of a kernel register application 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.
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_implicit_prefetch(vaddr, hint)	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.
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.

Table B-1. Pseudo-Code Functions (Cont'd)

Function	Operation
mem_read(paddr, size, border, mattr, otype, hint)	Returns the `size` bytes starting at the physical memory location specified by `paddr` with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or ACQUIRE.
fp_mem_to_fr_format(mem, size)	Converts a floating-point value in memory format to floating-point register format.
mem_write(value, paddr, size, border, mattr, otype, hint)	Writes the least significant `size` bytes of `value` into memory starting at the physical memory address specified by `paddr` with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or RELEASE. No value is returned.
mem_xchg(data, paddr, size, byte_order, mattr, otype, hint)	Returns size bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. After the read, the least significant `size` bytes of data are written to `size` bytes in memory starting at the physical address specified by `paddr`. The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and must be ACQUIRE.
mem_xchg_add(add_val, paddr, size, byte_order, mattr, otype, hint)	Returns `size` bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. The least significant `size` bytes of the sum of the value read from memory and `add_val` is then written to `size` bytes in memory starting at the physical address specified by `paddr`. The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
mem_xchg_cond(cmp_val, data, paddr, size, byte_order, mattr, otype, hint)	Returns `size` bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. If the value read from memory is equal to `cmp_val,` then the least significant `size` bytes of data are written to `size bytes` in memory starting at the physical address specified by `paddr`. If the write is performed, the read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
ordering_fence()	Ensures prior data memory references are made visible before future data memory references are made visible by the processor.
pr_phys_to_virt(phys_id)	Returns the virtual register id of the predicate from the physical register id, `phys_id` of the predicate.
rotate_regs()	Decrements the Register Rename Base registers, effectively rotating the register files. CFM.rrb.gr is decremented only if CFM.sor is non-zero.

Table B-1. Pseudo-Code Functions (Cont'd)

Function	Operation
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_invalidate_non_current_regs()	All registers outside the current frame are invalidated.
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 rse_store for a list.
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_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. The privilege level of the store is obtained from AR[RSC].pl. The byte order of the store is obtained from AR[RSC].be. For mandatory RSE stores, type is MANDATORY. RSE stores do not invalidate ALAT entries.
rse_update_internal_stack_pointers(new_store_pointer)	Given a new value for AR[BSPSTORE] (new_store_pointer) this function computes the new value for AR[BSP]. This value is equal to new_store_pointer plus the number of dirty registers plus the number of intervening NaT collections. This means that the size of the dirty partition is the same before and after a write to AR[BSPSTORE]. All clean registers are moved to the invalid partition.
sign_ext(value, pos)	Returns a 64 bit number with bits pos-1 through 0 taken from value and bit pos-1 of value replicated in bit positions pos through 63. If pos is greater than or equal to 64, value is returned.
tlb_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_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). If a fault is generated, tlb_translate_nonaccess does not return.
unimplemented_physical_address(paddr)	Return TRUE if the presented physical address is unimplemented on this processor model; FALSE otherwise. This function is model-specific.
impl_undefined_natd_gr_read(paddr, size, be, mattr, otype, ldhint)	defines register return data for a speculative load to a NaTed address. This function may return data from another address space.
unimplemented_virtual_address(vaddr)	Return TRUE if the presented virtual address is unimplemented on this processor model; FALSE otherwise. This function is model-specific.

Table B-1. Pseudo-Code Functions (Cont'd)

Function	Operation
fp_update_fpsr(sf, tmp_fp_env)	Copies a floating-point instruction's local state into the global FPSR.
zero_ext(value, pos)	Returns a 64 bit unsigned number with bits `pos`-1 through 0 taken from `value` and zeroes in bit positions `pos` through 63. If `pos` is greater than or equal to 64, `value` is returned.

IA-64 Instruction Formats C

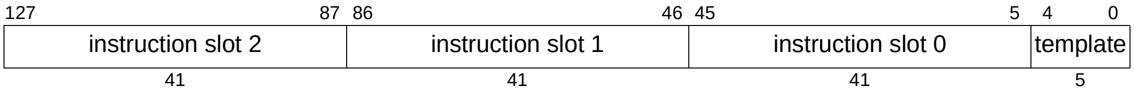
Each IA-64 instruction is categorized into one of six types; each instruction type may be executed on one or more execution unit types. Table C-1 lists the instruction types and the execution unit type on which they are executed:

Table C-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

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 C-1.

Figure C-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 C-2 indicates the defined combinations. The three rightmost columns correspond to the three instruction slots in a bundle; listed within each column is the execution unit type controlled by that instruction slot for each encoding of the template field. A double line to the right of an instruction slot indicates that a stop occurs at that point within the current bundle. See "Instruction Encoding Overview" on page 3-14 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 C-2) are reserved and cause an Illegal Operation fault.

Extended instructions, used for long immediate integer instructions, occupy two instruction slots.

Table C-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
05	M-unit	L-unit	X-unit ‖
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit ‖

Table C-2. Template Field Encoding and Instruction Slot Mapping (Cont'd)

Template	Slot 0	Slot 1	Slot 2
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			

C.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 C-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 C-3) behave in one of three ways:

- Ignored major ops (white entries in Table C-3) execute as nop instructions.
- Reserved major ops (light gray in the gray scale version of Table C-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 C-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.

Table C-3. Major Opcode Assignments

Major Op (bits 40:37)	Instruction Type				
	I/A	M/A	F	B	L+X
0	Misc 0	Mem Mgmt 0	FP Misc 0	Misc/Indirect Branch 0	Misc 0
1	1	Mem Mgmt 1	FP Misc 1	Indirect Call 1	1
2	2	2	2	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	7	7
8	ALU/MM ALU 8	ALU/MM ALU 8	fma 8	8	8
9	Add Imm$_{22}$ 9	Add Imm$_{22}$ 9	fma 9	9	9
A	A	A	fms A	A	A
B	B	B	fms B	B	B
C	Compare C	Compare C	fnma C	C	C
D	Compare D	Compare D	fnma D	D	D
E	Compare E	Compare E	fselect/xma E	E	E
F	F	F	F	F	F

Table C-4 on page C-4 summarizes all the instruction formats. The instruction fields are color-coded for ease of identification, as described in Table C-5 on page C-6.

The instruction field names, used throughout this chapter, are described in Table C-6 on page C-7. The set of special notations (such as whether an instruction must be first in an instruction group) are listed in Table C-7 on page C-7. 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 C-65 on page C-77 shows how the immediates are formed from the instruction fields for each instruction which has an immediate.

Table C-4. Instruction Format Summary

		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
ALU	A1	8					x_{2a}		v_e	x_4				x_{2b}		r_3							r_2							r_1							qp					
Shift L and Add	A2	8					x_{2a}		v_e	x_4				ct_{2d}		r_3							r_2							r_1							qp					
ALU Imm$_8$	A3	8				s	x_{2a}		v_e	x_4				x_{2b}		r_3							imm$_{7b}$							r_1							qp					
Add Imm$_{14}$	A4	8				s	x_{2a}		v_e	imm$_{6d}$						r_3							imm$_{7b}$							r_1							qp					
Add Imm$_{22}$	A5	9				s	imm$_{9d}$									imm$_{5c}$					r_3		imm$_{7b}$							r_1							qp					
Compare	A6	C - E				t_b	x_2		t_a	p_2						r_3							r_2							c	p_1						qp					
Compare to Zero	A7	C - E				t_b	x_2		t_a	p_2						r_3							0							c	p_1						qp					
Compare Imm$_8$	A8	C - E				s	x_2		t_a	p_2						r_3							imm$_{7b}$							c	p_1						qp					
MM ALU	A9	8				z_a	x_{2a}		z_b	x_4				x_{2b}		r_3							r_2							r_1							qp					
MM Shift and Add	A10	8				z_a	x_{2a}		z_b	x_4				ct_{2d}		r_3							r_2							r_1							qp					
MM Multiply Shift	I1	7				z_a	x_{2a}		z_b	v_e	ct_{2d}		x_{2b}			r_3							r_2							r_1							qp					
MM Mpy/Mix/Pack	I2	7				z_a	x_{2a}		z_b	v_e	x_{2c}		x_{2b}			r_3							r_2							r_1							qp					
MM Mux1	I3	7				z_a	x_{2a}		z_b	v_e	x_{2c}		x_{2b}						mbt$_{4c}$				r_2							r_1							qp					
MM Mux2	I4	7				z_a	x_{2a}		z_b	v_e	x_{2c}		x_{2b}		mht$_{8c}$								r_2							r_1							qp					
Shift R Variable	I5	7				z_a	x_{2a}		z_b	v_e	x_{2c}		x_{2b}			r_3							r_2							r_1							qp					
MM Shift R Fixed	I6	7				z_a	x_{2a}		z_b	v_e	x_{2c}		x_{2b}			r_3								count$_{5b}$						r_1							qp					
Shift L Variable	I7	7				z_a	x_{2a}		z_b	v_e	x_{2c}		x_{2b}			r_3							r_2							r_1							qp					
MM Shift L Fixed	I8	7				z_a	x_{2a}		z_b	v_e	x_{2c}		x_{2b}					ccount$_{5c}$					r_2							r_1							qp					
Popcount	I9	7				z_a	x_{2a}		z_b	v_e	x_{2c}		x_{2b}			r_3							0							r_1							qp					
Shift Right Pair	I10	5					x_2		x	count$_{6d}$						r_3							r_2							r_1							qp					
Extract	I11	5					x_2		x	len$_{6d}$						r_3							pos$_{6b}$						y	r_1							qp					
Dep.Z	I12	5					x_2		x	len$_{6d}$						y	cpos$_{6c}$						r_2							r_1							qp					
Dep.Z Imm$_8$	I13	5				s	x_2		x	len$_{6d}$						y	cpos$_{6c}$						imm$_{7b}$							r_1							qp					
Deposit Imm$_1$	I14	5				s	x_2		x	len$_{6d}$						r_3							cpos$_{6b}$							r_1							qp					
Deposit	I15	4				cpos$_{6d}$						len$_{4d}$				r_3							r_2							r_1							qp					
Test Bit	I16	5				t_b	x_2		t_a	p_2						r_3							pos$_{6b}$						y	c	p_1						qp					
Test NaT	I17	5				t_b	x_2		t_a	p_2						r_3													y	c	p_1						qp					
Break/Nop	I19	0				i	x_3			x_6							imm$_{20a}$																				qp					
Int Spec Check	I20	0				s	x_3			imm$_{13c}$													r_2							imm$_{7a}$							qp					
Move to BR	I21	0					x_3													x			r_2											b_1			qp					
Move from BR	I22	0					x_3			x_6																	b_2			r_1							qp					
Move to Pred	I23	0				s	x_3				mask$_{8c}$												r_2							mask$_{7a}$							qp					
Move to Pred Imm$_{44}$	I24	0				s	x_3			imm$_{27a}$																											qp					
Move from Pred/IP	I25	0					x_3			x_6																				r_1							qp					
Move to AR	I26	0					x_3			x_6						ar_3							r_2														qp					
Move to AR Imm$_8$	I27	0				s	x_3			x_6						ar_3							imm$_{7b}$														qp					
Move from AR	I28	0					x_3			x_6						ar_3														r_1							qp					
Sxt/Zxt/Czx	I29	0					x_3			x_6						r_3														r_1							qp					
		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 C-4. Instruction Format Summary (Cont'd)

		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 Load	M1	4	m	x_6			hint	x	r_3					r_1	qp
Int Load +Reg	M2	4	m	x_6			hint	x	r_3		r_2			r_1	qp
Int Load +Imm	M3	5	s	x_6			hint	i	r_3		imm_{7b}			r_1	qp
Int Store	M4	4	m	x_6			hint	x	r_3		r_2				qp
Int Store +Imm	M5	5	s	x_6			hint	i	r_3		r_2			imm_{7a}	qp
FP Load	M6	6	m	x_6			hint	x	r_3					f_1	qp
FP Load +Reg	M7	6	m	x_6			hint	x	r_3		r_2			f_1	qp
FP Load +Imm	M8	7	s	x_6			hint	i	r_3		imm_{7b}			f_1	qp
FP Store	M9	6	m	x_6			hint	x	r_3		f_2				qp
FP Store +Imm	M10	7	s	x_6			hint	i	r_3		f_2			imm_{7a}	qp
FP Load Pair	M11	6	m	x_6			hint	x	r_3		f_2			f_1	qp
FP Load Pair +Imm	M12	6	m	x_6			hint	x	r_3		f_2			f_1	qp
Line Prefetch	M13	6	m	x_6			hint	x	r_3						qp
Line Prefetch +Reg	M14	6	m	x_6			hint	x	r_3		r_2				qp
Line Prefetch +Imm	M15	7	s	x_6			hint	i	r_3		imm_{7b}				qp
(Cmp &) Exchg	M16	4	m	x_6			hint	x	r_3		r_2			r_1	qp
Fetch & Add	M17	4	m	x_6			hint	x	r_3			s	i_{2b}	r_1	qp
Set FR	M18	6	m	x_6				x			r_2			f_1	qp
Get FR	M19	4	m	x_6				x			f_2			r_1	qp
Int Spec Check	M20	1	s	x_3	imm_{13c}						r_2			imm_{7a}	qp
FP Spec Check	M21	1	s	x_3	imm_{13c}						f_2			imm_{7a}	qp
Int ALAT Check	M22	0	s	x_3	imm_{20b}									r_1	qp
FP ALAT Check	M23	0	s	x_3	imm_{20b}									f_1	qp
Sync/Srlz/ALAT	M24	0		x_3	x_2	x_4									qp
RSE Control	M25	0		x_3	x_2	x_4									0
Int ALAT Inval	M26	0		x_3	x_2	x_4								r_1	qp
FP ALAT Inval	M27	0		x_3	x_2	x_4								f_1	qp
Flush Cache	M28	1		x_3	x_6				r_3						qp
Move to AR	M29	1		x_3	x_6				ar_3		r_2				qp
Move to AR Imm$_8$	M30	0	s	x_3	x_2	x_4			ar_3		imm_{7b}				qp
Move from AR	M31	1		x_3	x_6				ar_3					r_1	qp
Alloc	M34	1		x_3		sor			sol		sof			r_1	0
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/Nop	M37	0	i	x_3	x_2	x_4				imm_{20a}					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
		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 C-4. Instruction Format Summary (Cont’d)

<table>
<tr><th></th><th></th><th>40</th><th>39</th><th>38</th><th>37</th><th>36</th><th>35</th><th>34</th><th>33</th><th>32</th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>IP-Relative Branch</td><td>B1</td><td colspan="4">4</td><td>s</td><td>d</td><td colspan="2">wh</td><td colspan="20">imm_{20b}</td><td>p</td><td colspan="3"></td><td colspan="3">btype</td><td colspan="6">qp</td><td></td></tr>
<tr><td>Counted Branch</td><td>B2</td><td colspan="4">4</td><td>s</td><td>d</td><td colspan="2">wh</td><td colspan="20">imm_{20b}</td><td>p</td><td colspan="3"></td><td colspan="3">btype</td><td colspan="6">0</td><td></td></tr>
<tr><td>IP-Relative Call</td><td>B3</td><td colspan="4">5</td><td>s</td><td>d</td><td colspan="2">wh</td><td colspan="20">imm_{20b}</td><td>p</td><td colspan="3"></td><td colspan="3">b_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>Indirect Branch</td><td>B4</td><td colspan="4">0</td><td></td><td>d</td><td colspan="2">wh</td><td colspan="6">x_6</td><td colspan="11"></td><td colspan="3">b_2</td><td>p</td><td colspan="3"></td><td colspan="3">btype</td><td colspan="6">qp</td><td></td></tr>
<tr><td>Indirect Call</td><td>B5</td><td colspan="4">1</td><td></td><td>d</td><td colspan="3">wh</td><td colspan="16"></td><td colspan="3">b_2</td><td>p</td><td colspan="3"></td><td colspan="3">b_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>Misc</td><td>B8</td><td colspan="4">0</td><td colspan="4"></td><td colspan="6">x_6</td><td colspan="21"></td><td colspan="6">0</td><td></td></tr>
<tr><td>Break/Nop</td><td>B9</td><td colspan="4">0/2</td><td>i</td><td colspan="3"></td><td colspan="6">x_6</td><td></td><td colspan="20">imm_{20a}</td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Arithmetic</td><td>F1</td><td colspan="4">8 - D</td><td>x</td><td colspan="2">sf</td><td colspan="7">f_4</td><td colspan="7">f_3</td><td colspan="7">f_2</td><td colspan="7">f_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>Fixed Multiply Add</td><td>F2</td><td colspan="4">E</td><td>x</td><td colspan="2">x_2</td><td colspan="7">f_4</td><td colspan="7">f_3</td><td colspan="7">f_2</td><td colspan="7">f_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Select</td><td>F3</td><td colspan="4">E</td><td>x</td><td colspan="2"></td><td colspan="7">f_4</td><td colspan="7">f_3</td><td colspan="7">f_2</td><td colspan="7">f_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Compare</td><td>F4</td><td colspan="4">4</td><td>r_b</td><td colspan="2">sf</td><td>r_a</td><td colspan="6">p_2</td><td colspan="7">f_3</td><td colspan="7">f_2</td><td>t_a</td><td colspan="6">p_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Class</td><td>F5</td><td colspan="4">5</td><td colspan="2"></td><td colspan="2">fc_2</td><td colspan="6">p_2</td><td colspan="7">$fclass_{7c}$</td><td colspan="7">f_2</td><td>t_a</td><td colspan="6">p_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Recip Approx</td><td>F6</td><td colspan="4">0 - 1</td><td>q</td><td colspan="2">sf</td><td>x</td><td colspan="6">p_2</td><td colspan="7">f_3</td><td colspan="7">f_2</td><td colspan="7">f_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Recip Sqrt App</td><td>F7</td><td colspan="4">0 - 1</td><td>q</td><td colspan="2">sf</td><td>x</td><td colspan="6">p_2</td><td colspan="7">f_3</td><td colspan="7"></td><td colspan="7">f_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Min/Max/Pcmp</td><td>F8</td><td colspan="4">0 - 1</td><td></td><td colspan="2">sf</td><td>x</td><td colspan="6">x_6</td><td colspan="7">f_3</td><td colspan="7">f_2</td><td colspan="7">f_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Merge/Logical</td><td>F9</td><td colspan="4">0 - 1</td><td colspan="3"></td><td>x</td><td colspan="6">x_6</td><td colspan="7">f_3</td><td colspan="7">f_2</td><td colspan="7">f_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>Convert FP to Fixed</td><td>F10</td><td colspan="4">0 - 1</td><td></td><td colspan="2">sf</td><td>x</td><td colspan="6">x_6</td><td colspan="7"></td><td colspan="7">f_2</td><td colspan="7">f_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>Convert Fixed to FP</td><td>F11</td><td colspan="4">0</td><td colspan="3"></td><td>x</td><td colspan="6">x_6</td><td colspan="7"></td><td colspan="7">f_2</td><td colspan="7">f_1</td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Set Controls</td><td>F12</td><td colspan="4">0</td><td></td><td colspan="2">sf</td><td>x</td><td colspan="6">x_6</td><td colspan="7">$omask_{7c}$</td><td colspan="7">$amask_{7b}$</td><td colspan="7"></td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Clear Flags</td><td>F13</td><td colspan="4">0</td><td></td><td colspan="2">sf</td><td>x</td><td colspan="6">x_6</td><td colspan="21"></td><td colspan="6">qp</td><td></td></tr>
<tr><td>FP Check Flags</td><td>F14</td><td colspan="4">0</td><td>s</td><td colspan="2">sf</td><td>x</td><td colspan="6">x_6</td><td></td><td colspan="20">imm_{20a}</td><td colspan="6">qp</td><td></td></tr>
<tr><td>Break/Nop</td><td>F15</td><td colspan="4">0</td><td>i</td><td colspan="2"></td><td>x</td><td colspan="6">x_6</td><td></td><td colspan="20">imm_{20a}</td><td colspan="6">qp</td><td></td></tr>
<tr><td>Break/Nop</td><td>X1</td><td colspan="4">0</td><td>i</td><td colspan="3">x_3</td><td colspan="6">x_6</td><td></td><td colspan="20">imm_{20a}</td><td colspan="6">qp</td><td>imm_{41}</td></tr>
<tr><td>Move Imm_{64}</td><td>X2</td><td colspan="4">6</td><td>i</td><td colspan="9">imm_{9d}</td><td colspan="5">imm_{5c}</td><td>i_c</td><td>v_c</td><td colspan="7">imm_{7b}</td><td colspan="7">r_1</td><td colspan="6">qp</td><td>imm_{41}</td></tr>
<tr><td></td><td></td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
</table>

Table C-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	Reserved Instruction
Qualifying Predicate	Reserved Inst if PR[qp] is 1
Ignored Field/Instruction	

Table C-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
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*
len_{4d}, len_{6d}	extract/deposit length immediate
m	memory reference post-modify opcode extension
$mask_x$	predicate immediate mask
mbt_{4c}, mht_{8c}	multimedia mux1/mux2 immediate
p	sequential prefetch hint opcode extension
p_1, p_2	predicate register target
pos_{6b}	test bit/extract bit position immediate
q	floating-point reciprocal/reciprocal square-root opcode extension
qp	qualifying predicate register source
r_n	general register source/target
s	immediate sign bit
sf	floating-point status field opcode extension
sof, sol, sor	alloc size of frame, size of locals, size of rotating immediates
t_a, t_b	compare type opcode extension
v_x	reserved opcode extension field
wh	branch whether hint opcode extension
x, x_n	opcode extension of length 1 or *n*
y	extract/deposit/test bit/test NaT opcode extension
z_a, z_b	multimedia operand size opcode extension

Table C-7. Special Instruction Notations

Notation	Description
f	instruction must be the first in an instruction group
l	instruction must be the last in an instruction group
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 C-18, "M-Unit Instruction Encodings" on page C-32, "B-Unit Instruction Encodings" on page C-58, "F-Unit Instruction Encodings" on page C-64, and "X-Unit Instruction Encodings" on page C-75.

Within each section, the instructions are grouped by function, and appear with their instruction format in the same order as in Table C-4 “Instruction Format Summary” on page C-4. The opcode extension fields are briefly described and tables present the opcode extension assignments. Unused instruction encodings (appearing as blank entries in the opcode extensions tables) behave in one of three ways:

- Ignored instructions (white entries in the tables) execute as nop instructions.
- Reserved instructions (light gray in the gray scale version of the tables, brown 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.

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.

C.2 A-Unit Instruction Encodings

C.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 C-8 shows the 2-bit x_{2a} and 1-bit v_e assignments, Table C-9 shows the integer ALU 4-bit+2-bit assignments, and Table C-12 on page C-15 shows the multimedia ALU 1-bit+2-bit assignments (which also share major opcode 8).

Table C-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 C-9)	
	1	Multimedia ALU 1-bit+2-bit Ext (Table C-12)	
	2	adds – imm_{14} A4	
	3	addp4 – imm_{14} A4	

Table C-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				

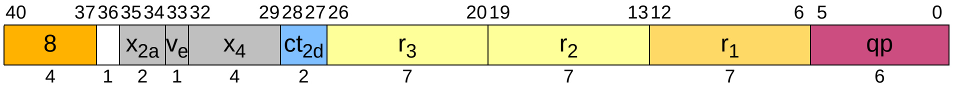
C.2.1.1. Integer ALU – Register-Register

	40 37	36	35 34	33	32 29	28 27	26 20	19 13	12 6	5 0
A1	8		x_{2a}	v_e	x_4	x_{2b}	r_3	r_2	r_1	qp
	4	1	2	1	4	2	7	7	7	6

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

C.2.1.2. Shift Left and Add

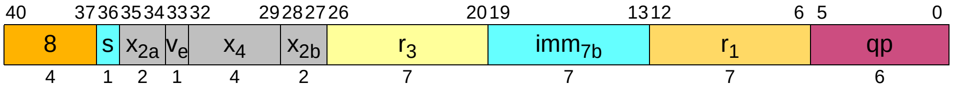
A2



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

C.2.1.3. Integer ALU – Immediate$_8$-Register

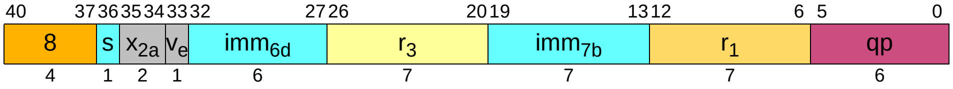
A3



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

C.2.1.4. Add Immediate$_{14}$

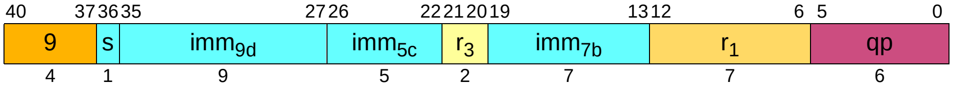
A4



Instruction	Operands	Opcode	Extension	
			x_{2a}	v_e
adds	$r_1 = imm_{14}, r_3$	8	2	0
addp4			3	

C.2.1.5. Add Immediate$_{22}$

A5



Instruction	Operands	Opcode
addl	$r_1 = imm_{22}, r_3$	9

C.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 C-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 C-11.

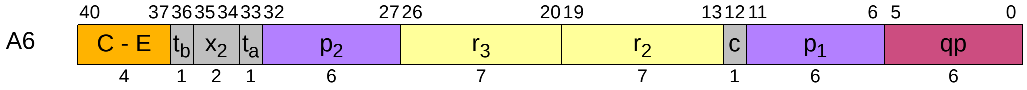
Table C-10. Integer Compare Opcode Extensions

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 C-11. Integer Compare Immediate Opcode Extensions

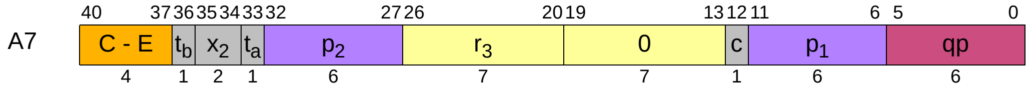
x_2 Bits 35:34	t_a Bit 33	c Bit 12	Opcode Bits 40:37		
			C	D	E
2	0	0	cmp.lt – imm_8 A8	cmp.ltu – imm_8 A8	cmp.eq – imm_8 A8
		1	cmp.lt.unc – imm_8 A8	cmp.ltu.unc – imm_8 A8	cmp.eq.unc – imm_8 A8
	1	0	cmp.eq.and – imm_8 A8	cmp.eq.or – imm_8 A8	cmp.eq.or.andcm – imm_8 A8
		1	cmp.ne.and – imm_8 A8	cmp.ne.or – 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

C.2.2.1. Integer Compare – Register-Register



Instruction	Operands	Opcode	Extension			
			x_2	t_b	t_a	c
cmp.lt	$p_1, p_2 = r_2, r_3$	C	0	0	0	0
cmp.ltu		D				
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				

C.2.2.2. Integer Compare to Zero – Register



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				

C.2.2.3. Integer Compare – Immediate-Register

	40 37	36	35 34	33	32 27	26 20	19 13	12	11 6	5 0
A8	C - E	s	x_2	t_a	p_2	r_3	imm_{7b}	c	p_1	qp
	4	1	2	1	6	7	7	1	6	6

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			

C.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 C-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 C-13 on page C-15.

Table C-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 C-13)
			1	Multimedia ALU Size 2 (Table C-14)
		1	0	Multimedia ALU Size 4 (Table C-15)
			1	

Table C-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 C-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 C-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				

C.2.3.1. Multimedia ALU

	40 37	36	35 34	33	32 29	28 27	26 20	19 13	12 6	5 0
A9	8	z_a	x_{2a}	z_b	x_4	x_{2b}	r_3	r_2	r_1	qp
	4	1	2	1	4	2	7	7	7	6

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		

C.2.3.2. Multimedia Shift and Add

	40 37	36	35 34	33	32 29	28 27	26 20	19 13	12 6	5 0
A10	8	z_a	x_{2a}	z_b	x_4	ct_{2d}	r_3	r_2	r_1	qp
	4	1	2	1	4	2	7	7	7	6

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

C.3 I-Unit Instruction Encodings

C.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 C-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 C-17.

Table C-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 C-17)	
		1	Multimedia Size 2 (Table C-18)	
	1	0	Multimedia Size 4 (Table C-19)	
		1	Variable Shift (Table C-20)	

Table C-17. Multimedia Max/Min/Mix/Pack/Unpack 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 C-18. Multimedia Multiply/Shift/Max/Min/Mix/Pack/Unpack Size 2 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	0	1	0	0	0	pshr2.u – var I5	pshl2 – var I7		
					1	pmpyshr2.u I1			
					2	pshr2 – var I5			
					3	pmpyshr2 I1			
				1	0				
					1	pshr2.u – fixed I6		popcnt I9	
					2				
					3	pshr2 – fixed I6			
				2	0	pack2.uss I2	unpack2.h I2	mix2.r I2	
					1				pmpy2.r I2
					2	pack2.sss I2	unpack2.l I2	mix2.l I2	
					3	pmin2 I2	pmax2 I2		pmpy2.l I2
				3	0				
					1		pshl2 – fixed I8		
					2			mux2 I4	
					3				

Table C-19. Multimedia Shift/Mix/Pack/Unpack Size 4 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	1	0	0	0	0	pshr4.u – var I5	pshl4 – var I7		
					1				
					2	pshr4 – var I5			
					3				
				1	0				
					1	pshr4.u – fixed I6			
					2				
					3	pshr4 – fixed I6			
				2	0		unpack4.h I2	mix4.r I2	
					1				
					2	pack4.sss I2	unpack4.l I2	mix4.l I2	
					3				
				3	0				
					1		pshl4 – fixed I8		
					2				
					3				

Table C-20. Variable Shift 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				

C.3.1.1. Multimedia Multiply and Shift

	40 37	36	35 34	33	32	31 30	29 28	27	26 20	19 13	12 6	5 0
I1	7	z_a	x_{2a}	z_b	v_e	ct_{2d}	x_{2b}		r_3	r_2	r_1	qp
	4	1	2	1	1	2	2	1	7	7	7	6

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

C.3.1.2. Multimedia Multiply/Mix/Pack/Unpack

	40 37	36	35 34	33	32	31 30	29 28	27	26 20	19 13	12 6	5 0
I2	7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		r_3	r_2	r_1	qp
	4	1	2	1	1	2	2	1	7	7	7	6

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

C.3.1.3. Multimedia Mux1

	40 37	36	35 34	33	32	31 30	29 28	27 24	23 20	19 13	12 6	5 0
I3	7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		mbt_{4c}	r_2	r_1	qp
	4	1	2	1	1	2	2	4	4	7	7	6

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

C.3.1.4. Multimedia Mux2

	40 37	36	35 34	33	32	31 30	29 28	27 20	19 13	12 6	5 0
I4	7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}	mht_{8c}	r_2	r_1	qp
	4	1	2	1	1	2	2	8	7	7	6

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

C.3.1.5. Shift Right – Variable

	40 37	36	35 34	33	32	31 30	29 28	27	26 20	19 13	12 6	5 0
I5	7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		r_3	r_2	r_1	qp
	4	1	2	1	1	2	2	1	7	7	7	6

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				

C.3.1.6. Multimedia Shift Right – Fixed

	40 37	36	35 34	33	32	31 30	29 28	27	26 20	19	18 14	13	12 6	5 0
I6	7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		r_3		$count_{5b}$		r_1	qp
	4	1	2	1	1	2	2	1	7	1	5	1	7	6

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				

C.3.1.7. Shift Left – Variable

	40 37	36	35 34	33	32	31 30	29 28	27	26 20	19 13	12 6	5 0
I7	7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		r_3	r_2	r_1	qp
	4	1	2	1	1	2	2	1	7	7	7	6

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				

C.3.1.8. Multimedia Shift Left – Fixed

	40 37	36	35 34	33	32	31 30	29 28	27 25	24 20	19 13	12 6	5 0
I8	7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		$ccount_{5c}$	r_2	r_1	qp
	4	1	2	1	1	2	2	3	5	7	7	6

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				

C.3.1.9. Population Count

	40 37	36	35 34	33	32	31 30	29 28	27	26 20	19 13	12 6	5 0
I9	7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		r_3	0	r_1	qp
	4	1	2	1	1	2	2	1	7	7	7	6

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

C.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 C-21 shows the test bit, extract, and shift right pair assignments.

Table C-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 C-23)	Test NaT (Table C-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 C-22 shows these assignments.

Table C-22. Deposit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	x Bit 33	y Bit 26	
			0	1
5	0	1	Test Bit/Test NaT (Table C-23)	
	1		dep.z I12	dep.z – imm_8 I13
	2			
	3		dep – imm_1 I14	

C.3.2.1. Shift Right Pair

I10

40 37	36	35 34	33	32 27	26 20	19 13	12 6	5 0
5		x_2	x	$count_{6d}$	r_3	r_2	r_1	qp
4	1	2	1	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_2	x
shrp	$r_1 = r_2, r_3, count_6$	5	3	0

C.3.2.2. Extract

I11

40 37	36	35 34	33	32 27	26 20	19 14	13	12 6	5 0
5		x_2	x	len_{6d}	r_3	pos_{6b}	y	r_1	qp
4	1	2	1	6	7	6	1	7	6

Instruction	Operands	Opcode	Extension		
			x_2	x	y
extr.u	$r_1 = r_3, pos_6, len_6$	5	1	0	0
extr					1

C.3.2.3. Zero and Deposit

	40 37	36	35 34	33	32 27	26	25 20	19 13	12 6	5 0
I12	5		x_2	x	len_{6d}	y	$cpos_{6c}$	r_2	r_1	qp
	4	1	2	1	6	1	6	7	7	6

Instruction	Operands	Opcode	Extension		
			x_2	x	y
dep.z	$r_1 = r_2, pos_6, len_6$	5	1	1	0

C.3.2.4. Zero and Deposit Immediate$_8$

	40 37	36	35 34	33	32 27	26	25 20	19 13	12 6	5 0
I13	5	s	x_2	x	len_{6d}	y	$cpos_{6c}$	imm_{7b}	r_1	qp
	4	1	2	1	6	1	6	7	7	6

Instruction	Operands	Opcode	Extension		
			x_2	x	y
dep.z	$r_1 = imm_8, pos_6, len_6$	5	1	1	1

C.3.2.5. Deposit Immediate$_1$

	40 37	36	35 34	33	32 27	26 20	19 14	13	12 6	5 0
I14	5	s	x_2	x	len_{6d}	r_3	$cpos_{6b}$		r_1	qp
	4	1	2	1	6	7	6	1	7	6

Instruction	Operands	Opcode	Extension	
			x_2	x
dep	$r_1 = imm_1, r_3, pos_6, len_6$	5	3	1

C.3.2.6. Deposit

	40 37	36 31	30 27	26 20	19 13	12 6	5 0
I15	4	$cpos_{6d}$	len_{4d}	r_3	r_2	r_1	qp
	4	6	4	7	7	7	6

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

C.3.3 Test Bit

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

Table C-23. Test Bit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	t_a Bit 33	t_b Bit 36	c Bit 12	y Bit 13	
					0	1
5	0	0	0	0	tbit.z I16	tnat.z I17
				1	tbit.z.unc I16	tnat.z.unc I17
			1	0	tbit.z.and I16	tnat.z.and I17
				1	tbit.nz.and I16	tnat.nz.and I17
		1	0	0	tbit.z.or I16	tnat.z.or I17
				1	tbit.nz.or I16	tnat.nz.or I17
			1	0	tbit.z.or.andcm I16	tnat.z.or.andcm I17
				1	tbit.nz.or.andcm I16	tnat.nz.or.andcm I17

C.3.3.1. Test Bit

	40 37	36	35 34	33	32 27	26 20	19 14	13	12	11 6	5 0
I16	5	t_b	x_2	t_a	p_2	r_3	pos_{6b}	y	c	p_1	qp
	4	1	2	1	6	7	6	1	1	6	6

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

C.3.3.2. Test NaT

	40 37	36	35 34	33	32 27	26 20	19 14	13	12	11 6	5 0
I17	5	t_b	x_2	t_a	p_2	r_3		y	c	p_1	qp
	4	1	2	1	6	7	6	1	1	6	6

Instruction	Operands	Opcode	Extension				
			x_2	t_a	t_b	y	c
tnat.z	$p_1, p_2 = r_3$	5	0	0	0	1	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

C.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 C-24 shows the 3-bit assignments and Table C-25 summarizes the 6-bit assignments.

Table C-24. Misc I-Unit 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	6-bit Ext (Table C-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 I22

Table C-25. Misc I-Unit 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
0	0	0	break.i I19	zxt1 I29		mov from ip I25
		1	nop.i I19	zxt2 I29		mov from b I22
		2		zxt4 I29		mov.i from ar I28
		3				mov from pr I25
		4		sxt1 I29		
		5		sxt2 I29		
		6		sxt4 I29		
		7				
		8		czx1.l I29		
		9		czx2.l I29		
		A	mov.i to ar – imm$_8$ I27		mov.i to ar I26	
		B				
		C		czx1.r I29		
		D		czx2.r I29		
		E				
		F				

C.3.4.1. Break/Nop (I-Unit)

I19

40 37	36	35 33	32 27	26	25 6	5 0
0	i	x_3	x_6		imm$_{20a}$	qp
4	1	3	6	1	20	6

Instruction	Operands	Opcode	Extension	
			x_3	x_6
break.i	*imm$_{21}$*	0	0	00
nop.i				01

C.3.4.2. Integer Speculation Check (I-Unit)

I20

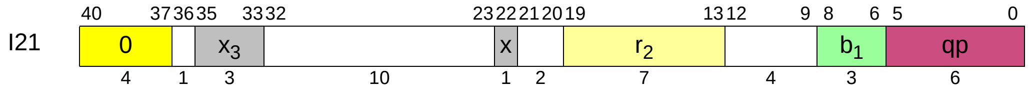
40 37	36	35 33	32 20	19 13	12 6	5 0
0	s	x_3	imm$_{13c}$	r_2	imm$_{7a}$	qp
4	1	3	13	7	7	6

Instruction	Operands	Opcode	Extension
			x_3
chk.s.i	r_2, *target$_{25}$*	0	1

C.3.5 GR/BR Moves

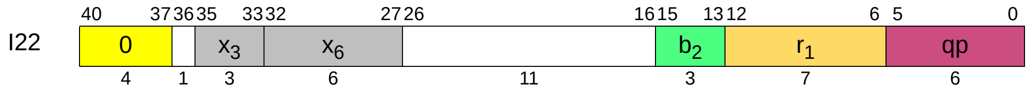
The GR/BR move instructions are encoded in major opcode 0. See "Miscellaneous I-Unit Instructions" on page C-27 for a summary of the opcode extensions. The mov to BR instruction uses a 1-bit opcode extension field (x) in bit 22 to distinguish the return form from the normal form.

C.3.5.1. Move to BR



Instruction	Operands	Opcode	Extension	
			x_3	x
mov	$b_1 = r_2$	0	7	0
mov.ret				1

C.3.5.2. Move from BR

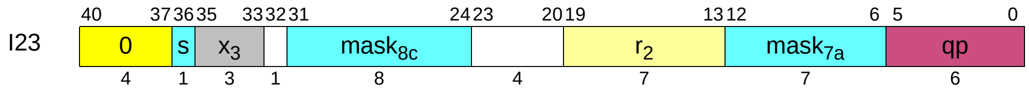


Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	$r_1 = b_2$	0	0	31

C.3.6 GR/Predicate/IP Moves

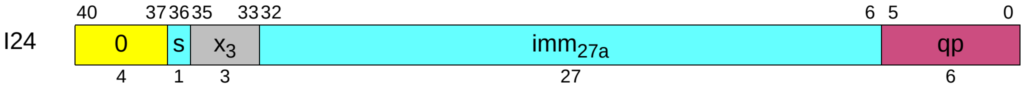
The GR/Predicate/IP move instructions are encoded in major opcode 0. See "Miscellaneous I-Unit Instructions" on page C-27 for a summary of the opcode extensions.

C.3.6.1. Move to Predicates – Register



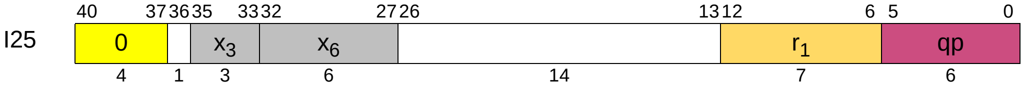
Instruction	Operands	Opcode	Extension
			x_3
mov	pr = r_2, $mask_{17}$	0	3

C.3.6.2. Move to Predicates – Immediate$_{44}$



Instruction	Operands	Opcode	Extension
			x_3
mov	pr.rot = imm_{44}	0	2

C.3.6.3. Move from Predicates/IP

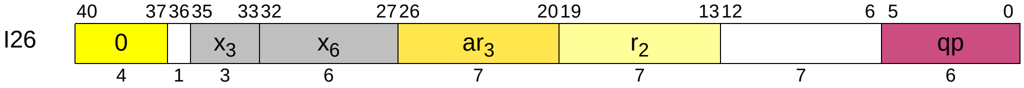


Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	r_1 = ip	0	0	30
	r_1 = pr			33

C.3.7 GR/AR Moves (I-Unit)

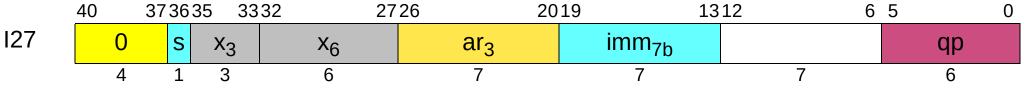
The I-Unit GR/AR move instructions are encoded in major opcode 0. (Some ARs are accessed using memory management instructions on the M-unit. See "GR/AR Moves (M-Unit)" on page C-53.) See "Miscellaneous I-Unit Instructions" on page C-27 for a summary of the I-Unit GR/AR opcode extensions.

C.3.7.1. Move to AR – Register (I-Unit)



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$ar_3 = r_2$	0	0	2A

C.3.7.2. Move to AR – Immediate$_8$ (I-Unit)



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$ar_3 = imm_8$	0	0	0A

C.3.7.3. Move from AR (I-Unit)

	40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
I28	0		x_3	x_6	ar_3		r_1	qp
	4	1	3	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$r_1 = ar_3$	0	0	32

C.3.8 Sign/Zero Extend/Compute Zero Index

	40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
I29	0		x_3	x_6	r_3		r_1	qp
	4	1	3	6	7	7	7	6

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

C.4 M-Unit Instruction Encodings

C.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 C-26. 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 C-27.

Table C-26. 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 C-28)
	0	1	Semaphore/get FR (Table C-31)
	1	0	Load +Reg (Table C-29)
	1	1	

Table C-27. 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 C-32)
	0	1	FP Load Pair/set FR (Table C-35)
	1	0	FP Load +Reg (Table C-33)
	1	1	FP Load Pair +Imm (Table C-36)

The integer load/store opcode extensions are summarized in Table C-28 on page C-33, Table C-29 on page C-33, and Table C-30 on page C-34, and the semaphore and get FR opcode extensions in Table C-31 on page C-34. The floating-point load/store opcode extensions are summarized in Table C-32 on page C-35, Table C-33 on page C-35, and Table C-34 on page C-36, the floating-point load pair and set FR opcode extensions in Table C-35 on page C-36 and Table C-36 on page C-37.

Table C-28. Integer Load/Store Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	0	0	ld1 M1	ld2 M1	ld4 M1	ld8 M1
			1	ld1.s M1	ld2.s M1	ld4.s M1	ld8.s M1
			2	ld1.a M1	ld2.a M1	ld4.a M1	ld8.a M1
			3	ld1.sa M1	ld2.sa M1	ld4.sa M1	ld8.sa M1
			4	ld1.bias M1	ld2.bias M1	ld4.bias M1	ld8.bias M1
			5	ld1.acq M1	ld2.acq M1	ld4.acq M1	ld8.acq M1
			6				ld8.fill M1
			7				
			8	ld1.c.clr M1	ld2.c.clr M1	ld4.c.clr M1	ld8.c.clr M1
			9	ld1.c.nc M1	ld2.c.nc M1	ld4.c.nc M1	ld8.c.nc M1
			A	ld1.c.clr.acq M1	ld2.c.clr.acq M1	ld4.c.clr.acq M1	ld8.c.clr.acq M1
			B				
			C	st1 M4	st2 M4	st4 M4	st8 M4
			D	st1.rel M4	st2.rel M4	st4.rel M4	st8.rel M4
			E				st8.spill M4
			F				

Table C-29. 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 C-30. 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 C-31. Semaphore/Get FR Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	1	0	cmpxchg1.acq M16	cmpxchg2.acq M16	cmpxchg4.acq M16	cmpxchg8.acq M16
			1	cmpxchg1.rel M16	cmpxchg2.rel M16	cmpxchg4.rel M16	cmpxchg8.rel M16
			2	xchg1 M16	xchg2 M16	xchg4 M16	xchg8 M16
			3				
			4			fetchadd4.acq M17	fetchadd8.acq M17
			5			fetchadd4.rel M17	fetchadd8.rel M17
			6				
			7	getf.sig M19	getf.exp M19	getf.s M19	getf.d M19
			8				
			9				
			A				
			B				
			C				
			D				
			E				
			F				

Table C-32. Floating-point Load/Store/Lfetch Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	0	0	0	ldfe M6	ldf8 M6	ldfs M6	ldfd M6
			1	ldfe.s M6	ldf8.s M6	ldfs.s M6	ldfd.s M6
			2	ldfe.a M6	ldf8.a M6	ldfs.a M6	ldfd.a M6
			3	ldfe.sa M6	ldf8.sa M6	ldfs.sa M6	ldfd.sa M6
			4				
			5				
			6				ldf.fill M6
			7				
			8	ldfe.c.clr M6	ldf8.c.clr M6	ldfs.c.clr M6	ldfd.c.clr M6
			9	ldfe.c.nc M6	ldf8.c.nc M6	ldfs.c.nc M6	ldfd.c.nc M6
			A				
			B	lfetch M13	lfetch.excl M13	lfetch.fault M13	lfetch.fault.excl M13
			C	stfe M9	stf8 M9	stfs M9	stfd M9
			D				
			E				stf.spill M9
			F				

Table C-33. Floating-point Load/Lfetch +Reg Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	1	0	0	ldfe M7	ldf8 M7	ldfs M7	ldfd M7
			1	ldfe.s M7	ldf8.s M7	ldfs.s M7	ldfd.s M7
			2	ldfe.a M7	ldf8.a M7	ldfs.a M7	ldfd.a M7
			3	ldfe.sa M7	ldf8.sa M7	ldfs.sa M7	ldfd.sa M7
			4				
			5				
			6				ldf.fill M7
			7				
			8	ldfe.c.clr M7	ldf8.c.clr M7	ldfs.c.clr M7	ldfd.c.clr M7
			9	ldfe.c.nc M7	ldf8.c.nc M7	ldfs.c.nc M7	ldfd.c.nc M7
			A				
			B	lfetch M14	lfetch.excl M14	lfetch.fault M14	lfetch.fault.excl M14
			C				
			D				
			E				
			F				

Table C-34. Floating-point Load/Store/Lfetch +Imm Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 35:32	Bits 31:30			
		0	1	2	3
7	0	ldfe M8	ldf8 M8	ldfs M8	ldfd M8
	1	ldfe.s M8	ldf8.s M8	ldfs.s M8	ldfd.s M8
	2	ldfe.a M8	ldf8.a M8	ldfs.a M8	ldfd.a M8
	3	ldfe.sa M8	ldf8.sa M8	ldfs.sa M8	ldfd.sa M8
	4				
	5				
	6				ldf.fill M8
	7				
	8	ldfe.c.clr M8	ldf8.c.clr M8	ldfs.c.clr M8	ldfd.c.clr M8
	9	ldfe.c.nc M8	ldf8.c.nc M8	ldfs.c.nc M8	ldfd.c.nc M8
	A				
	B	lfetch M15	lfetch.excl M15	lfetch.fault M15	lfetch.fault.excl M15
	C	stfe M10	stf8 M10	stfs M10	stfd M10
	D				
	E				stf.spill M10
	F				

Table C-35. 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 C-36. Floating-point Load Pair +Imm Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	1	1	0		ldfp8 M12	ldfps M12	ldfpd M12
			1		ldfp8.s M12	ldfps.s M12	ldfpd.s M12
			2		ldfp8.a M12	ldfps.a M12	ldfpd.a M12
			3		ldfp8.sa M12	ldfps.sa M12	ldfpd.sa M12
			4				
			5				
			6				
			7				
			8		ldfp8.c.clr M12	ldfps.c.clr M12	ldfpd.c.clr M12
			9		ldfp8.c.nc M12	ldfps.c.nc M12	ldfpd.c.nc M12
			A				
			B				
			C				
			D				
			E				
			F				

The load and store instructions all have a 2-bit opcode extension field in bits 29:28 (hint) which encodes locality hint information. Table C-37 and Table C-38 summarize these assignments.

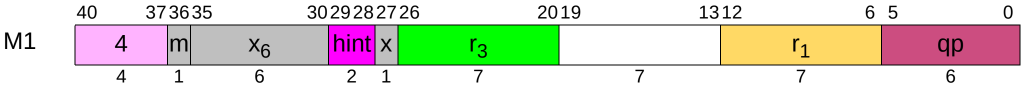
Table C-37. Load Hint Completer

Hint Bits 29:28	*ldhint*
0	*none*
1	.nt1
2	
3	.nta

Table C-38. Store Hint Completer

Hint Bits 29:28	*sthint*
0	*none*
1	
2	
3	.nta

C.4.1.1. Integer Load



Instruction	Operands	Opcode	Extension			
			m	x	x_6	Hint
ld1.*ldhint*	r_1 = [r_3]	4	0	0	00	Table C-37 on page C-37
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	

C.4.1.2. Integer Load – Increment by Register

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M2	4	m	x_6	hint	x	r_3	r_2	r_1	qp
	4	1	6	2	1	7	7	7	6

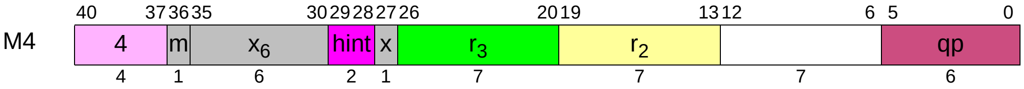
Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ld1.*ldhint*	r_1 = [r_3], r_2	4	1	0	00	Table C-37 on page C-37
ld2.*ldhint*					01	
ld4.*ldhint*					02	
ld8.*ldhint*					03	
ld1.s.*ldhint*					04	
ld2.s.*ldhint*					05	
ld4.s.*ldhint*					06	
ld8.s.*ldhint*					07	
ld1.a.*ldhint*					08	
ld2.a.*ldhint*					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	

C.4.1.3. Integer Load – Increment by Immediate

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M3	5	s	x_6	hint	i	r_3	imm_{7b}	r_1	qp
	4	1	6	2	1	7	7	7	6

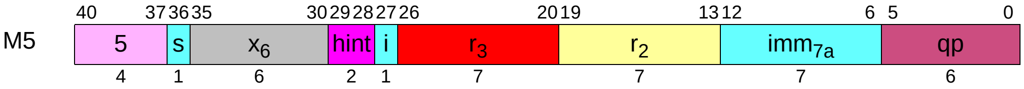
Instruction	Operands	Opcode	Extension	
			x_6	Hint
ld1.*ldhint*	r_1 = [r_3], imm_9	5	00	Table C-37 on page C-37
ld2.*ldhint*			01	
ld4.*ldhint*			02	
ld8.*ldhint*			03	
ld1.s.*ldhint*			04	
ld2.s.*ldhint*			05	
ld4.s.*ldhint*			06	
ld8.s.*ldhint*			07	
ld1.a.*ldhint*			08	
ld2.a.*ldhint*			09	
ld4.a.*ldhint*			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	

C.4.1.4. Integer Store



Instruction	Operands	Opcode	Extension			
			m	x	x_6	Hint
st1.*sthint*	$[r_3] = r_2$	4	0	0	30	Table C-38 on page C-37
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	

C.4.1.5. Integer Store – Increment by Immediate



Instruction	Operands	Opcode	Extension	
			x_6	Hint
st1.*sthint*	$[r_3] = r_2, imm_9$	5	30	Table C-38 on page C-37
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	

C.4.1.6. Floating-point Load

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M6	6	m	x_6	hint	x	r_3		f_1	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	Hint
ldfs.*ldhint*	f_1 = [r_3]	6	0	0	02	Table C-37 on page C-37
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	

C.4.1.7. Floating-point Load – Increment by Register

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M7	6	m	x_6	hint	x	r_3	r_2	f_1	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	Hint
ldfs.*ldhint*	f_1 = [r_3], r_2	6	1	0	02	Table C-37 on page C-37
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	

C.4.1.8. Floating-point Load – Increment by Immediate

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M8	7	s	x_6	hint	i	r_3	imm_{7b}	f_1	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_6	Hint
ldfs.*ldhint*	f_1 = [r_3], imm_9	7	02	Table C-37 on page C-37
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	

C.4.1.9. Floating-point Store

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M9	6	m	x_6	hint	x	r_3	f_2		qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	Hint
stfs.*sthint*	$[r_3] = f_2$	6	0	0	32	Table C-38 on page C-37
stfd.*sthint*					33	
stf8.*sthint*					31	
stfe.*sthint*					30	
stf.spill.*sthint*					3B	

C.4.1.10. Floating-point Store – Increment by Immediate

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M10	7	s	x_6	hint	i	r_3	f_2	imm_{7a}	qp
	4	1	6	2	1	7	7	7	6

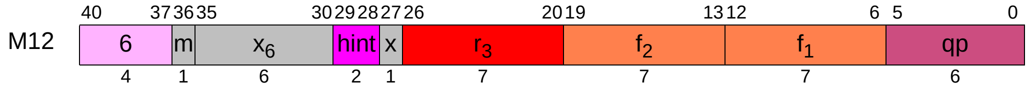
Instruction	Operands	Opcode	Extension	
			x_6	Hint
stfs.*sthint*	$[r_3] = f_2, imm_9$	7	32	Table C-38 on page C-37
stfd.*sthint*			33	
stf8.*sthint*			31	
stfe.*sthint*			30	
stf.spill.*sthint*			3B	

C.4.1.11. Floating-point Load Pair

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M11	6	m	x_6	hint	x	r_3	f_2	f_1	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	Hint
ldfps.*ldhint*	$f_1, f_2 = [r_3]$	6	0	1	02	Table C-37 on page C-37
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	

C.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	Table C-37 on page C-37
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	

C.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 C-32 for a summary of the opcode extensions.

The line prefetch instructions all have a 2-bit opcode extension field in bits 29:28 (hint) which encodes locality hint information as shown in Table C-39.

Table C-39. Line Prefetch Hint Completer

Hint Bits 29:28	*lfhint*
0	*none*
1	.nt1
2	.nt2
3	.nta

C.4.2.1. Line Prefetch

	40 37	36	35 30	29 28	27	26 20	19 6	5 0
M13	6	m	x_6	hint	x	r_3		qp
	4	1	6	2	1	7	14	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	Hint
lfetch.*lfhint*	[r_3]	6	0	0	2C	Table C-39 on page C-47
lfetch.excl.*lfhint*					2D	
lfetch.fault.*lfhint*					2E	
lfetch.fault.excl.*lfhint*					2F	

C.4.2.2. Line Prefetch – Increment by Register

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M14	6	m	x_6	hint	x	r_3	r_2		qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	Hint
lfetch.*lfhint*	[r_3], r_2	6	1	0	2C	Table C-39 on page C-47
lfetch.excl.*lfhint*					2D	
lfetch.fault.*lfhint*					2E	
lfetch.fault.excl.*lfhint*					2F	

C.4.2.3. Line Prefetch – Increment by Immediate

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M15	7	s	x_6	hint	i	r_3	imm_{7b}		qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_6	Hint
lfetch.*lfhint*	[r_3], imm_9	7	2C	Table C-39 on page C-47
lfetch.excl.*lfhint*			2D	
lfetch.fault.*lfhint*			2E	
lfetch.fault.excl.*lfhint*			2F	

C.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 C-32 for a summary of the opcode extensions.

C.4.3.1. Exchange/Compare and Exchange

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M16	4	m	x_6	hint	x	r_3	r_2	r_1	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	Hint
cmpxchg1.acq.*ldhint*	r_1 = [r_3], r_2, ar.ccv	4	0	1	00	Table C-37 on page C-37
cmpxchg2.acq.*ldhint*					01	
cmpxchg4.acq.*ldhint*					02	
cmpxchg8.acq.*ldhint*					03	
cmpxchg1.rel.*ldhint*					04	
cmpxchg2.rel.*ldhint*					05	
cmpxchg4.rel.*ldhint*					06	
cmpxchg8.rel.*ldhint*					07	
xchg1.*ldhint*	r_1 = [r_3], r_2				08	
xchg2.*ldhint*					09	
xchg4.*ldhint*					0A	
xchg8.*ldhint*					0B	

C.4.3.2. Fetch and Add – Immediate

	40 37	36	35 30	29 28	27	26 20	19 16	15	14 13	12 6	5 0
M17	4	m	x_6	hint	x	r_3		s	i_{2b}	r_1	qp
	4	1	6	2	1	7	4	1	2	7	6

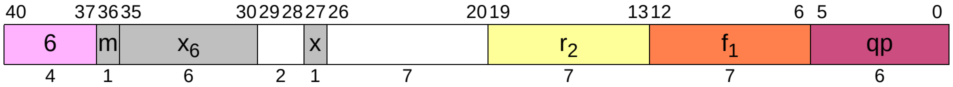
Instruction	Operands	Opcode	Extension			
			m	x	x_6	Hint
fetchadd4.acq.*ldhint*	r_1 = [r_3], inc_3	4	0	1	12	Table C-37 on page C-37
fetchadd8.acq.*ldhint*					13	
fetchadd4.rel.*ldhint*					16	
fetchadd8.rel.*ldhint*					17	

C.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 C-32 for a summary of the opcode extensions.

C.4.4.1. Set FR

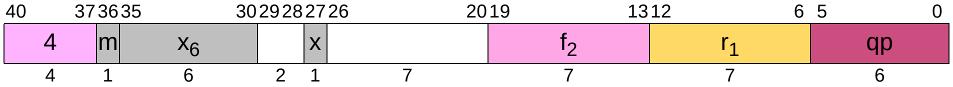
M18



Instruction	Operands	Opcode	Extension		
			m	x	x_6
setf.sig	$f_1 = r_2$	6	0	1	1C
setf.exp					1D
setf.s					1E
setf.d					1F

C.4.4.2. Get FR

M19



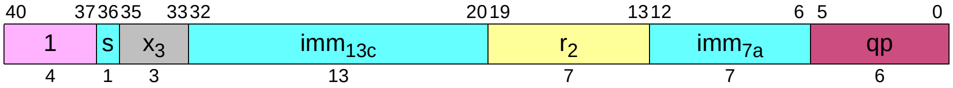
Instruction	Operands	Opcode	Extension		
			m	x	x_6
getf.sig	$r_1 = f_2$	4	0	1	1C
getf.exp					1D
getf.s					1E
getf.d					1F

C.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 memory management instructions. See "Memory Management" on page C-55 for a summary of the opcode extensions.

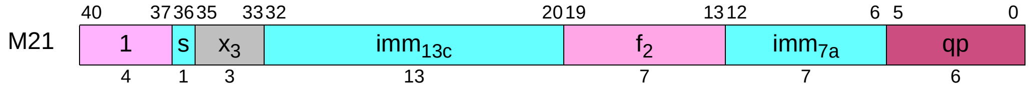
C.4.5.1. Integer Speculation Check (M-Unit)

M20



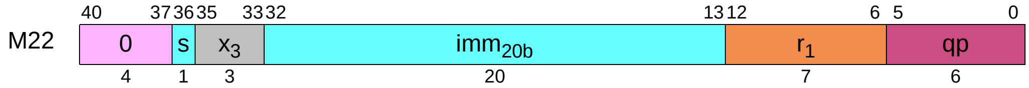
Instruction	Operands	Opcode	Extension
			x_3
chk.s.m	r_2, $target_{25}$	1	1

C.4.5.2. Floating-point Speculation Check



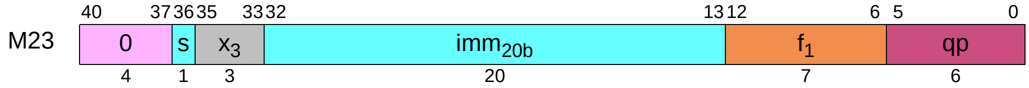
Instruction	Operands	Opcode	Extension
			x_3
chk.s	f_2, $target_{25}$	1	3

C.4.5.3. Integer Advanced Load Check



Instruction	Operands	Opcode	Extension
			x_3
chk.a.nc	r_1, $target_{25}$	0	4
chk.a.clr			5

C.4.5.4. Floating-point Advanced Load Check

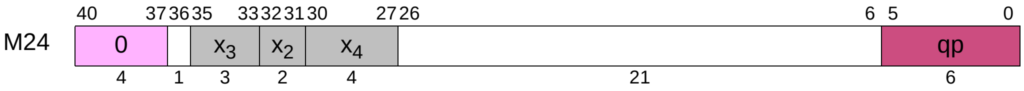


Instruction	Operands	Opcode	Extension
			x_3
chk.a.nc	f_1, $target_{25}$	0	6
chk.a.clr			7

C.4.6 Cache/Synchronization/RSE/ALAT

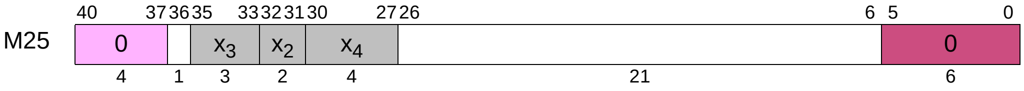
The cache/synchronization/RSE/ALAT instructions are encoded in major opcode 0 along with the memory management instructions. See “Memory Management” on page C-55 for a summary of the opcode extensions.

C.4.6.1. Sync/Fence/Serialize/ALAT Control



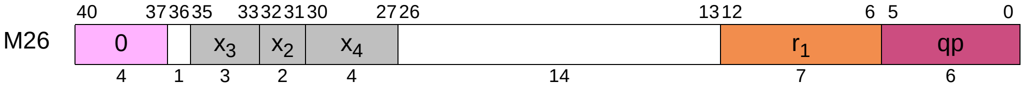
Instruction	Opcode	Extension		
		x_3	x_4	x_2
invala	0	0	0	1
mf			2	2
mf.a			3	
srlz.i			1	3
sync.i			3	

C.4.6.2. RSE Control



Instruction	Opcode	Extension		
		x_3	x_4	x_2
flushrs [f]	0	0	C	0

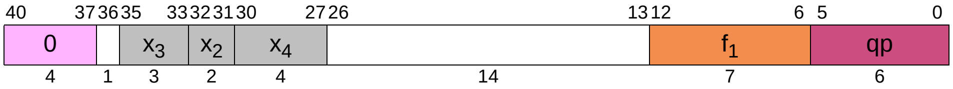
C.4.6.3. Integer ALAT Entry Invalidate



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
invala.e	r_1	0	0	2	1

C.4.6.4. Floating-point ALAT Entry Invalidate

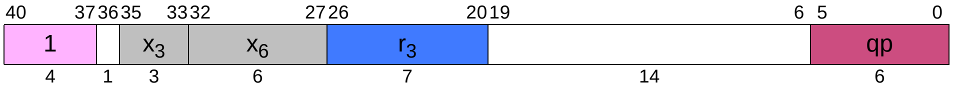
M27



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
invala.e	f_1	0	0	3	1

C.4.6.5. Flush Cache

M28



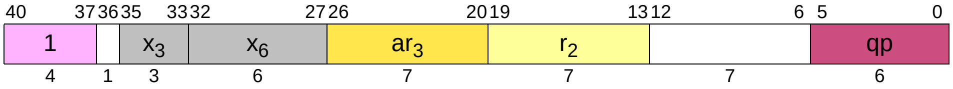
Instruction	Operands	Opcode	Extension	
			x_3	x_6
fc	r_3	1	0	30

C.4.7 GR/AR Moves (M-Unit)

The M-Unit GR/AR move instructions are encoded in major opcode 0 along with the memory management instructions. (Some ARs are accessed using system control instructions on the I-unit. See "GR/AR Moves (I-Unit)" on page C-30.) See "Memory Management" on page C-55 for a summary of the M-Unit GR/AR opcode extensions.

C.4.7.1. Move to AR – Register (M-Unit)

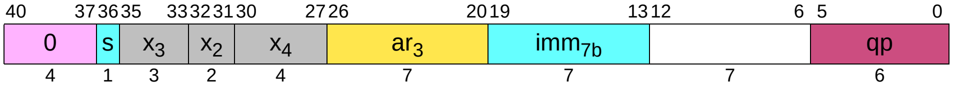
M29



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.m	$ar_3 = r_2$	1	0	2A

C.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

C.4.7.3. Move from AR (M-Unit)

	40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
M31	1		x_3	x_6	ar_3		r_1	qp
	4	1	3	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.m	$r_1 = ar_3$	1	0	22

C.4.8 Miscellaneous M-Unit Instructions

The miscellaneous M-unit instructions are encoded in major opcode 0 along with the memory management instructions. See “Memory Management” on page C-55 for a summary of the opcode extensions.

C.4.8.1. Allocate Register Stack Frame

	40 37	36	35 33	32 31	30 27	26 20	19 13	12 6	5 0
M34	1		x_3		sor	sol	sof	r_1	0
	4	1	3	2	4	7	7	7	6

Instruction	Operands	Opcode	Extension
			x_3
alloc [f]	r_1 = ar.pfs, i, l, o, r	1	6

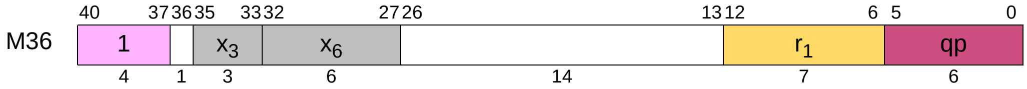
Note: The three immediates in the instruction encoding are formed from the operands as follows:
sof = $i + l + o$
sol = $i + l$
sor = $r >> 3$

C.4.8.2. Move to PSR

	40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
M35	1		x_3	x_6		r_2		qp
	4	1	3	6	7	7	7	6

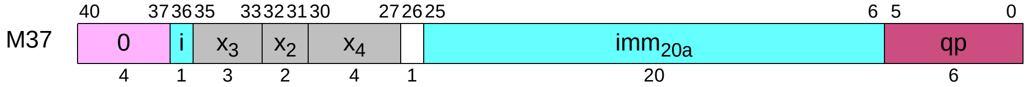
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	psr.um = r_2	1	0	29

C.4.8.3. Move from PSR



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	r_1 = psr.um	1	0	21

C.4.8.4. Break/Nop (M-Unit)



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
break.m	imm_{21}	0	0	0	0
nop.m				1	

C.4.9 Memory Management

All 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 C-40 shows the 3-bit assignments for opcode 0, Table C-41 summarizes the 4-bit+2-bit assignments for opcode 0, Table C-42 shows the 3-bit assignments for opcode 1, and Table C-43 summarizes the 6-bit assignments for opcode 1.

Table C-40. Opcode 0 Memory Management 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	Memory Management 4-bit+2-bit Ext (Table C-41)
	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 C-41. Opcode 0 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		
		1	nop.m M37			srlz.i M24
		2		invala.e – int M26	mf M24	
		3		invala.e – fp M27	mf.a M24	sync.i M24
		4	sum M44			
		5	rum M44			
		6				
		7				
		8			mov.m to ar – imm$_8$ M30	
		9				
		A				
		B				
		C	flushrs M25			
		D				
		E				
		F				

Table C-42. Opcode 1 Memory Management 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
1	0	Memory Management 6-bit Ext (Table C-43)
	1	chk.s.m – int M20
	2	
	3	chk.s – fp M21
	4	
	5	
	6	alloc M34
	7	

Table C-43. Opcode 1 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				fc M28
		1			mov from psr.um M36	
		2			mov.m from ar M31	
		3				
		4				
		5		mov from pmd M43		
		6				
		7		mov from cpuid M43		
		8				
		9			mov to psr.um M35	
		A			mov.m to ar M29	
		B				
		C				
		D				
		E				
		F				

C.4.9.1. Move from Indirect Register

	40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
M43	1		x_3	x_6	r_3		r_1	qp
	4	1	3	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	r_1 = pmd[r_3]	1	0	15
	r_1 = cpuid[r_3]			17

C.4.9.2. Set/Reset User Mask

	40 37	36	35 33	32 31	30 27	26 6	5 0
M44	0	i	x_3	i_{2d}	x_4	imm_{21a}	qp
	4	1	3	2	4	21	6

Instruction	Operands	Opcode	Extension	
			x_3	x_4
sum	imm_{24}	0	0	4
rum				5

C.5 B-Unit Instruction Encodings

The branch-unit includes branch and miscellaneous instructions.

C.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 C-44.

Table C-44. IP-Relative Branch Types

Opcode Bits 40:37	btype Bits 8:6	
4	0	br.cond B1
	1	🟫
	2	br.wexit B1
	3	br.wtop B1
	4	🟫
	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 C-45 summarizes these assignments.

Table C-45. 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	🟪	Indirect Branch (Table C-46)	🟫
	1		🟪	Indirect Return (Table C-47)	🟫
	2	🟪	🟪	🟫	🟫
	3	🟪	🟪	🟫	🟫
	4	clrrrb B8	🟪	🟫	🟫
	5	clrrrb.pr B8	🟪	🟫	🟫
	6	🟪	🟪	🟫	🟫
	7	🟪	🟪	🟫	🟫
	8	🟪	🟪	🟫	🟫
	9	🟪	🟪	🟫	🟫
	A	🟪	🟪	🟫	🟫
	B	🟪	🟪	🟫	🟫
	C	🟪	🟪	🟫	🟫
	D	🟪	🟪	🟫	🟫
	E	🟪	🟪	🟫	🟫
	F	🟪	🟪	🟫	🟫

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 C-46.

Table C-46. 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	🟫
		3	🟫
		4	🟫
		5	🟫
		6	🟫
		7	🟫

The indirect return branch instructions encoded within major opcodes 0 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table C-47.

Table C-47. Indirect Return Branch Types

Opcode Bits 40:37	x_6 Bits 32:27	btype Bits 8:6	
0	21	0	
		1	
		2	
		3	
		4	br.ret B4
		5	
		6	
		7	

All of the branch instructions have a 1-bit opcode extension field, p, in bit 12 which provides a sequential prefetch hint. Table C-48 summarizes these assignments.

Table C-48. Sequential Prefetch Hint Completer

p Bit 12	*ph*
0	.few
1	.many

The IP-relative and indirect branch instructions all have a 2-bit opcode extension field in bits 34:33 (wh) which encodes branch prediction "whether" hint information as shown in Table C-49. Indirect call instructions have a 3-bit opcode extension field in bits 34:32 (wh) for "whether" hint information as shown in Table C-50.

Table C-49. Branch Whether Hint Completer

wh Bits 34:33	*bwh*
0	.sptk
1	.spnt
2	.dptk
3	.dpnt

Table C-50. Indirect Call Whether Hint Completer

wh Bits 34:32	*bwh*
0	
1	.sptk
2	
3	.spnt
4	
5	.dptk

Table C-50. Indirect Call Whether Hint Completer

wh Bits 34:32	*bwh*
6	
7	.dpnt

The branch instructions also have a 1-bit opcode extension field in bit 35 (d) which encodes a branch cache deallocation hint as shown in Table C-51.

Table C-51. Branch Cache Deallocation Hint Completer

d Bit 35	*dh*
0	*none*
1	.clr

C.5.1.1. IP-Relative Branch

	40 37	36	35	34 33 32	13	12	11 9	8 6	5 0
B1	4	s	d	wh	imm$_{20b}$	p		btype	qp
	4	1	1	2	20	1	3	3	6

Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cond.*bwh.ph.dh*	*target*$_{25}$	4	0	Table C-48 on page C-60	Table C-49 on page C-60	Table C-51 on page C-61
br.wexit.*bwh.ph.dh*^{t}			2			
br.wtop.*bwh.ph.dh*^{t}			3			

C.5.1.2. IP-Relative Counted Branch

	40 37	36	35	34 33 32	13	12	11 9	8 6	5 0
B2	4	s	d	wh	imm$_{20b}$	p		btype	0
	4	1	1	2	20	1	3	3	6

Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cloop.*bwh.ph.dh*^{t}	*target*$_{25}$	4	5	Table C-48 on page C-60	Table C-49 on page C-60	Table C-51 on page C-61
br.cexit.*bwh.ph.dh*^{t}			6			
br.ctop.*bwh.ph.dh*^{t}			7			

C.5.1.3. IP-Relative Call

	40 37	36	35	34 33	32 13	12	11 9	8 6	5 0
B3	5	s	d	wh	imm_{20b}	p		b_1	qp
	4	1	1	2	20	1	3	3	6

Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.*bwh.ph.dh*	b_1 = $target_{25}$	5	Table C-48 on page C-60	Table C-49 on page C-60	Table C-51 on page C-61

C.5.1.4. Indirect Branch

	40 37	36	35	34 33	32 27	26 16	15 13	12	11 9	8 6	5 0
B4	0		d	wh	x_6		b_2	p		btype	qp
	4	1	1	2	6	11	3	1	3	3	6

Instruction	Operands	Opcode	Extension				
			x_6	btype	p	wh	d
br.cond.*bwh.ph.dh*	b_2	0	20	0	Table C-48 on page C-60	Table C-49 on page C-60	Table C-51 on page C-61
br.ia.*bwh.ph.dh*				1			
br.ret.*bwh.ph.dh*			21	4			

C.5.1.5. Indirect Call

	40 37	36	35	34 32	31 16	15 13	12	11 9	8 6	5 0
B5	1		d	wh		b_2	p		b_1	qp
	4	1	1	3	16	3	1	3	3	6

Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.*bwh.ph.dh*	$b_1 = b_2$	1	Table C-48 on page C-60	Table C-50 on page C-60	Table C-51 on page C-61

C.5.2 Nop

The nop instruction is encoded in major opcode 2. The nop instruction in major opcode 2 uses a 6-bit opcode extension field in bits 32:27 (x_6). Table C-52 summarizes these assignments.

Table C-52. Indirect Predict/Nop Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 30:27	Bits 32:31			
		0	1	2	3
2	0	nop.b B9			
	1				
	2				
	3				
	4				
	5				
	6				
	7				
	8				
	9				
	A				
	B				
	C				
	D				
	E				
	F				

C.5.3 Miscellaneous B-Unit Instructions

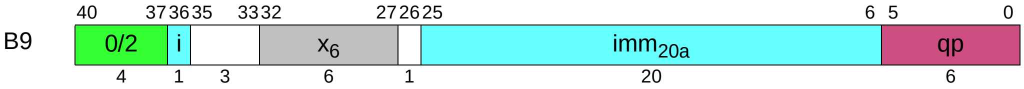
The miscellaneous branch-unit instructions include a number of instructions encoded within major opcode 0 using a 6-bit opcode extension field in bits 32:27 (x_6) as described in Table C-45 on page C-59.

C.5.3.1. Miscellaneous (B-Unit)

	40 37	36 33	32 27	26 6	5 0
B8	0		x_6		0
	4	4	6	21	6

Instruction	Opcode	Extension
		x_6
clrrrb ^{l}	0	04
clrrrb.pr ^{l}		05

C.5.3.2. Break/Nop (B-Unit)



Instruction	Operands	Opcode	Extension
			x_6
break.b	imm_{21}	0	00
nop.b		2	

C.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 C-53 shows the 1-bit x assignments, Table C-56 shows the additional 1-bit q assignments for the reciprocal approximation instructions; Table C-54 and Table C-55 summarize the 6-bit x_6 assignments.

Table C-53. Miscellaneous Floating-point 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	
0	0	6-bit Ext (Table C-54)
	1	Reciprocal Approximation (Table C-56)
1	0	6-bit Ext (Table C-55)
	1	Reciprocal Approximation (Table C-56)

Table C-54. Opcode 0 Miscellaneous Floating-point 6-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
0	0	0	break.f F15	fmerge.s F9		
		1	nop.f F15	fmerge.ns F9		
		2		fmerge.se F9		
		3				
		4	fsetc F12	fmin F8		fswap F9
		5	fclrf F13	fmax F8		fswap.nl F9
		6		famin F8		fswap.nr F9
		7		famax F8		
		8	fchkf F14	fcvt.fx F10	fpack F9	
		9		fcvt.fxu F10		fmix.lr F9
		A		fcvt.fx.trunc F10		fmix.r F9
		B		fcvt.fxu.trunc F10		fmix.l F9
		C		fcvt.xf F11	fand F9	fsxt.r F9
		D			fandcm F9	fsxt.l F9
		E			for F9	
		F			fxor F9	

Table C-55. Opcode 1 Miscellaneous Floating-point 6-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
1	0	0		fpmerge.s F9		fpcmp.eq F8
		1		fpmerge.ns F9		fpcmp.lt F8
		2		fpmerge.se F9		fpcmp.le F8
		3				fpcmp.unord F8
		4		fpmin F8		fpcmp.neq F8
		5		fpmax F8		fpcmp.nlt F8
		6		fpamin F8		fpcmp.nle F8
		7		fpamax F8		fpcmp.ord F8
		8		fpcvt.fx F10		
		9		fpcvt.fxu F10		
		A		fpcvt.fx.trunc F10		
		B		fpcvt.fxu.trunc F10		
		C				
		D				
		E				
		F				

Table C-56. 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 C-57 summarizes these assignments.

Table C-57. Floating-point Status Field Completer

sf Bits 35:34	*sf*
0	.s0
1	.s1
2	.s2
3	.s3

C.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 C-58.

Table C-58. Floating-point Arithmetic 1-bit Opcode Extensions

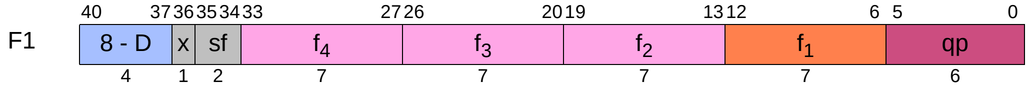
x Bit 36	Opcode Bits 40:37					
	8	9	A	B	C	D
0	fma F1	fma.d F1	fms F1	fms.d F1	fnma F1	fnma.d F1
1	fma.s F1	fpma F1	fms.s F1	fpms F1	fnma.s F1	fpnma F1

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 C-59.

Table C-59. 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

C.6.1.1. Floating-point Multiply Add



Instruction	Operands	Opcode	Extension	
			x	sf
fma.*sf*	$f_1 = f_3, f_4, f_2$	8	0	Table C-57 on page C-66
fma.s.*sf*			1	
fma.d.*sf*		9	0	
fpma.*sf*			1	
fms.*sf*		A	0	
fms.s.*sf*			1	
fms.d.*sf*		B	0	
fpms.*sf*			1	
fnma.*sf*		C	0	
fnma.s.*sf*			1	
fnma.d.*sf*		D	0	
fpnma.*sf*			1	

C.6.1.2. Fixed-point Multiply Add

	40 37	36	35 34	33 27	26 20	19 13	12 6	5 0
F2	E	x	x_2	f_4	f_3	f_2	f_1	qp
	4	1	2	7	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x	x_2
xma.l	$f_1 = f_3, f_4, f_2$	E	1	0
xma.h				3
xma.hu				2

C.6.2 Parallel Floating-point Select

	40 37	36	35 34	33 27	26 20	19 13	12 6	5 0
F3	E	x		f_4	f_3	f_2	f_1	qp
	4	1	2	7	7	7	7	6

Instruction	Operands	Opcode	Extension
			x
fselect	$f_1 = f_3, f_4, f_2$	E	0

C.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 C-60. The sf assignments are shown in Table C-57 on page C-66.

The parallel floating-point compare instructions are described on page C-71.

Table C-60. 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 C-61.

Table C-61. 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

C.6.3.1. Floating-point Compare

	40 37	36	35 34	33	32 27	26 20	19 13	12	11 6	5 0
F4	4	r_b	sf	r_a	p_2	f_3	f_2	t_a	p_1	qp
	4	1	2	1	6	7	7	1	6	6

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	Table C-57 “Floating-point Status Field Completer” on page C-66
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		

C.6.3.2. Floating-point Class

	40 37	36 35	34 33	32 27	26 20	19 13	12	11 6	5 0
F5	5		fc_2	p_2	$fclass_{7c}$	f_2	t_a	p_1	qp
	4	2	2	6	7	7	1	6	6

Instruction	Operands	Opcode	Extension
			t_a
fclass.m	$p_1, p_2 = f_2, fclass_9$	5	0
fclass.m.unc			1

C.6.4 Approximation

C.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.

	40 37	36	35 34	33	32 27	26 20	19 13	12 6	5 0
F6	0 - 1	q	sf	x	p_2	f_3	f_2	f_1	qp
	4	1	2	1	6	7	7	7	6

Instruction	Operands	Opcode	Extension		
			x	q	sf
frcpa.*sf*	$f_1, p_2 = f_2, f_3$	0	1	0	Table C-57 on page C-66
fprcpa.*sf*		1			

C.6.4.2. Floating-point Reciprocal Square Root Approximation

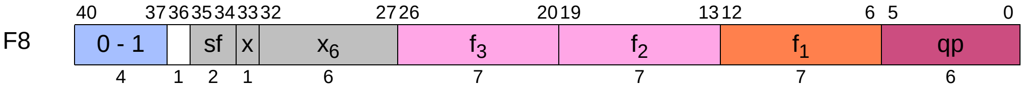
There are two Reciprocal Square Root Approximation instructions. The first, in major op 0, encodes the full register variant. The second, in major op 1, encodes the parallel variant.

	40 37	36	35 34	33	32 27	26 20	19 13	12 6	5 0
F7	0 - 1	q	sf	x	p_2	f_3		f_1	qp
	4	1	2	1	6	7	7	7	6

Instruction	Operands	Opcode	Extension		
			x	q	sf
frsqrta.*sf*	$f_1, p_2 = f_3$	0	1	1	Table C-57 on page C-66
fprsqrta.*sf*		1			

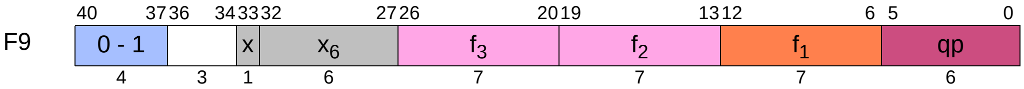
C.6.5 Minimum/Maximum and Parallel Compare

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	Table C-57 on page C-66
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	

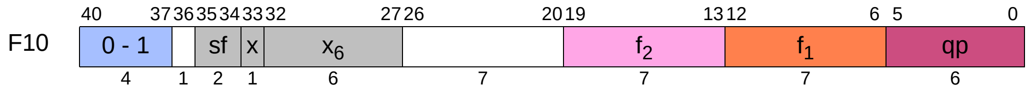
C.6.6 Merge and Logical



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

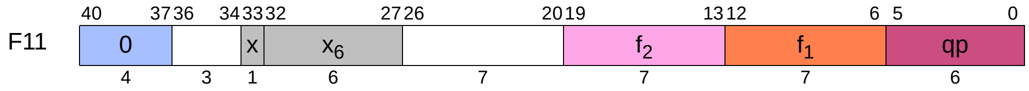
C.6.7 Conversion

C.6.7.1. Convert Floating-point to Fixed-point



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fcvt.fx.*sf*	$f_1 = f_2$	0	0	18	Table C-57 on page C-66
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	

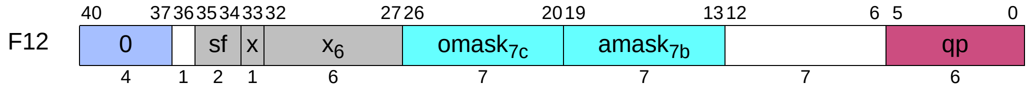
C.6.7.2. Convert Fixed-point to Floating-point



Instruction	Operands	Opcode	Extension	
			x	x_6
fcvt.xf	$f_1 = f_2$	0	0	1C

C.6.8 Status Field Manipulation

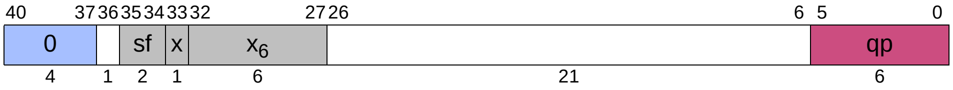
C.6.8.1. Floating-point Set Controls



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fsetc.*sf*	$amask_7, omask_7$	0	0	04	Table C-57 on page C-66

C.6.8.2. Floating-point Clear Flags

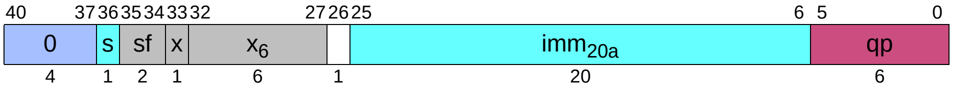
F13



Instruction	Opcode	Extension		
		x	x_6	sf
fclrf.*sf*	0	0	05	Table C-57 on page C-66

C.6.8.3. Floating-point Check Flags

F14

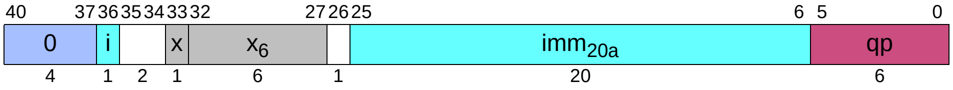


Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fchkf.*sf*	$target_{25}$	0	0	08	Table C-57 on page C-66

C.6.9 Miscellaneous F-Unit Instructions

C.6.9.1. Break/Nop (F-Unit)

F15



Instruction	Operands	Opcode	Extension	
			x	x_6
break.f	imm_{21}	0	0	00
nop.f				01

C.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.

C.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 C-62 shows the 3-bit assignments and Table C-63 summarizes the 6-bit assignments. These instructions are executed by an I-unit.

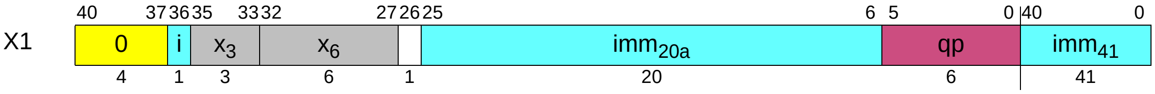
Table C-62. Misc X-Unit 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	6-bit Ext (Table C-63)
	1	
	2	
	3	
	4	
	5	
	6	
	7	

Table C-63. Misc X-Unit 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
0	0	0	break.x X1			
		1	nop.x X1			
		2				
		3				
		4				
		5				
		6				
		7				
		8				
		9				
		A				
		B				
		C				
		D				
		E				
		F				

C.7.1.1. Break/Nop (X-Unit)



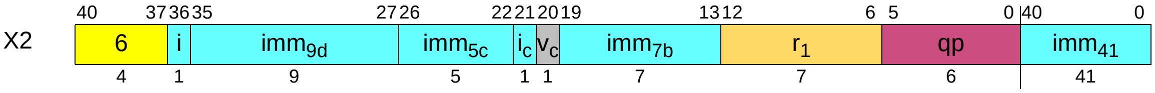
Instruction	Operands	Opcode	Extension	
			x_3	x_6
break.x	imm_{62}	0	0	00
nop.x				01

C.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 C-64. This instruction is executed by an I-unit.

Table C-64. Move Long 1-bit Opcode Extensions

Opcode Bits 40:37	v_c Bit 20	
6	0	movl X2
	1	



Instruction	Operands	Opcode	Extension
			v_c
movl	$r_1 = imm_{64}$	6	0

C.8 Immediate Formation

Table C-65 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 C-65. Immediate Formation

Instruction Format	Immediate Formation
A2	$count_2 = ct_{2d} + 1$
A3 A8 I27 M30	$imm_8 = sign\_ext(s \ll 7 \mid imm_{7b}, 8)$
A4	$imm_{14} = sign\_ext(s \ll 13 \mid imm_{6d} \ll 7 \mid imm_{7b}, 14)$
A5	$imm_{22} = sign\_ext(s \ll 21 \mid imm_{5c} \ll 16 \mid imm_{9d} \ll 7 \mid imm_{7b}, 22)$
A10	$count_2 = (ct_{2d} > 2)\ ?\ reservedQP$[a] $:\ ct_{2d} + 1$
I1	$count_2 = (ct_{2d} == 0)\ ?\ 0 : (ct_{2d} == 1)\ ?\ 7 : (ct_{2d} == 2)\ ?\ 15 : 16$
I3	$mbtype_4 = (mbt_{4c} == 0)\ ?\ @brcst : (mbt_{4c} == 8)\ ?\ @mix : (mbt_{4c} == 9)\ ?\ @shuf : (mbt_{4c} == 0xA)\ ?\ @alt : (mbt_{4c} == 0xB)\ ?\ @rev : reservedQP$[a]
I4	$mhtype_8 = mht_{8c}$
I6	$count_5 = count_{5b}$
I8	$count_5 = 31 - ccount_{5c}$
I10	$count_6 = count_{6d}$
I11	$len_6 = len_{6d} + 1$ $pos_6 = pos_{6b}$
I12	$len_6 = len_{6d} + 1$ $pos_6 = 63 - cpos_{6c}$
I13	$len_6 = len_{6d} + 1$ $pos_6 = 63 - cpos_{6c}$ $imm_8 = sign\_ext(s \ll 7 \mid imm_{7b}, 8)$
I14	$len_6 = len_{6d} + 1$ $pos_6 = 63 - cpos_{6b}$ $imm_1 = sign\_ext(s, 1)$
I15	$len_4 = len_{4d} + 1$ $pos_6 = 63 - cpos_{6d}$
I16	$pos_6 = pos_{6b}$
I19 M37	$imm_{21} = i \ll 20 \mid imm_{20a}$
I23	$mask_{17} = sign\_ext(s \ll 16 \mid mask_{8c} \ll 8 \mid mask_{7a} \ll 1, 17)$
I24	$imm_{44} = sign\_ext(s \ll 43 \mid imm_{27a} \ll 16, 44)$
M3 M8 M15	$imm_9 = sign\_ext(s \ll 8 \mid i \ll 7 \mid imm_{7b}, 9)$
M5 M10	$imm_9 = sign\_ext(s \ll 8 \mid i \ll 7 \mid imm_{7a}, 9)$
M17	$inc_3 = sign\_ext(((s)\ ?\ -1 : 1) * ((i_{2b} == 3)\ ?\ 1 : 1 \ll (4 - i_{2b})), 6)$
I20 M20 M21	$target_{25} = IP + (sign\_ext(s \ll 20 \mid imm_{13c} \ll 7 \mid imm_{7a}, 21) \ll 4)$
M22 M23	$target_{25} = IP + (sign\_ext(s \ll 20 \mid imm_{20b}, 21) \ll 4)$

Table C-65. Immediate Formation (Cont'd)

Instruction Format	Immediate Formation

M34	il = sol
	o = sof – sol
	r = sor << 3

M44	imm_{24} = i << 23 | i_{2d} << 21 | imm_{21a}
B1 B2 B3	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20b}, 21) << 4)
B9	imm_{21} = i << 20 | imm_{20a}
F5	$fclass_9$ = $fclass_{7c}$ << 2 | fc_2
F12	$amask_7$ = $amask_{7b}$ $omask_7$ = $omask_{7c}$
F14	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20a}, 21) << 4)
F15	imm_{21} = i << 20 | imm_{20a}
X1	imm_{62} = imm_{41} << 21 | i << 20 | imm_{20a}
X2	imm_{64} = i << 63 | imm_{41} << 22 | i_c << 21 | imm_{5c} << 16 | imm_{9d} << 7 | imm_{7b}

a. This encoding causes an Illegal Operation fault if the value of the qualifying predicate is 1.