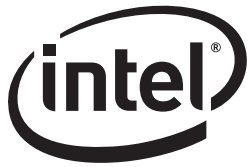


Intel® Itanium® Architecture Software Developer's Manual

Volume 1: Application Architecture

Revision 2.2

January 2006

Document Number: 245317-005

Contents

Part I: Application Architecture Guide

Figures

Part I: Application Architecture Guide

Part II: Optimization Guide for the Intel® Itanium® Architecture

Tables

Part I: Application Architecture Guide

Part II: Optimization Guide for the Intel® Itanium® Architecture

Part I: Application Architecture Guide

About this Manual 1

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

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

1.1 Overview of Volume 1: Application Architecture

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

1.1.1 Part 1: Application Architecture Guide

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 Overview of Volume 2: System Architecture

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

1.2.1 Part 1: System Architecture Guide

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

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

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

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

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

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

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

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

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

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

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

1.2.2 Part 2: System Programmer’s Guide

Chapter 1, “About the System Programmer’s Guide” gives an introduction to the second section of the system architecture guide.

Chapter 2, “MP Coherence and Synchronization” describes multiprocessing synchronization primitives and the Itanium memory ordering model.

Chapter 3, “Interruptions and Serialization” describes how the processor serializes execution around interruptions and what state is preserved and made available to low-level system code when interruptions are taken.

Chapter 4, “Context Management” describes how operating systems need to preserve Itanium register contents and state. This chapter also describes system architecture mechanisms that allow an operating system to reduce the number of registers that need to be spilled/filled on interruptions, system calls, and context switches.

Chapter 5, “Memory Management” introduces various memory management strategies.

Chapter 6, “Runtime Support for Control and Data Speculation” describes the operating system support that is required for control and data speculation.

Chapter 7, “Instruction Emulation and Other Fault Handlers” describes a variety of instruction emulation handlers that Itanium architecture-based operating systems are expected to support.

Chapter 8, “Floating-point System Software” discusses how processors based on the Itanium architecture handle floating-point numeric exceptions and how the software stack provides complete IEEE-754 compliance.

Chapter 9, “IA-32 Application Support” describes the support an Itanium architecture-based operating system needs to provide to host IA-32 applications.

Chapter 10, “External Interrupt Architecture” describes the external interrupt architecture with a focus on how external asynchronous interrupt handling can be controlled by software.

Chapter 11, “I/O Architecture” describes the I/O architecture with a focus on platform issues and support for the existing IA-32 I/O port space.

Chapter 12, “Performance Monitoring Support” describes the performance monitor architecture with a focus on what kind of support is needed from Itanium architecture-based operating systems.

Chapter 13, “Firmware Overview” introduces the firmware model, and how various firmware layers (PAL, SAL, EFI) work together to enable processor and system initialization, and operating system boot.

1.2.3 Appendices

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

1.3 Overview of Volume 3: Instruction Set Reference

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

1.3.1 Part 1: Intel® Itanium® Instruction Set Descriptions

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

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

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

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

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

1.3.2 Part 2: IA-32 Instruction Set Descriptions

Chapter 1, “Base IA-32 Instruction Reference” provides a detailed description of all base IA-32 instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 2, “IA-32 Intel® MMX™ Technology Instruction Reference” provides a detailed description of all IA-32 Intel® MMX™ technology instructions designed to increase performance of multimedia intensive applications. Organized in alphabetical order by assembly language mnemonic.

Chapter 3, “IA-32 SSE Instruction Reference” provides a detailed description of all IA-32 Streaming SIMD Extension (SSE) instructions designed to increase performance of multimedia intensive applications, and is organized in alphabetical order by assembly language mnemonic.

1.4 Terminology

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

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

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

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

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

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

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

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

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

1.5 Related Documents

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

- ***Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*** – This document (Document number 251110) describes model-specific architectural features incorporated into the Intel® Itanium® 2 processor, the second processor based on the Itanium architecture.
- ***Intel® Itanium® Processor Reference Manual for Software Development*** – This document (Document number 245320) describes model-specific architectural features incorporated into the Intel® Itanium® processor, the first processor based on the Itanium architecture.
- ***IA-32 Intel® Architecture Software Developer's Manual*** – This set of manuals describes the Intel 32-bit architecture. They are available from the Intel Literature Department by calling 1-800-548-4725 and requesting Document Numbers 243190, 243191and 243192.
- ***Intel® Itanium® Software Conventions and Runtime Architecture Guide*** – This document (Document number 245358) defines general information necessary to compile, link, and execute a program on an Itanium architecture-based operating system.

- ***Intel® Itanium® Processor Family System Abstraction Layer Specification*** – This document (Document number 245359) specifies requirements to develop platform firmware for Itanium architecture-based systems.
- ***Extensible Firmware Interface Specification*** – This document defines a new model for the interface between operating systems and platform firmware.

1.6 Revision History

Date of Revision	Revision Number	Description
December 2005	2.2	Added TF instruction in Vol 3 Ch 2. Updated IA-32 CPUID I-page in Vol 4 Ch 2. Add support for the absence of INIT, PMI, and LINT pins in Vol 2, Part I, Section 5.8. Add text to "ev" field of Vol 2, Section 7.2.1 Table 7.4 to define a PMU external notification mechanism as implementation dependent. Extensions to PAL procedures to support data poisoning in Vol 2, Part I, Ch 11. Virtualization Addendum - Requires that processors have a way to enable/disable vmsw instruction in Vol 2, Part I, Sections 2.2, 3.4 and 11.9.3. Change the description of CR[IFA] and CR[ITIR] to provide hardware the option of checking them for reserved values on a write. Also mention this option in the description of the Translation Insertion Format. Addition of new return status to PAL_TEST_PROC in Vol 2, Part I, Ch 11. Fix small holes in INTA/XTP definition in Vol 2, Part I, Sections 5.8.4.3 and 5.8.4.4. Virtualization Addendum - Unimplemented Virtual Address Checking in Vol 3 Ch 2. Fix small discrepancies in the cmp8xchg16 definition in Vol 3 Ch 2. Change rules about overlapping inserts to allow Itanium 2 behavior in Vol 2, Part I, Section 4.1.8. Update PAL_BUS_GET/SET_FEATURES bit 52 definition in Vol 2 Ch 11. Allow register fields in CR.LID register to be read-only and CR.LID checking on interruption messages by processors optional. See Vol 2, Part I, Ch 5 “Interruptions” and Section 11.2.2 PALE_RESET Exit State for details. Relaxed reserved and ignored fields checkings in IA-32 application registers in Vol 1 Ch 6 and Vol 2, Part I, Ch 10. Introduced visibility constraints between stores and local purges to ensure TLB consistency for UP VHPT update and local purge scenarios. See Vol 2, Part I, Ch 4 and description of `ptc.l` instruction in Vol 3 for details. Architecture extensions for processor Power/Performance states (P-states). See Vol 2 PAL Chapter for details. Introduced Unimplemented Instruction Address fault. Relaxed ordering constraints for VHPT walks. See Vol 2, Part I, Ch 4 and 5 for details. Architecture extensions for processor virtualization. All instructions which must be last in an instruction group results in undefined behavior when this rule is violated. Added architectural sequence that guarantees increasing ITC and PMD values on successive reads.

Date of Revision	Revision Number	Description
December 2005 (Continued)	2.2	Addition of PAL_BRAND_INFO, PAL_GET_HW_POLICY, PAL_MC_ERROR_INJECT, PAL_MEMORY_BUFFER, PAL_SET_HW_POLICY and PAL_SHUTDOWN procedures. Allows IPI-redirection feature to be optional. Undefined behavior for 1-byte accesses to the non-architected regions in the IPI block. Modified insertion behavior for TR overlaps. See Vol 2, Part I, Ch 4 for details. “Bus parking” feature is now optional for PAL_BUS_GET_FEATURES. FR32-127 is now preserved in PAL calling convention. New return value from PAL_VM_SUMMARY procedure to indicate the number of multiple concurrent outstanding TLB purges. Performance Monitor Data (PMD) registers are no longer sign-extended. New memory attribute transition sequence for memory on-line delete. See Vol 2, Part I, Ch 4 for details. Added 'shared error' (se) bit to the Processor State Parameter (PSP) in PAL_MC_ERROR_INFO procedure. Clarified PMU interrupts as edge-triggered. Modified ‘proc_number’ parameter in PAL_LOGICAL_TO_PHYSICAL procedure. Modified pal_copy_info alignment requirements. New bit in PAL_PROC_GET_FEATURES for variable P-state performance. Clarified descriptions for check_target_register and check_target_register_sof. Various fixes in dependency tables in Vol 3 Ch 5. Clarified effect of sending IPIs to non-existent processor in Vol 2, Part I, Ch 5. Clarified instruction serialization requirements for interruptions in Vol 2, Part II, Ch 3. Updated performance monitor context switch routine in Vol 2, Part I, Ch 7.
October 2002	2.1	Added New `fc.i` Instruction (Sections 4.4.6.1 and 4.4.6.2, Part I, Vol. 1; Sections 4.3.3, 4.4.1, 4.4.5, 4.4.7, 5.5.2, and 7.1.2, Part I, Vol. 2; Sections 2.5, 2.5.1, 2.5.2, 2.5.3, and 4.5.2.1, Part II, Vol. 2; and Sections 2.2, 3, 4.1, 4.4.6.5, and 4.4.10.10, Part I, Vol. 3). Added New Atomic Operations `ld16,st16,cmp8xchg16` (Sections 3.1.8, 3.1.8.6, 4.4.1, 4.4.2, and 4.4.3, Part I, Vol. 1; Section 4.5, Part I, Vol. 2; and Sections 2.2, 3, 5.3.2, and 5.4, Part I, Vol. 3). Added Spontaneous NaT Generation on Speculative Load (Sections 5.5.5 and 11.9, Part I, Vol. 2 and Sections 2.2 and 3, Part I, Vol. 3). Added New Hint Instruction (Section 2.2, Part I, Vol. 3). Added Fault Handling Semantics for `lfetch.fault` Instruction (Section 2.2, Part I, Vol. 3). Added Capability for Allowing Multiple PAL_A_SPEC and PAL_B Entries in the Firmware Interface Table (Section 11.1.6, Part I, Vol. 2). Added BR1 to Min-state Save Area and Clarified Alignment (Sections 11.3.2.3 and 11.3.3, Part I, Vol. 2). Added New PAL Procedures: PAL_LOGICAL_TO_PHYSICAL and PAL_CACHE_SHARED_INFO (Section 11.9.1, Part I, Vol. 2). Added Op Fields to PAL_MC_ERROR_INFO (Section 11.9, Part I, Vol. 2). Added New Error Exit States (Section 11.2.2.2, Part I, Vol. 2). Added Performance Counter Standardization (Sections 7.2.3 and 11.6, Part I, Vol. 2). Modified `CPUID[4]` for Atomic Operations and Spontaneous Deferral (Section 3.1.11, Part I, Vol. 1).

Date of Revision	Revision Number	Description
October 2002 (continued)	2.1	Modified PAL_FREQ_RATIOS (Section 11.2.2, Part I, Vol. 2). Modified PAL_VERSION (Section 11.9, Part I, Vol. 2). Modified PAL_CACHE_INFO Store Hints (Section 11.9, Part I, Vol. 2). Modified PAL_MC_RESUME (Sections 11.3.3 and 11.4, Part I, Vol. 2). Modified IA_32_Exception (Debug) IIPA Description (Section 9.2, Part I, Vol. 2). Clarified Predicate Behavior of `alloc` Instruction (Section 4.1.2, Part I, Vol. 1 and Section 2.2, Part I, Vol. 3). Clarified ITC clocking (Section 3.1.8.10, Part I, Vol. 1; Section 3.3.4.2, Part I, Vol. 2; and Section 10.5.5, Part II, Vol. 2). Clarified Interval Time Counter (ITC) Fault (Section 3.3.2, Part I, Vol. 2). Clarified Interruption Control Registers (Section 3.3.5, Part I, Vol. 2). Clarified Freeze Bit Functionality in Context Switching and Interrupt Generation (Sections 7.2.1, 7.2.2, 7.2.4.1, and 7.2.4.2, Part I, Vol. 2). Clarified PAL_BUS_GET/SET_FEATURES (Section 11.9.3, Part I, Vol. 2). Clarified PAL_CACHE_FLUSH (Section 11.9, Part I, Vol. 2). Clarified Cache State upon Recovery Check (Section 11.2, Part I, Vol. 2). Clarified PALE_INIT Exit State (Section 11.4.2, Part I, Vol. 2). Clarified Processor State Parameter (Section 11.4.2.1, Part I, Vol. 2). Clarified Firmware Address Space at Reset (Section 11.1, Part I, Vol. 2). Clarified PAL PMI, AR.ITC, and PMD Register Values (Sections 11.3, 11.5.1, and 11.5.2, Part I, Vol. 2). Clarified Invalid Arguments to PAL (Section 11.9.2.4, Part I, Vol. 2). Clarified `itr/itc` Instructions (Section 2.2, Part I, Vol. 3).
December 2001	2.0	Volume 1: Faults in ld.c that hits ALAT clarification (Section 4.4.5.3.1). IA-32 related changes (Section 6.2.5.4, Section 6.2.3, Section 6.2.4, Section 6.2.5.3). Load instructions change (Section 4.4.1). Volume 2: Class pr-writers-int clarification (Table A-5). PAL_MC_DRAIN clarification (Section 4.4.6.1). VHPT walk and forward progress change (Section 4.1.1.2). IA-32 IBR/DBR match clarification (Section 7.1.1). ISR figure changes (pp. 8-5, 8-26, 8-33 and 8-36). PAL_CACHE_FLUSH return argument change - added new status return argument (Section 11.8.3). PAL self-test Control and PAL_A procedure requirement change - added new arguments, figures, requirements (Section 11.2). PAL_CACHE_FLUSH clarifications (Section 11). Non-speculative reference clarification (Section 4.4.6). RID and Preferred Page Size usage clarification (Section 4.1). VHPT read atomicity clarification (Section 4.1). IIP and WC flush clarification (Section 4.4.5). Revised RSE and PMC typographical errors (Section 6.4). Revised DV table (Section A.4).

Date of Revision	Revision Number	Description
December 2001 (continued)	2.0	Memory attribute transitions - added new requirements (Section 4.4). MCA for WC/UC aliasing change (Section 4.4.1). Bus lock deprecation - changed behavior of DCR 'lc' bit (Section 3.3.4.1, Section 10.6.8, Section 11.8.3). PAL_PROC_GET/SET_FEATURES changes - extend calls to allow implementation-specific feature control (Section 11.8.3). Split PAL_A architecture changes (Section 11.1.6). Simple barrier synchronization clarification (Section 13.4.2). Limited speculation clarification - added hardware-generated speculative references (Section 4.4.6). PAL memory accesses and restrictions clarification (Section 11.9). PSP validity on INITs from PAL_MC_ERROR_INFO clarification (Section 11.8.3). Speculation attributes clarification (Section 4.4.6). PAL_A FIT entry, PAL_VM_TR_READ, PSP, PAL_VERSION clarifications (Sections 11.8.3 and 11.3.2.1). TLB searching clarifications (Section 4.1). IA-32 related changes (Section 10.3, Section 10.3.2, Section 10.3.2, Section 10.3.3.1, Section 10.10.1). IPSR.ri and ISR.ei changes (Table 3-2, Section 3.3.5.1, Section 3.3.5.2, Section 5.5, Section 8.3, and Section 2.2). Volume 3: IA-32 CPUID clarification (p. 5-71). Revised figures for extract, deposit, and alloc instructions (Section 2.2). RCPPS, RCPSS, RSQRTPS, and RSQRTSS clarification (Section 7.12). IA-32 related changes (Section 5.3). tak, tpa change (Section 2.2).
July 2000	1.1	Volume 1: Processor Serial Number feature removed (Chapter 3). Clarification on exceptions to instruction dependency (Section 3.4.3). Volume 2: Clarifications regarding "reserved" fields in ITIR (Chapter 3). Instruction and Data translation must be enabled for executing IA-32 instructions (Chapters 3,4 and 10). FCR/FDR mappings, and clarification to the value of PSR.ri after an RFI (Chapters 3 and 4). Clarification regarding ordering data dependency. Out-of-order IPI delivery is now allowed (Chapters 4 and 5). Content of EFLAG field changed in IIM (p. 9-24). PAL_CHECK and PAL_INIT calls – exit state changes (Chapter 11). PAL_CHECK processor state parameter changes (Chapter 11). PAL_BUS_GET/SET_FEATURES calls – added two new bits (Chapter 11). PAL_MC_ERROR_INFO call – Changes made to enhance and simplify the call to provide more information regarding machine check (Chapter 11). PAL_ENTER_IA_32_Env call changes – entry parameter represents the entry order; SAL needs to initialize all the IA-32 registers properly before making this call (Chapter 11). PAL_CACHE_FLUSH – added a new cache_type argument (Chapter 11. PAL_SHUTDOWN – removed from list of PAL calls (Chapter 11). Clarified memory ordering changes (Chapter 13). Clarification in dependence violation table (Appendix A).

Date of Revision	Revision Number	Description
July 2000 (continued)	1.1	Volume 3: fmix instruction page figures corrected (Chapter 2). Clarification of “reserved” fields in ITIR (Chapters 2 and 3). Modified conditions for alloc/loadrs/flushrs instruction placement in bundle/ instruction group (Chapters 2 and 4). IA-32 JMPE instruction page typo fix (p. 5-238). Processor Serial Number feature removed (Chapter 5).
January 2000	1.0	Initial release of document.

Introduction to the Intel® Itanium® Architecture 2

The Itanium architecture was designed to overcome the performance limitations of traditional architectures and provide maximum headroom for the future. To achieve this, the Itanium architecture 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 Itanium 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 Itanium architecture also provides binary compatibility with the IA-32 instruction set. Processors based on the Itanium architecture can run IA-32 applications on an Itanium architecture-based operating system that supports execution of IA-32 applications. Such processors can run IA-32 application binaries on IA-32 legacy operating systems assuming the platform and firmware support exists in the system. The Itanium architecture also provides the capability to support mixed IA-32 and Itanium architecture-based code execution.

2.1 Operating Environments

The Itanium architecture supports two operating system environments (see Figure 2-1):

- IA-32 System Environment: supports IA-32 32-bit operating systems.
- Itanium System Environment: supports Itanium architecture-based operating systems.

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

Figure 2-1. System Environments

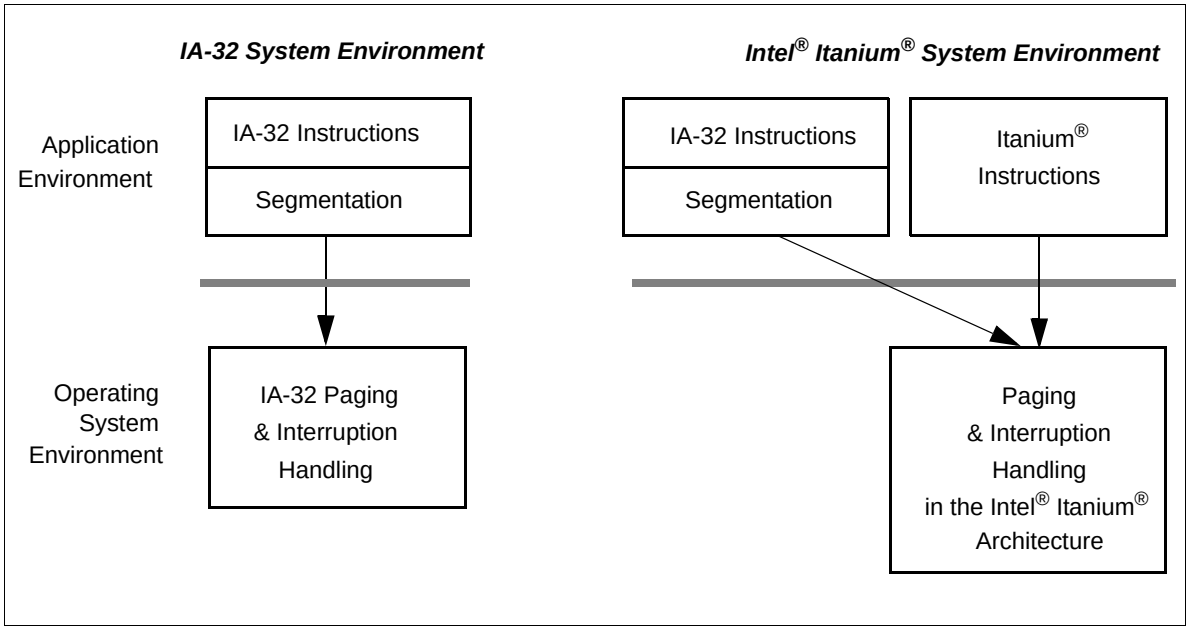


Table 2-1. Major Operating Environments

System Environment	Application Environment	Usage
IA-32 System Environment	IA-32 Instruction Set	IA-32 PM, RM and VM86 application and operating system environment. Compatible with IA-32 Intel® Pentium®, Pentium® Pro, Pentium® II and Pentium® III processors.
	Intel® Itanium® Instruction Set	Not supported, Itanium architecture-based applications cannot execute in the IA-32 System Environment.
Itanium® System Environment	IA-32 Protected Mode	IA-32 Protected Mode applications in the Intel® Itanium® System Environment.
	IA-32 Real Mode	IA-32 Real Mode applications in the Intel® Itanium® System Environment.
	IA-32 Virtual Mode	IA-32 Virtual 86 Mode applications in the Intel® Itanium® System Environment.
	Intel® Itanium® Instruction Set	Itanium architecture-based applications on Intel® Itanium architecture-based operating systems.

2.2 Instruction Set Transition Model Overview

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

- `jmpe` (IA-32 instruction) Jump to an Itanium target instruction, and transition to the Itanium instruction set.
- `br.ia` (Itanium instruction) Branch to an IA-32 target instruction, and change the instruction set to IA-32.
- `rfi` (Itanium instruction) “Return from interruption” is defined to return to an IA-32 or Itanium instruction.
- Interrupts transition the processor to the Itanium instruction set for all interrupt conditions.

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. See Section 6.1, “Instruction Set Modes” for additional details.

2.3 Intel® Itanium® Instruction Set Features

Itanium architecture incorporates features which enable high sustained performance and remove barriers to further performance increases. The Itanium 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
- Focused 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 the Itanium architecture.

2.4 Instruction Level Parallelism

Instruction Level Parallelism (ILP) is the ability to execute multiple instructions at the same time. The Itanium 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 Itanium architecture enables the compiler to manage work in progress and schedule simultaneous threads of computation.

The Itanium 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 Itanium architecture enables the compiler to exploit speculative information without sacrificing the correct execution of an application (see “Speculation” on page 1:14). In traditional architectures, procedure calls limit performance since registers need to be spilled and filled. The Itanium architecture 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 “Register Stack” on page 1:16.

2.5 Compiler to Processor Communication

The Itanium 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, it allows compiled code to manage the processor hardware using runtime information. These communication mechanisms are vital in minimizing the performance penalties associated with branches and cache misses.

The cost of branches is minimized by permitting code to communicate branch information to the hardware in advance of the actual branch.

Every memory load and store in the Itanium architecture 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. A processor based on the Itanium architecture can use this information to determine the placement of cache lines in the cache hierarchy to improve utilization. This is particularly important as the cost of cache misses is expected to increase.

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: (1) it must be statistically frequent enough that the probability it will require recovery is small, and (2) 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. The Itanium architecture 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 dependency to a data dependency. 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 dependency 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 dependency 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

The Itanium architecture 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.

Branch predict instructions are provided which can be used to communicate an early indication of the target address and the location of the branch. The compiler will try to indicate whether a branch should be predicted dynamically or statically. The processor can use this information to initialize branch prediction structures, enabling good prediction even the first time a branch is encountered. This is beneficial for unconditional branches or in situations where the compiler has information about likely branch behavior.

For indirect branches, a branch register is used to hold the target address. Branch predict instructions provide an indication of which register will be used in situations when the target address can be computed early. A branch predict instruction can also signal that an indirect branch is a procedure return, enabling the efficient use of call/return stack prediction structures.

Special loop-closing branches are provided to accelerate counted loops and modulo-scheduled loops. These branches and their associated branch predict instructions 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. The Itanium architecture 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

The Itanium architecture 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.

The Itanium architecture 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

The Itanium architecture 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. They are useful for creating high performance compression/decompression algorithms that are used by applications which have sound and video. Itanium multimedia instructions are semantically compatible with HP's MAX-2* multimedia technology and Intel's MMX and SSE technology instructions.

2.12 Intel® Itanium® System Architecture Features

2.12.1 Support for Multiple Address Space Operating Systems

Most contemporary commercial operating systems utilize a Multiple Address Space (MAS) model with the following characteristics:

Protection is enforced among processes by placing each process within a unique address space. Translation Lookaside Buffers (TLBs), which hold virtual to physical mappings, often need to be flushed on a process context switch.

Some memory areas may be shared among processes, e.g. kernel areas and shared libraries. Most operating systems assume at least one local and one global space.

To promote sharing of data between processes, MAS operating systems aggressively use virtual aliases to map physical memory locations into the address spaces of multiple processes. Virtual aliases create multiple TLB entries for the same physical data leading to reduced TLB efficiency.

The MAS model is supported by dividing the virtual address space into several regions. Region identifiers associated with each region are used to tag translations to a given address space. On a process switch, region identifiers uniquely identify the set of translations belonging to a process, thereby avoiding TLB flushes. Region identifiers also provide a unique intermediate virtual address that help avoid thrashing problems in virtual-indexed caches and TLBs. Regions provide efficient global/shared areas between processes, while reducing the occurrences of virtual aliasing.

2.12.2 Support for Single Address Space Operating Systems

A single address space (SAS) operating system style architecture is the basis for much of the current design work on future 64-bit operating systems. As operating systems (and other large, complex programs like databases) migrate from monolithic programs into cooperating subsystems, an SAS architecture becomes an important performance differentiation in future systems. The SAS or hybrid environments enable a more efficient use of hardware resources.

Common mechanisms are used in both SAS and MAS models such as page level access rights to enforce protection, although the reliance on the feature set will differ under each model. While most of the architected features are utilized in each model, protection keys exist to enable a single global address space operating environment.

2.12.3 System Performance and Scalability

Performance and scalability are achieved through a variety of features. Memory attributes, locking primitives, cache coherency, and memory ordering model work together to allow the efficient sharing of data in a multiprocessor environment. In addition, the Itanium architecture enables low latency fault, trap, and interrupt handlers along with light-weight domain crossings. Performance analysis is aided by the inclusion of several performance monitors, and mechanisms to support software profiling.

2.12.4 System Security and Supportability

Security and supportability result from a number of primitives which provide a very powerful runtime and debug environment. The protection model includes four protection rings and enables increased system integrity by offering a more sophisticated protection scheme than has generally been available. The machine check model allows detailed information to be provided describing the type of error involved and supports recovery for many types of errors. Several mechanisms are provided for debugging both system and application software.

2.13 Terminology

This following terms are used in the remainder of this document:

- **Itanium Instruction Set –** The Itanium architecture defines the 64-bit instruction set extensions to the IA-32 architecture.
- **IA-32 Architecture –** The 32-bit and 16-bit Intel architecture as described in the *IA-32 Intel® Architecture Software Developer's Manual*.
- **Itanium System Environment –** System environment that supports the execution of both IA-32 and Itanium architecture-based code.
- **IA-32 System Environment –** Operating system privileged environment as defined by the *IA-32 Intel® Architecture Software Developer's Manual*. Resources include virtual paging, control registers, debugging, performance monitoring, machine checks, and the set of privileged instructions.
- **Platform –** Application and operating system resources external to the processor such as: memory maps, external devices (e.g. DMA), keyboard controllers, buses (e.g. PCI), option cards, interrupt controllers, bridges, etc.
- **Itanium Architecture-based Firmware –** The Processor Abstraction Layer (PAL) and System Abstraction Layer (SAL).
- **Processor Abstraction Layer (PAL) –** The firmware layer which abstracts processor features that are implementation dependent.
- **System Abstraction Layer (SAL) –** The firmware layer which abstracts platform features that are implementation dependent.

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 application architectural state and the rules for execution sequencing. See Chapter 6 for details on IA-32 instruction set execution.

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 predication and branching, PR0 - PR63.
- **Branch Registers (BRs)** – Registers used in branching, BR0 - BR7.
- **Instruction Pointer (IP)** – Register which holds the bundle address of the currently executing 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 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 features.

IA-32 application register state is entirely contained within the larger Itanium application register set and is accessible by Itanium instructions. IA-32 instructions cannot access the Itanium register set. See Section 6.2, “IA-32 Application Register State Model” for details on IA-32 register assignments.

3.1.1 Reserved and Ignored Registers and Fields

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

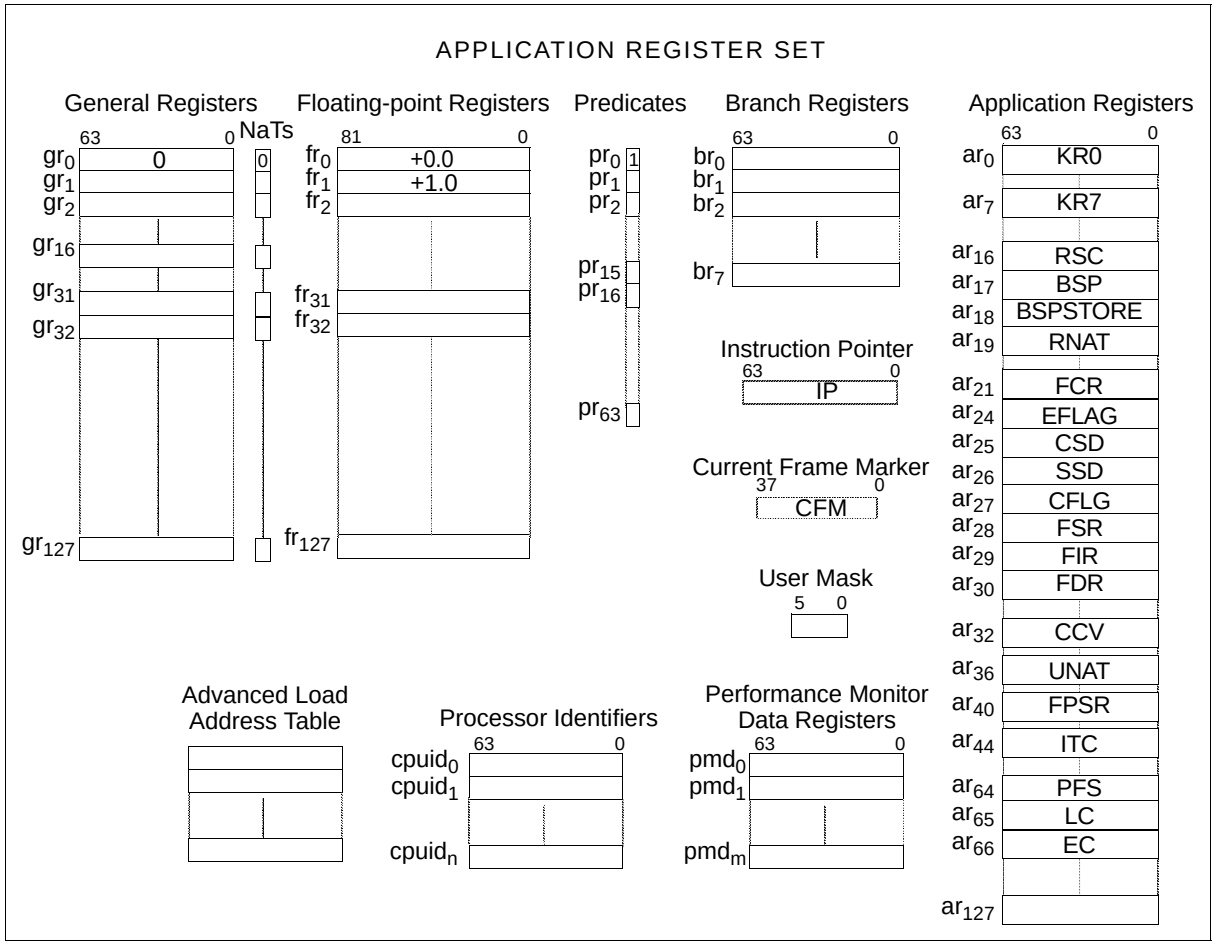
Type	Read	Write
Reserved register	Illegal Operation fault	Illegal Operation fault
Ignored register	0	Value written is discarded
Reserved field	0	Write of non-zero causes Reserved Reg/Field fault
Ignored field	0 (unless noted otherwise)	Value written is discarded

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 state 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.

Figure 3-1. Application Register Model



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 “Register Stack” on page 1:43 for a description. A portion of the stacked registers can be programmatically renamed to accelerate loops. See “Modulo-scheduled Loop Support” on page 1:70..

General registers 8 through 31 contain the IA-32 integer, segment selector and segment descriptor registers. See “IA-32 General Purpose Registers” on page 1:114 for details on IA-32 register assignments.

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 1:70..

Floating-point registers 8 through 31 contain the IA-32 floating-point and multi-media registers when executing IA-32 instructions. For details, See “IA-32 Floating-point Registers” on page 1:122..

3.1.4 Predicate Registers

A set of 64 (1-bit) **predicate registers** are used to hold the results of 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 1:50.

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 1:70..

3.1.5 Branch Registers

A set of 8 (64-bit) **branch registers** are used to hold 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 1:69.

3.1.6 Instruction Pointer

The Instruction Pointer (IP) holds the address of the bundle which contains the current executing 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 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 1:34.. 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. See “IA-32 Instruction Pointer” on page 1:115. for IA-32 instruction set execution details.

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 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 1:43).

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 3.1.8.11, “Previous Function State (PFS – AR 64)”). 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 1:70.

Figure 3-2. Frame Marker Format

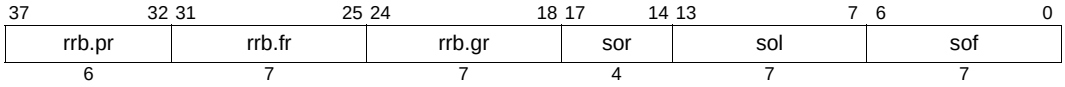


Table 3-2. Frame Marker Field Description

Field	Bits	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 Itanium instruction set architectures. These registers can be accessed by Itanium architecture-based applications (except where noted). Table 3-3 contains a list of the application registers.

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 / Compare and Store Data register	
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. See Section 10.3.2, “IA-32 System EFLAG Register” on page 2:235 for details.

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.

See Section 10.2, “System Register Model” on page 2:231 for the field definition of each IA-32 application register.

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

Eight user-visible 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. See Section 10.1, “Instruction Set Transitions” on page 2:231 for register details when calling IA-32 code.

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 Register Stack Engine (RSE). Refer to Chapter 6, “Register Stack Engine” in Volume 2 for details. 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

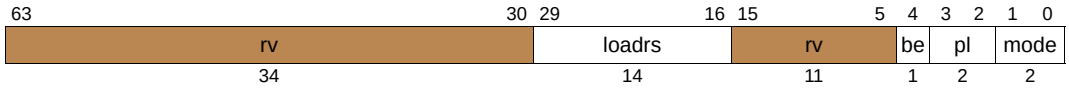


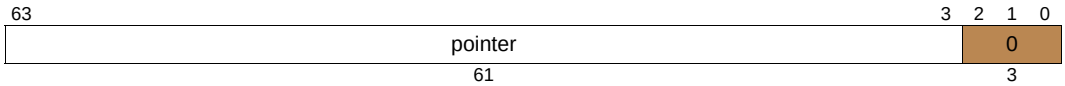
Table 3-4. RSC Field Description

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

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. See Section 6.1, “RSE and Backing Store Overview” on page 2:129.

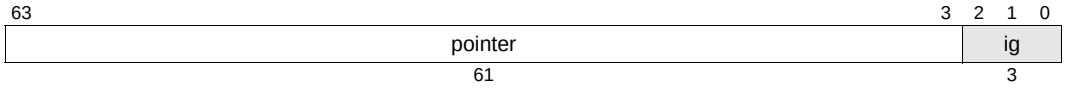
Figure 3-4. BSP Register Format



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. See Section 6.1, “RSE and Backing Store Overview” on page 2:129.

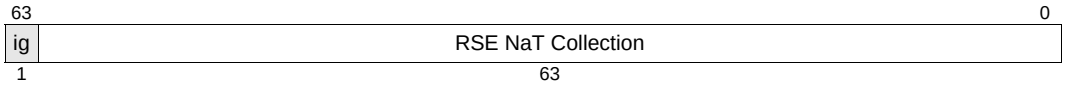
Figure 3-5. BSPSTORE Register Format



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. See Section 6.1, “RSE and Backing Store Overview” on page 2:129.

Figure 3-6. RNAT Register Format



3.1.8.6 Compare and Store Data register (CSD – AR 25)

The Compare and Store Data register is a 64-bit register that provides data to be stored by the Itanium `st16` and `cmp8xchg16` instructions, and receives data loaded by the Itanium `ld16` instruction.

For implementations that do not support the `ld16`, `st16` and `cmp8xchg16` instructions, bits 61:60 may be optionally implemented. This means that on move application register instructions the implementation can either ignore writes and return zero on reads, or write the value and return the last value written on reads. For implementations that do support the `ld16`, `st16` and `cmp8xchg16` instructions, all bits of CSD are implemented.

For IA-32 execution, this register is the IA-32 Code Segment Descriptor. See Section 6.2.3, “IA-32 Segment Registers” on page 1:115.

3.1.8.7 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 Itanium `cmpxchg` instruction.

3.1.8.8 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 `ld8.fill` and `st8.spill` instructions.

3.1.8.9 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 Itanium floating-point instructions. FPSR does not control or reflect the status of IA-32 floating-point instructions. For more details on the FPSR, see "Floating-point Status Register" on page 1:84.

3.1.8.10 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 input clock to the processor. The ITC may be clocked at a somewhat lower frequency than the instruction execution frequency. This clocking relationship is described in the PAL procedure PAL_FREQ_RATIOS on page 2:381. The ITC is guaranteed to be clocked at a constant rate, even if the instruction execution frequency may vary.

A sequence of reads of the ITC is guaranteed to return ever-increasing values (except for the case of the counter wrapping back to 0) corresponding to the program order of the reads. Applications can directly sample the ITC for time-based calculations.

System software can secure the interval time counter from non-privileged 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 similar to ITC counter. 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).

3.1.8.11 Previous Function State (PFS – AR 64)

The 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 a `br.call` or `brl.call` is executed, the CFM, EC, and PSR.cpl are copied to the PFS and the old contents of the PFS are discarded. When a `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. For more details on the PSR see Chapter 3, "System State and Programming Model" in Volume 2.

The PFS.pfm has the same layout as the CFM (see Section 3.1.7, "Current Frame Marker"), and the PFS.pec has the same layout as the EC (see Section 3.1.8.13, "Epilog Count Register (EC – AR 66)").

Figure 3-7. PFS Format

Table 3-5. PFS Field Description

Field	Bits	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

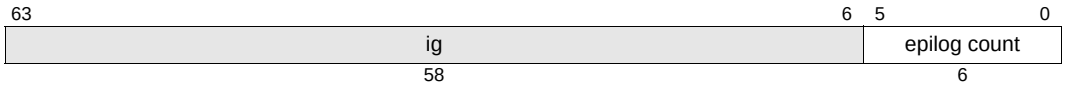
3.1.8.12 Loop Count Register (LC – AR 65)

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

3.1.8.13 Epilog Count Register (EC – AR 66)

The Epilog Count register (EC) is a 6-bit register used for counting the final (epilog) stages in modulo-scheduled loops. See “Modulo-scheduled Loop Support” on page 1:70.. 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. Refer to Chapter 7, “Debugging and Performance Monitoring” in Volume 2 for details. Performance monitors can be used to gather performance information for the execution of both IA-32 and Itanium instruction sets.

3.1.10 User Mask (UM)

The user mask is a subset of the Processor Status Register and is accessible to application programs. The user mask controls memory access alignment, byte-ordering and user-configured performance monitors. It also records the modification state of floating-point registers. Figure 3-9 show the user mask format and Table 3-6 describes the user mask fields. For more details on the PSR refer to “Processor Status Register (PSR)” on page 2:22.

Figure 3-9. User Mask Format

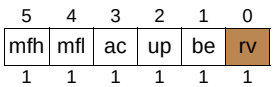


Table 3-6. User Mask Field Descriptions

Field	Bit	Description
rv	0	Reserved
be	1	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 (including IA-32) 0: user performance monitors are disabled 1: user performance monitors are enabled
ac	3	Alignment check for data memory references (including IA-32) 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 Intel® Itanium® 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. See Section 3.3.2, “Processor Status Register (PSR)” for conditions when IA-32 instructions set this bit.
mfh	5	Upper (f32.. f127) floating-point registers written – This bit is set to one when an Intel® Itanium® 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. See Section 3.3.2, “Processor Status Register (PSR)” for conditions when IA-32 instructions set this bit.

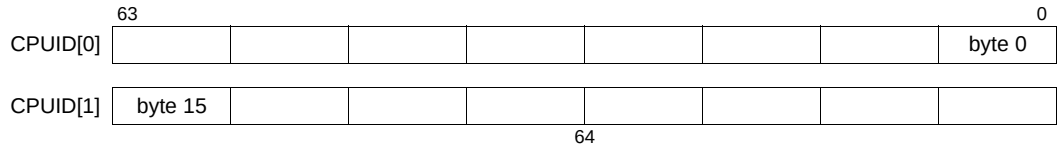
3.1.11 Processor Identification Registers

Application level processor identification information is available in a 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



CPUID register 2 is an ignored register (reads from this register return zero).

CPUID register 3 contains several fields indicating version information related to the processor implementation. Figure 3-11 and Table 3-7 specify the definitions of each field.

Figure 3-11. CPUID Register 3 – Version Information

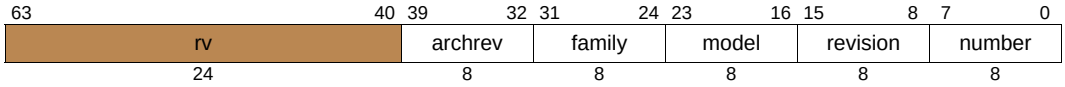


Table 3-7. CPUID Register 3 Fields

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 processor features. As shown in Figure 3-12, it is a set of flag bits used to indicate if a given feature is supported in the processor model. When a bit is one the feature is supported; when 0 the feature is not supported. The defined feature bits in the current architecture are listed in Table 3-8. 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.

CPUID register 4 is logically split into two halves, both of which contain general feature and capability information but which have different usage models and access capabilities; this information reflects the status of any enabled or disabled features. Both the upper and lower halves of CPUID register 4 are accessible through the move indirect register instruction; depending on the implementation, the latency for this access can be long and this access method is not appropriate for low-latency code versioning using self-selection. In addition, the upper half of CPUID register 4 is also accessible using the test feature instruction; the latency for this access is comparable to that of the test bit instruction and this access method enables low-latency code versioning using self selection.

This register does not contain IA-32 instruction set features. IA-32 instruction set features can be acquired by the IA-32 cpuid instruction.

Figure 3-12. CPUID Register 4 – General Features/Capability Bits

Table 3-8. CPUID Register 4 Fields

Field	Bits	Description
lb	0	Processor implements the long branch (brl) instructions.
sd	1	Processor implements spontaneous deferral (see Section 5.5.5, “Deferral of Speculative Load Faults” on page 2:100).
ao	2	Processor implements 16-byte atomic operations (see “ld — Load”, “st — Store” and “cmpxchg — Compare and Exchange” instructions in Volume 3).
rv	63:3	Reserved.

3.2 Memory

This section describes an Itanium architecture-based 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.

The system view of memory and of virtual memory management is given in Chapter 4, "Addressing and Protection" in Volume 2. The IA-32 instruction set view of memory and virtual memory management is defined in Section 10.6, "System Memory Model" on page 2:252.

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. For details on 32-bit addressing, refer to "32-bit Virtual Addressing" on page 2:67.

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 (three 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 Itanium architecture-based code. 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. 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 3-13 shows various loads in little-endian format. Figure 3-14 shows various loads in big endian format. Stores are not shown but behave similarly.

Figure 3-13. Little-endian Loads

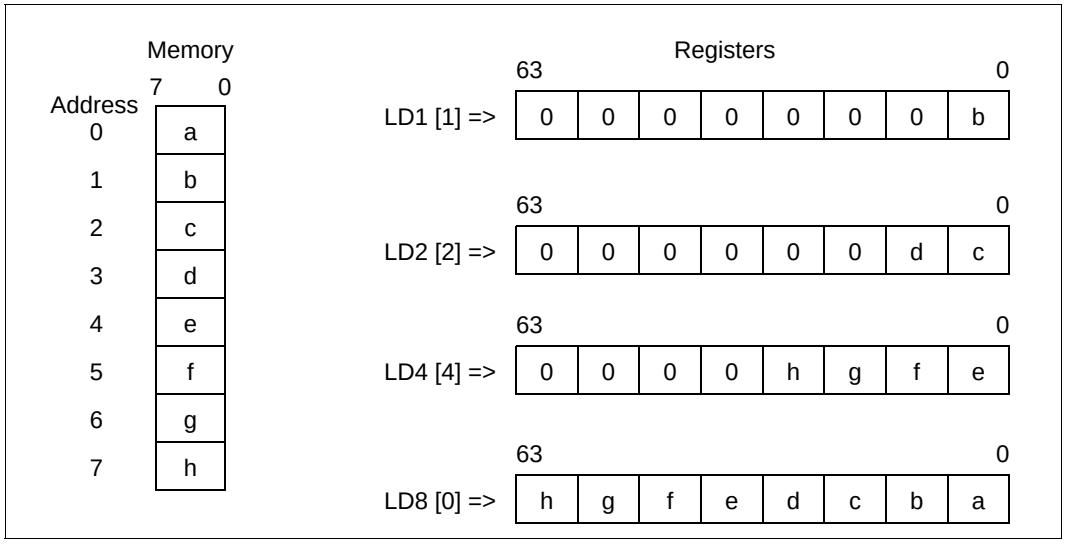
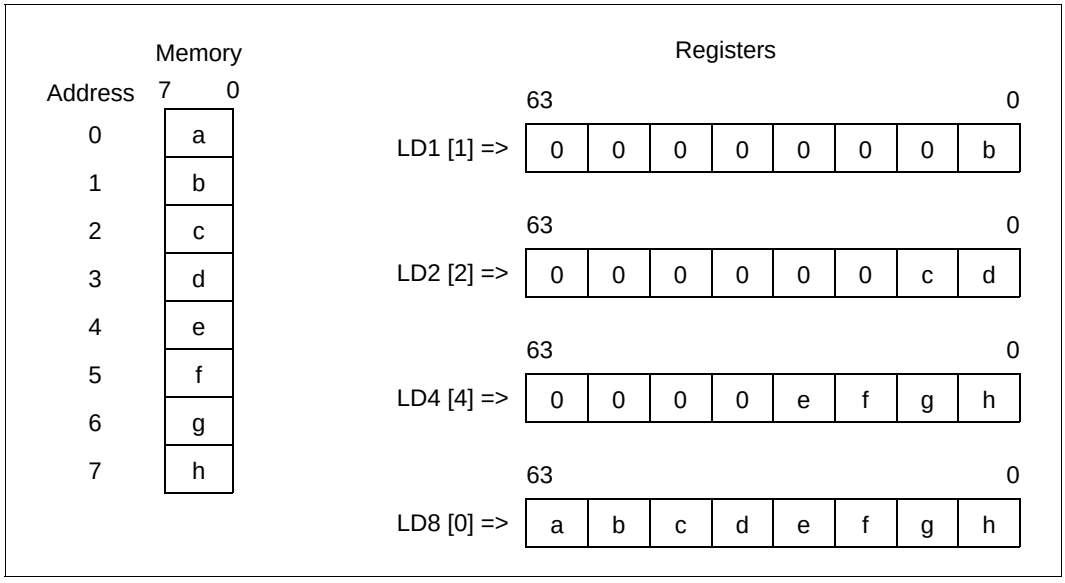


Figure 3-14. Big-endian Loads



3.3 Instruction Encoding Overview

Each instruction is categorized into one of six types; each instruction type may be executed on one or more execution unit types. Table 3-9 lists the instruction types and the execution unit type on which they are executed.

Table 3-9. Relationship between Instruction Type and Execution Unit Type

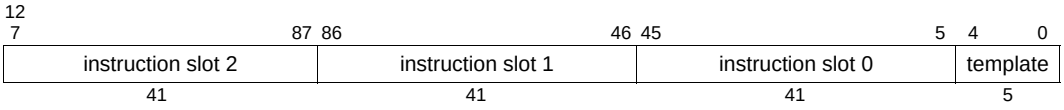
Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit

Table 3-9. Relationship between Instruction Type and Execution Unit Type (Continued)

Instruction Type	Description	Execution Unit Type
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit/B-unit

Three instructions are grouped together into 128-bit sized and aligned containers called **bundles**. Each bundle contains three 41-bit **instruction slots** and a 5-bit template field. The format of a bundle is depicted in Figure 3-15.

Figure 3-15. 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 3-10. 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 3-10 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 3-10. Template Field Encoding and Instruction Slot Mapping

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	L-unit	X-unit[a]
05	M-unit	L-unit	X-unit[a]
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit

Table 3-10. Template Field Encoding and Instruction Slot Mapping (Continued)

Template	Slot 0	Slot 1	Slot 2
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

a. The MLX template was formerly called MLI, and for compatibility, the X slot may encode break.i and nop.i in addition to any X-unit instruction.

Extended instructions, used for long immediate integer and long branch instructions, occupy two instruction slots. Depending on the major opcode, extended instructions execute on a B-unit (long branch/call) or an I-unit (all other L+X instructions).

3.4 Instruction Sequencing Considerations

Itanium architecture-based code 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 3-15 on page 1:35.

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, taken branch, Break Instruction fault due to a `break.b`, or Illegal Operation fault due to a Reserved or Reserved if PR[qp] is one encoding in the B-type opcode space. For the instructions in an instruction group to have well-defined behavior, they must meet the ordering and dependency requirements described below.

For the purpose of clarification, the following do not end instruction groups:

- Break instructions other than `break.b` (`break.f`, `break.i`, `break.m`, `break.x`)
- Check instructions (`chk.s`, `chk.a`, `fchkf`)
- `rfi` instructions not followed by a stop
- `brl` instructions not followed by a stop
- Interruptions other than a Break Instruction fault due to a `break.b` or an Illegal Operation fault due to a Reserved or Reserved if PR[qp] is 1 encoding in the B-type opcode space

Thus, even if one of the above causes a change in control flow, the instructions at sequentially increasing addresses beyond the location of the change in control flow up to the next true end of the instruction group had the change of control flow not occurred, can still cause undefined values to be seen at the target of the change of control flow, if they cause a dependency violation. There are never, however, any dependencies between the instructions at the target of the change in control flow and those preceding the change in control flow, even for the above cases.

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 dynamically succeeding instructions and the *update* of all dynamically 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. See Section 3.2, “Serialization” on page 2:17 for details.
- Within an instruction group, every instruction will behave as though its read of the memory and Advanced Load Address Table (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 Dependency Special Cases” on page 1:39 and “WAW Dependency Special Cases” on page 1:39). Write-after-read (WAR) register dependencies are allowed (except as noted in “WAR Dependency Special Cases” on page 1:40).

  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).

Some system state updates require more stringent requirements than those described here. See Section 3.2, “Serialization” on page 2:17 for details.

- 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 (see “Undefined Behavior” on page 1:41).

Exceptions are reported in instruction order. The dependency restrictions apply independent of the presence or absence of exceptions — that is, restrictions must be satisfied whether or not an exception occurs within an instruction group. At the point of exception delivery for a correctly formed instruction group, all prior instructions will have completed their update of architectural state. All subsequent instructions will not have updated architectural state. If an instruction group violates a dependency requirement, then the update of architectural state before and after an exception is not guaranteed (the fault handler sees an undefined value on the registers involved in a dependency violation even if the exception occurs between the first and second instructions in the violation). In the event multiple exceptions occur while executing instructions from the same instruction group, the exception occurring on the earliest instruction will be reported.

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.

For more details on resource dependencies, see Chapter 5, “Resource and Dependency Semantics” in Volume 3.

3.4.1 RAW Dependency Special Cases

There are four special cases in which RAW register dependencies within an instruction group are permitted. These special cases 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 dependencies on resources other than CFM affected by `alloc` and are thus not allowed in the same instruction group after an `alloc`: `flushrs`, `loadrs`, move from AR[BSPSTORE], move from AR[RNAT], `br.cexit`, `br.ctop`, `br.wexit`, `br.wtop`, `br.call`, `brl.call`, `br.ia`, `br.ret`, `clrrrb`, `cover`, and `rfi`. See Chapter 5, "Resource and Dependency Semantics" in Volume 3 for details. 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`, `brl.cond`, `brl.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. An indirect `brp` is considered a reader of the specified BR.

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.

3.4.2 WAW Dependency Special Cases

There are three special cases in which WAW register dependencies within an instruction group are permitted. The special cases 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 PR0. All WAW dependencies for PR0 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 WAW register dependencies with move to PR instructions that access the same predicate registers as another writer.

Note: The move to PR instructions only writes those PRs indicated by its mask, but the move from PR instructions 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.

3.4.3 WAR Dependency Special Cases

The WAR dependency between the reading of predicate register 63 by any B-type instruction and the subsequent writing of predicate register 63 by a modulo-scheduled loop type branch (`br.ctop`, `br.cexit`, `br.wtop`, or `br.wexit`) without an intervening stop is not allowed. Otherwise, WAR dependencies within an instruction group are allowed.

3.4.4 Processor Behavior on Dependency Violations

If a program violates read-after-write, write-after-write or write-after-read resource dependency rules within an instruction group, then processor behavior is undefined. Constraints on undefined behavior are described in "Undefined Behavior" on page 1:41.

To help debug code that violates the architectural resource dependency rules, some processor implementations may provide dependency violation detection hardware that may cause an instruction group that contains an illegal dependency to take an Illegal Dependency fault (defined in Chapter 5, "Interruptions" in Volume 2). However, even in implementations that provide such checking, software can not assume the processor will catch all dependency violations or even catch the same violation every time it occurs.

However, all processor models that provide dependency violation detection hardware are required to satisfy the following dependency violation reporting constraints:

- All detected dependency violations must be reported as Illegal Dependency Faults (defined in Chapter 5, “Interruptions” in Volume 2). When an Illegal Dependency fault is taken, the value of the resource subject to the dependency violation is undefined. Undetected dependency violations cause undefined program behavior as described in “Undefined Behavior” on page 1:41.
- All detected read-after-write and write-after-write dependency violations must be delivered as Illegal Dependency Faults on the second operation, i.e. on the reader in the RAW case, and on second resource writer in the WAW case.
- All detected write-after-read dependency violations (on predicate register 63) must be delivered as Illegal Dependency faults on the second operation, the predicate writer.
- Illegal Dependency faults are delivered strictly in program order. If an interruption, branch or speculation check are taken between the first and the second operation of a dependency violation, then the Illegal Dependency fault is not taken.

Note: Since an instruction group starts at a given entry point (stop or target of a control flow transfer), instructions that precede the entry point are not considered part of the instruction group and must not take part in any dependency violation checking. For example, if an `rfi` is done to slot 1 of a bundle, the instruction in slot 0 and instructions in bundles with lower memory addresses are not part of the new instruction group, and must not take part in any dependency violation checking.

3.5 Undefined Behavior

Architecturally undefined behavior that applies to one or more instructions is listed below:

- RAW and WAW register dependencies within the same instruction group are disallowed except as noted in Section 3.4, “Instruction Sequencing Considerations” on page 1:36. Their behavior within an instruction group is undefined. Undefined behavior includes the possibility of an Illegal Operation fault.
- Reading a register outside of the defined general register stack frame boundaries (as determined by the most recent `alloc`, return, or call) will return an undefined result. All processors will not raise an interruption in this situation.

An undefined scenario is an event or sequence of events whose outcome is not defined in the architecture. For the behavior of Itanium instructions, refer to Chapter 2, “Instruction Reference” in Volume 3. For the behavior of IA32 instructions, refer to Volume 3, Part II:, “IA-32 Instruction Set Descriptions.” Therefore, the result of an undefined scenario is strictly implementation dependent. User should not rely on these undefined behaviors for correct program behavior and compatibility across future implementations.

Application Programming Model 4

This section describes the architectural functionality from the perspective of the application programmer. Itanium 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, “Floating-point Programming Model” in Volume 1. Refer to *Volume 3: Instruction Set Reference* for detailed information on Itanium instructions.

The main features of the 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
- Privilege Level Transfer

4.1 Register Stack

As described in “General Registers” on page 1:23, 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 (for details on the RSE see Chapter 6, “Register Stack Engine” in Volume 2). 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` uses three immediate operands to determine the values of sol and sof: the size of inputs; the size of locals; and the size of outputs. 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 `br.call` or `brl.call` 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)

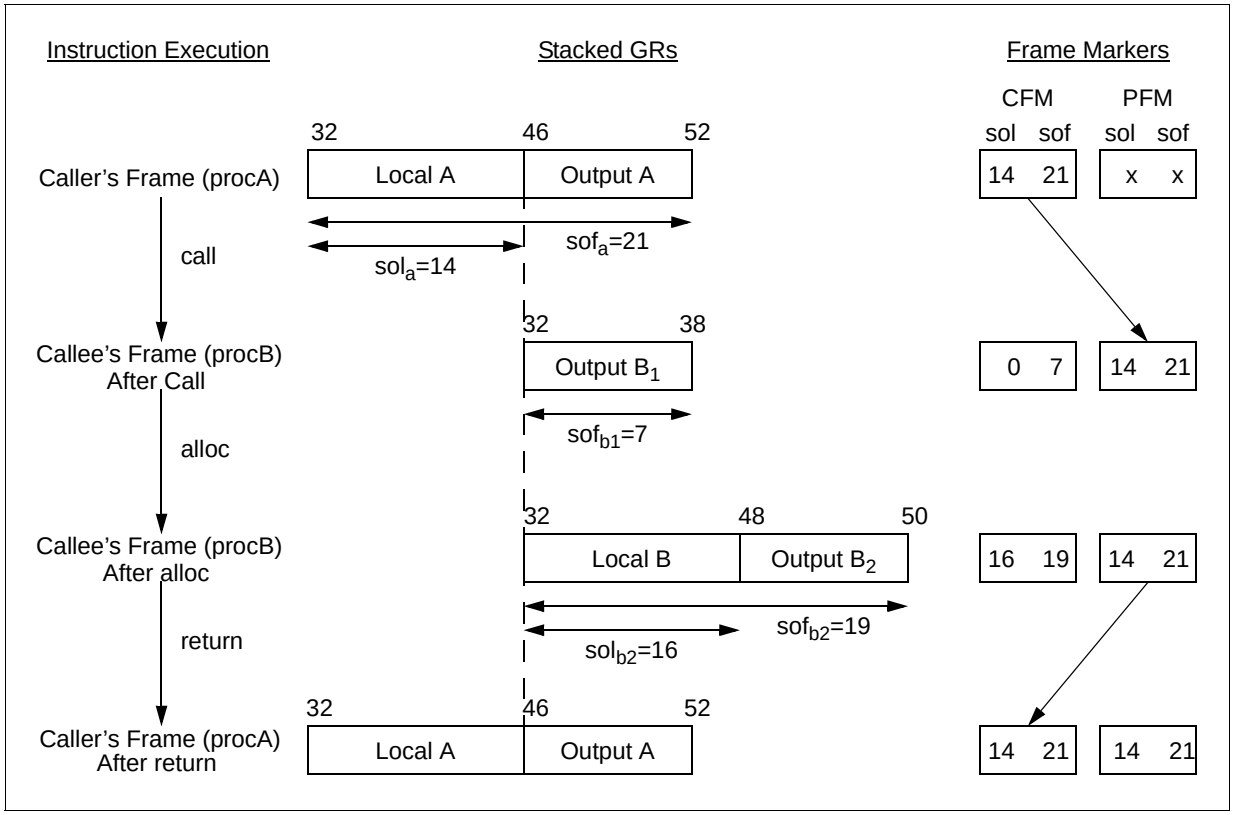
Values in the output area of the caller's register stack frame are visible to the callee. This overlap permits parameter and return value passing between procedures to take place entirely in registers.

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 `br.ret` 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`, `cover`, 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 Register Stack Engine (RSE) is required only by certain specialized application software such as user-level thread packages, debuggers, etc. See Chapter 6, "Register Stack Engine" in Volume 2.

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. If the qualifying predicate for `alloc` is not PR0, an Illegal Operation fault is raised. 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. See Chapter 6, "Register Stack Engine" in Volume 2 for details on explicit management of the RSE.

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.

The `cover` instruction creates a new frame of zero size (sof = sol = 0). The new frame is created above (not overlapping) the present frame. Both the local and output areas of the previous stack frame are automatically saved. A `cover` instruction must be the last instruction in an instruction group; otherwise, operation is undefined. A `cover` cannot be predicated.

The `loadrs` instruction ensures that the specified portion of the register stack is present in the physical registers. It stalls instruction execution until the number of bytes specified in the loadrs field of the RSC application register have been filled from the backing store by the RSE (starting from the current BSP). By specifying a zero value for RSC.loadrs, `loadrs` can be used to indicate that all stacked registers outside the current frame must be loaded from the backing store before being used. In addition, stacked registers outside the current frame (that have not been spilled by the RSE) will not be stored to the backing store. A `loadrs` instruction must be the first instruction in an instruction group otherwise the results are undefined. A `loadrs` 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 1:69.

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
`loadrs`	Load register stack from backing store
`cover`	Cover current stack frame

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.

The immediate form of addition uses a register and a 14-bit immediate; the immediate form of subtraction uses a register and an 8-bit immediate. In both cases, the immediate is sign-extended before being added or subtracted. 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 1:97 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 32-bit addresses is provided in the form of add instructions that perform region bit copying. This supports the virtual address translation model (see “32-bit Virtual Addressing” on page 2:67 for details). 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 1:49. 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 1:49.

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 instruction 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 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 instructions cannot be predicated. These instructions are: allocate stack frame (`alloc`), branch predict (`brp`), bank switch (`bsw`), clear rrb (`clrrrb`), cover stack frame (`cover`), enter privileged code (`epc`), flush register stack (`flushrs`), load register stack (`loadrs`), counted branches (`br.cloop`, `br.ctop`, `br.cexit`), and return from interruption (`rfi`).

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`, `fprcpa`, `frsqrta`, `fprsqrta`). Most of these compare instructions (all but `frcpa`, `fprcpa`, `frsqrta` and `fprsqrta`) 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 1:51. 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`, `fprcpa`, `frsqrta` and `fprsqrta` 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 1:56), 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. Please see the processor-specific documentation for further information. 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. All IA-32 memory references are performed little-endian.

Load, store and semaphore instructions are summarized in Table 4-12 and the state related to memory reference instructions is summarized in Table 4-13.

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
ld.fill	ldf.fill		Register Fill
st	stf		Store
st.rel			Ordered store
st.spill	stf.spill		Register Spill
cmpxchg			Compare and exchange
xchg			Exchange memory and GR
fetchadd			Fetch and add

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
CSD	Compare and Store Data application register

4.4.1 Load Instructions

Load instructions transfer data from memory to a general register, a general register and the Compare and Store Data (CSD) register, a floating-point register or a pair of floating-point registers.

For general register loads, access sizes of 1, 2, 4, 8, and 16 bytes are defined. For sizes less than eight bytes, the loaded value is zero extended to 64-bits. The 16-byte general-register load instructions load two adjacent 8-byte quantities into a general register and the CSD register. The 16-byte general-register load instructions cannot specify base register update.

For floating-point loads, the following access sizes are defined: single precision (4 bytes), double precision (8 bytes), double-extended precision (10 bytes), and integer/parallel FP (8 bytes). The value(s) loaded from memory are converted into floating-point register format (see “Memory Access Instructions” on page 1:87 for details).

The floating-point load pair instructions load two adjacent single precision (4 bytes each), double precision (8 bytes each), or integer/parallel FP (8 bytes each) numbers into two independent floating-point registers (see the `ldfp` instruction description for restrictions on target register specifiers). Floating-point load pair instructions can specify base register update, but only by an immediate value equal to double the data size.

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 1:56 and “Data Speculation” on page 1:59.

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 1:68. A biased load provides a hint to acquire exclusive ownership of the accessed line. See “Memory Hierarchy Control and Consistency” on page 1:64.

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 1:58.

4.4.2 Store Instructions

Store instructions transfer data from a general register, a general register and the CSD register, 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. The 16-byte general-register store instructions store two adjacent 8-byte quantities from a general register and the CSD register.

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 1:58.

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 `cmp8xchg16` instruction loads the target with 8 bytes from the memory location addressed by the first source; if this value is equal to the contents of the CCV register, then the second source and the CSD register are both stored into memory at the 16-byte-aligned address which contains the memory location loaded.

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.

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 1:59.

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 1:82 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 1:84.) 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 1:52). 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 1:50.

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 Operating System Control over Exception Deferral

An additional mechanism is defined that allows the operating system to control the exception behavior of speculative loads. The operating system has the option to select which exceptions are deferred automatically in hardware and which exceptions will be handled (and possibly deferred) by software. See Section 5.5.5, “Deferral of Speculative Load Faults” on page 2:100.

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

4.4.5.1 Data Speculation Concepts

An ambiguous memory dependency 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`
`st8 [r4] = r12`
`ld8 r6 = [r8];;`
`add r5 = r6, r7;;`
`st8 [r18] = r5`
	`ld8.a r6 = [r8];; // Advanced load`
	`// Other instructions`
	`st8 [r4] = r12`
	`ld8.c.clr r6 = [r8] // Check load`
	`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`
`st8 [r4] = r12`
`ld8 r6 = [r8];;`
`add r5 = r6, r7;;`
`st8 [r18] = r5`
	`ld8.a r6 = [r8];;`
	`// Other instructions`
	`add r5 = r6, r7;;`
	`// Other instructions`
	`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 wraparound 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 wraparounds.

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, there is flexibility in the actions that a processor can perform:

1. The implementation may choose to either leave the target register unchanged or to reload the value from memory.
2. If the implementation chooses to leave the target register unchanged and one or more exception conditions related to the data access or translation of the check load occurs, the implementation may choose to either raise the highest-priority of these faults or ignore them all and continue execution. The faults that can be ignored are those related to data access and translation (Data Nested TLB fault, Alternate Data TLB fault, VHPT Data fault, Data TLB fault, Data Page Not Present fault, Data NaT Page Consumption fault, Data Key Miss fault, Data Key Permission fault, Data Access Rights fault, Data Dirty Bit fault, Data Access Bit fault, Data Debug fault, Unaligned Data Reference fault, Unsupported Data Reference fault). See Table 5-6, “Interruption Priorities” on page 2:104.

3. If the implementation chooses to perform a reload, then any faults that occur because of the reload can not be ignored.
4. If the size, type, or address fields in the matching ALAT entry do not match that provided by a check load, the value returned by the check load is undefined. In such cases the implementation may choose to raise a fault or when the “no clear” variant of the check load is issued, an implementation may choose to update the address, size, or type fields of the matching ALAT entry or to leave the entry 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 1:68.

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, semaphores, or `ptc.ga` on other processors in the coherence domain.
2. The execution of store or semaphore instructions issued on the local processor.
3. Platform-visible removal of a cache line from the processor’s caches.

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. Note that some invalidation events may require that multiple entries be removed from the ALAT. For example, the `ptc.ga` instruction is page aligned, thus a `ptc.ga` from another processor would require that hardware invalidate all ALAT entries related to that page. Stores due to RSE spills are not checked for ALAT invalidation and do not cause ALAT entries to be removed. See Section 6.9, “RSE and ALAT Interaction” on page 2:141. When an external agent can observe that the processor has removed a physical address range from its caches, then that address range is guaranteed to be invalidated from that processor’s ALAT as well.

An implementation may invalidate entries over areas larger than explicitly required by a specific invalidation event, and more generally, to invalidate any ALAT entry at any time. For example, a `st1` only accesses one byte, but an implementation could choose to invalidate all ALAT entries whose memory region is in the same cache line. An implementation may also provide an ALAT with zero entries (i.e., all `ld.c`/`chk.a` instructions would act as if an ALAT miss had occurred).

Software is responsible for explicitly invalidating all affected ALAT entries whenever:

1. Software explicitly changes the virtual to physical register mapping on rotating registers that have been the target of advanced loads (`clrrrb`).

2. Software changes the virtual to physical memory mapping.
3. Software accesses the RSE backing store with advanced loads. See Section 6.9, “RSE and ALAT Interaction” on page 2:141 (since RSE stores do not invalidate ALAT entries).
4. Software explicitly changes the virtual to physical register mapping on stacked registers by switching the RSE backing stores. See Section 6.11.3, “Synchronous Backing Store Switch” on page 2:143.

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

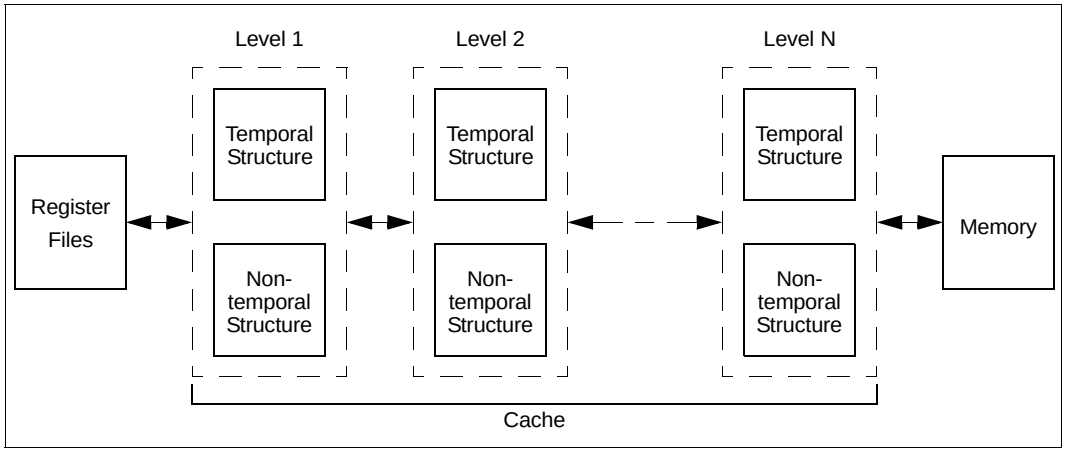
Mnemonic	Operation
`ld.a`, `ldf.a`, `ldfp.a`	GR and FR advanced load
`st`, `st.rel`, `st.spill`, `stf`, `stf.spill`	GR and FR store
`cmpxchg`, `fetchadd`, `xchg`	GR semaphore
`ld.c.clr`, `ld.c.clr.acq`, `ldf.c.clr`, `ldfp.c.clr`	GR and FR check load, clear on ALAT hit
`ld.c.nc`, `ldf.c.nc`, `ldfp.c.nc`	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

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. Please see the processor-specific documentation for further information.

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.

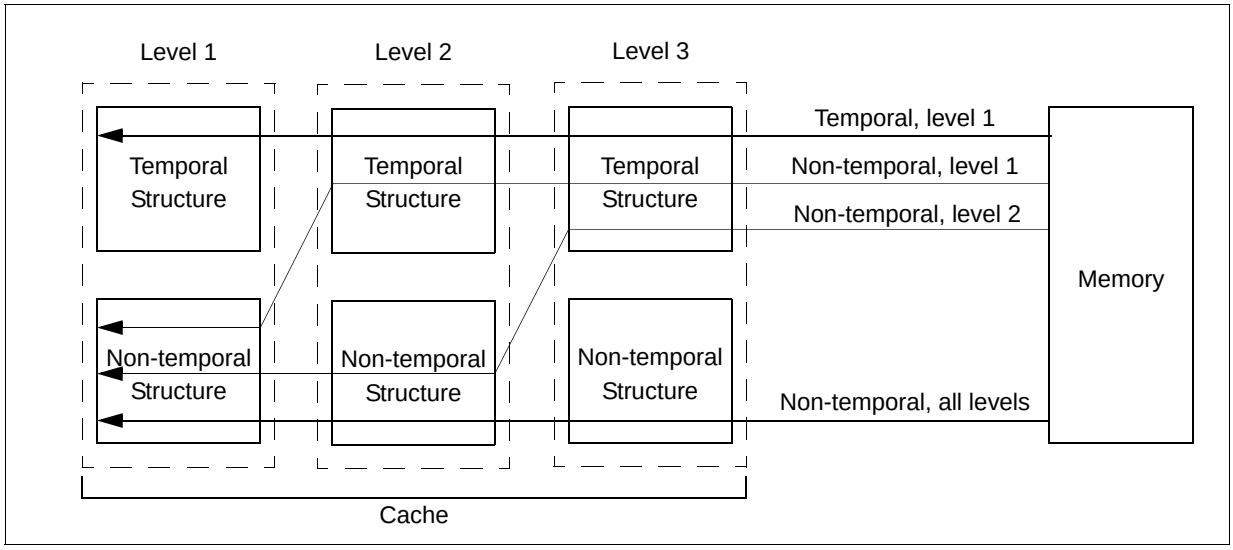
Table 4-18. Locality Hints Specified by Each Instruction Class

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. Hinting that a datum should not be cached in a temporal structure indicates that it is not likely to be read in the near future. For stores, the `.nta` completer also hints that the store may be part of a set of streaming stores that would likely overwrite the entire cache line without any data in that line first being read, enabling the processor to avoid fetching the data.

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 following instructions are defined for flush control: flush cache (`fc`, `fc.i`) and flush write buffers (`fwb`). 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. The `fc.i` instruction ensures the data cache line associated with an address is coherent with the instruction caches. The `fc.i` instruction is not required to invalidate the targeted cache line nor write the targeted cache line back to memory if it is inconsistent with memory, but

may do so if this is required to make the targeted cache line coherent with the instruction caches. The `fwb` instruction provides a hint to flush all pending buffered writes to memory (no indication of completion occurs).

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, fc.i`	Flush cache
`fwb`	Flush write buffers

4.4.6.2 Memory Consistency

In the Itanium architecture, 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 made coherent with instruction caches. Software may use the `fc.i` 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 3:236 in *Volume 3: Instruction Set Reference* 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. For definitions of the ordering rules related to non-cacheable memory, cache synchronization, and privileged instructions, refer to Section 4.4.7, “Sequentiality Attribute and Ordering” on page 2:77.

Table 4-21. Memory Ordering Instructions

Mnemonic	Operation
`ld.acq, ld.c.clr.acq`	Ordered load and ordered check load
`st.rel`	Ordered store

Table 4-21. Memory Ordering Instructions (Continued)

`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 three categories of branches: IP-relative branches, long 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. Long branches are IP-relative with a 60-bit displacement, allowing the target to be anywhere in the 64-bit address space. Because of the long immediate, long branches occupy two instruction slots. 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.cond` or `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 1:43). 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. The long branches, `brl.cond` or `brl`, and `brl.call` are identical to `br.cond` or `br`, and `br.call`, respectively, except for their longer displacement.

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

Table 4-22. Branch Types (Continued)

Mnemonic	Function	Branch Condition	Target Address
`brl.cond or brl`	Long conditional branch	Qualifying predicate	IP-rel
`brl.call`	Long conditional procedure call	Qualifying predicate	IP-rel

The counted loop type (`br.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 (`br.ctop`, `br.cexit`, `br.wtop`, `br.wexit`) are described in "Modulo-scheduled Loop Support" on page 1:70. The loop type branches (`br.cloop`, `br.ctop`, `br.cexit`, `br.wtop`, `br.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 and Table 4-24 summarize 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
`brl`	Long branch
`brp`	Provide early hint information about a future 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 three 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 wraparound 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 on the value of a rotating register base (rrb) contained in CFM. An independent 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, and 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, brl.call`	Clears all rrb’s
`cover`	Clears all rrb’s
`br.ret`	Restores CFM.rrb’s from PFM.rrb’s
`rfi`	Restores CFM.rrb’s from IFM.rrb’s if IFM.v==1
`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 in two ways: with branch hints as part of a branch instruction (referred to as hints), and with separate Branch Predict instructions (`brp`) where the entire instruction is hint information. Hints and `brp` instructions 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). Please see the processor-specific documentation for further information.

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

4.5.3 Branch Predict Instructions

Branch predict instructions are entire instructions whose only purpose is to provide early information about future branches. Branch predict instructions provide the following pieces of information:

- **Location of the branch:** A displacement in the `brp` instruction added to the IP of the bundle containing the `brp` instruction gives the IP of the bundle containing the future branch.
- **Target of the branch:** IP-relative `brp` instructions specify the target of the future branch with a 21-bit displacement (just like in branches). The displacement plus the IP of the bundle containing the `brp` instruction gives the target address. Indirect `brp` instructions specify the branch register which will be used by the future branch.
- **Branch importance:** This hint indicates to hardware that it should employ a very fast (but small) prediction structure for this branch (useful on tight loops).

- **Whether prediction strategy:** Same as the strategy hint on branches, except that the available hints are slightly different. Static not-taken is not provided (it's not useful to provide early indication of such branches), and only one form of Dynamic prediction is provided. Instead, two strategies are included to indicate that the branch will be a "positive" (CLOOP, CTOP, WTOP) or "negative" (CEXIT, WEXIT) loop-type.

The move to branch register instruction can also provide this same hint information, simplifying the setup for a hinted indirect branch.

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 wraparound 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
`mov =psr.um`	Move from user mask to GR
`mov psr.um=`	Move from GR to user mask
`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. The move to branch register instruction can also optionally include branch hints. See “Branch Prediction Hints” on page 1:72.

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.

In addition, instructions are defined to move values between the general register file and the user mask (`mov psr.um=` and `mov =psr.um`). The `sum` and `rum` instructions set and reset the user mask. The user mask is the non-privileged subset of the Process Status Register (PSR).

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 and Bit Strings

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 Bit Strings

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

Table 4-34. Bit Support Instructions

Mnemonic	Operation
`popcnt`	Count number of ones in source register

4.9 Privilege Level Transfer

Three instructions may cause a privilege level change: break (`break`), enter privileged code (`epc`) and branch return (`br.ret`). The `break` instruction is defined to cause a Break Instruction fault which can be used to transfer privilege levels. The `break` instruction contains an immediate which is made available to a dedicated fault handler. The `epc` instruction increases the privilege level without causing an interruption or a control flow transfer. The new privilege level is specified by the TLB entry for the page containing the `epc`, if virtual address translation for instruction fetches

is enabled. If the privilege level specified by PFS.ppl (in the Previous Function State application register) is lower than the current privilege level (as specified by PSR.cpl in the Processor Status Register) `epc` raises an Illegal Operation fault. The `br.ret` instruction is defined to demote the privilege level if PFS.ppl is lower than PSR.cpl. A `br.ret` will never increase privilege level.

Floating-point Programming Model 5

The 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 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

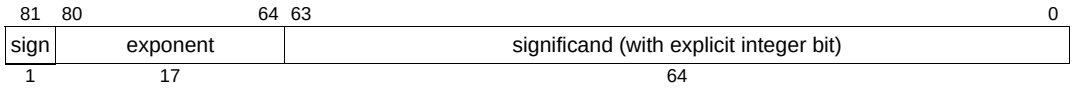
5.1.2 Floating-point Register Format

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}).
- 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).

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:

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

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

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

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.

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
IEEE Single Real Denormals	0/1	0x0FF81	0.000...01...(40)0s through 0.111...11...(40)0s
IEEE 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

Table 5-2. Floating-point Register Encodings (Continued)

Class or Subclass	Sign (1 bit)	Biased Exponent (17-bits)	Significand i.bb...bb (Explicit Integer Bit is Shown) (64-bits)
Unnormal numbers with the same value as IEEE Double-Extended Real Denormals	0/1	0x0C001	0.000...01 through 0.111...11
IEEE Double-Extended Real Denormals	0/1	0x00000	0.000...01 through 0.111...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
Double-Extended Real Pseudo-Denormals (IA-32 stack and memory format)	0/1	0x00000	1.000...00 through 1.111...11
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. Created by a masked real invalid operation.
b. Created by a masked integer invalid operation.
c. Created by an unsuccessful speculative memory operation.

All register encodings are allowed as inputs to arithmetic operations. The result of an arithmetic operation is always the most normalized register format 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 format encoding. Double-extended real denormal results are mapped to the register format 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

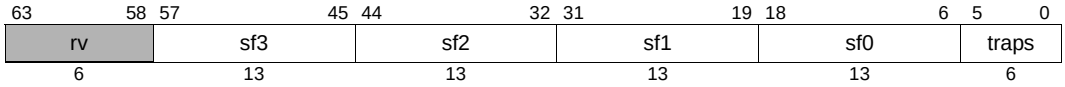


Table 5-3. Floating-point Status Register Field Description

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

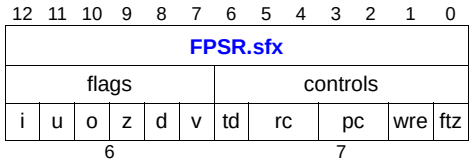


Table 5-4. Floating-point Status Register's Status Field Description

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)
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 this flag. As depicted in Table 5-2 on page 1:83, canonical single/double/double-extended denormal, double-extended pseudo-denormal and 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" (see "Definition of Tininess, Inexact and Underflow" on page 1:103) to be truncated to the correctly signed zero. Flush-to-Zero mode can be enabled only if Underflow is disabled. 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, "Floating-point Computation Model Control Definitions" on page 1:86. 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 format range, single precision
.d	ignored	1	53 bits	17 bits	Register format 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 format range, single precision
none	01	1	N.A.	N.A.	Reserved
none	10	1	53 bits	17 bits	Register format range, double precision
none	11	1	64 bits	17 bits	Register format 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 format 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., frcpa).

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 floating-point instructions. Refer to *Volume 3: Instruction Set Reference* for a detailed description.

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 Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:64.) 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. See “ldfp — Floating-point Load Pair” on page 3:147. 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, “Control Speculation”). 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, Figure 5-7, Figure 5-8 and Figure 5-9 describe how single precision, double precision, double-extended precision, integer/parallel FP, 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

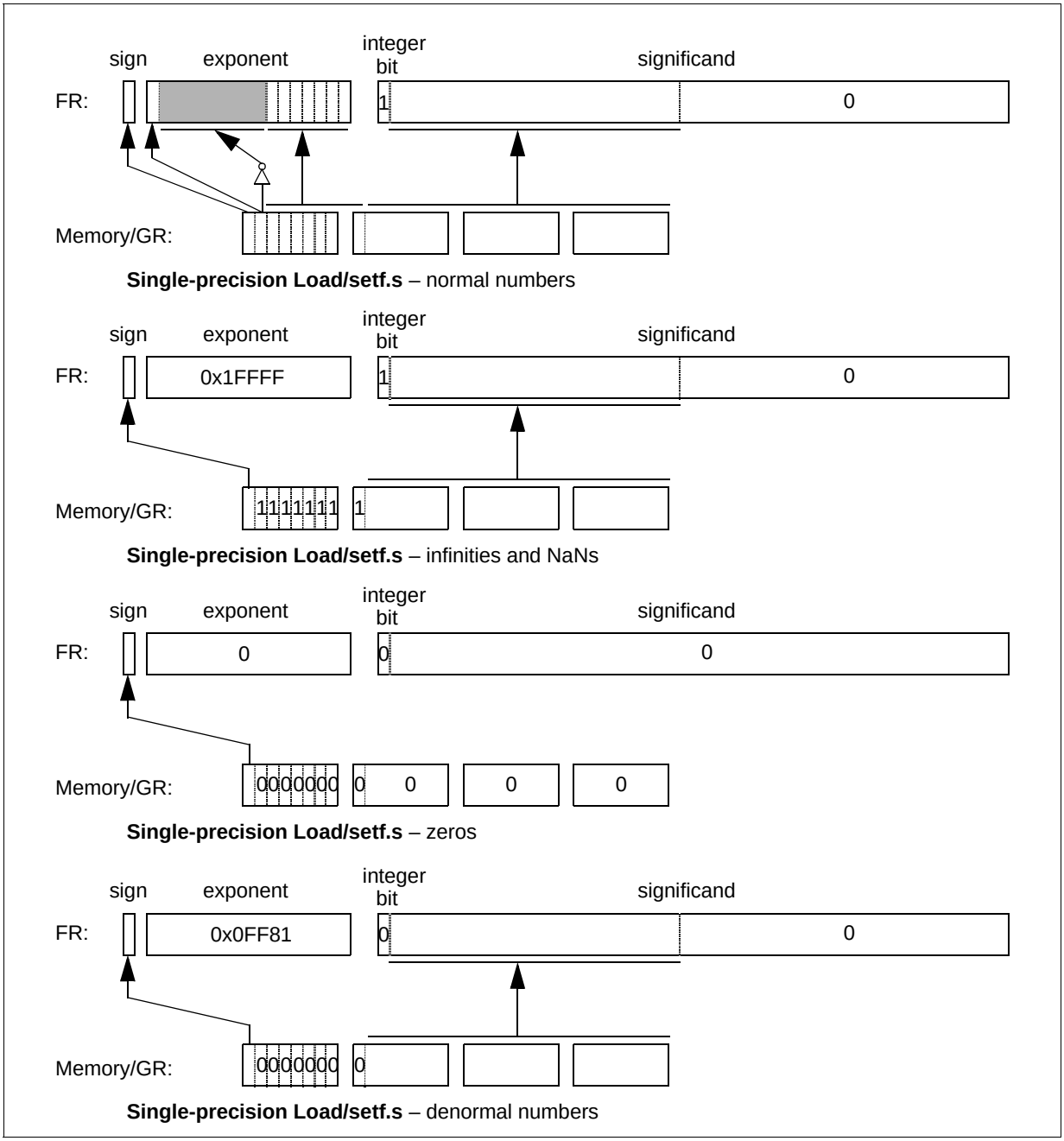


Figure 5-5. Memory to Floating-point Register Data Translation – Double Precision

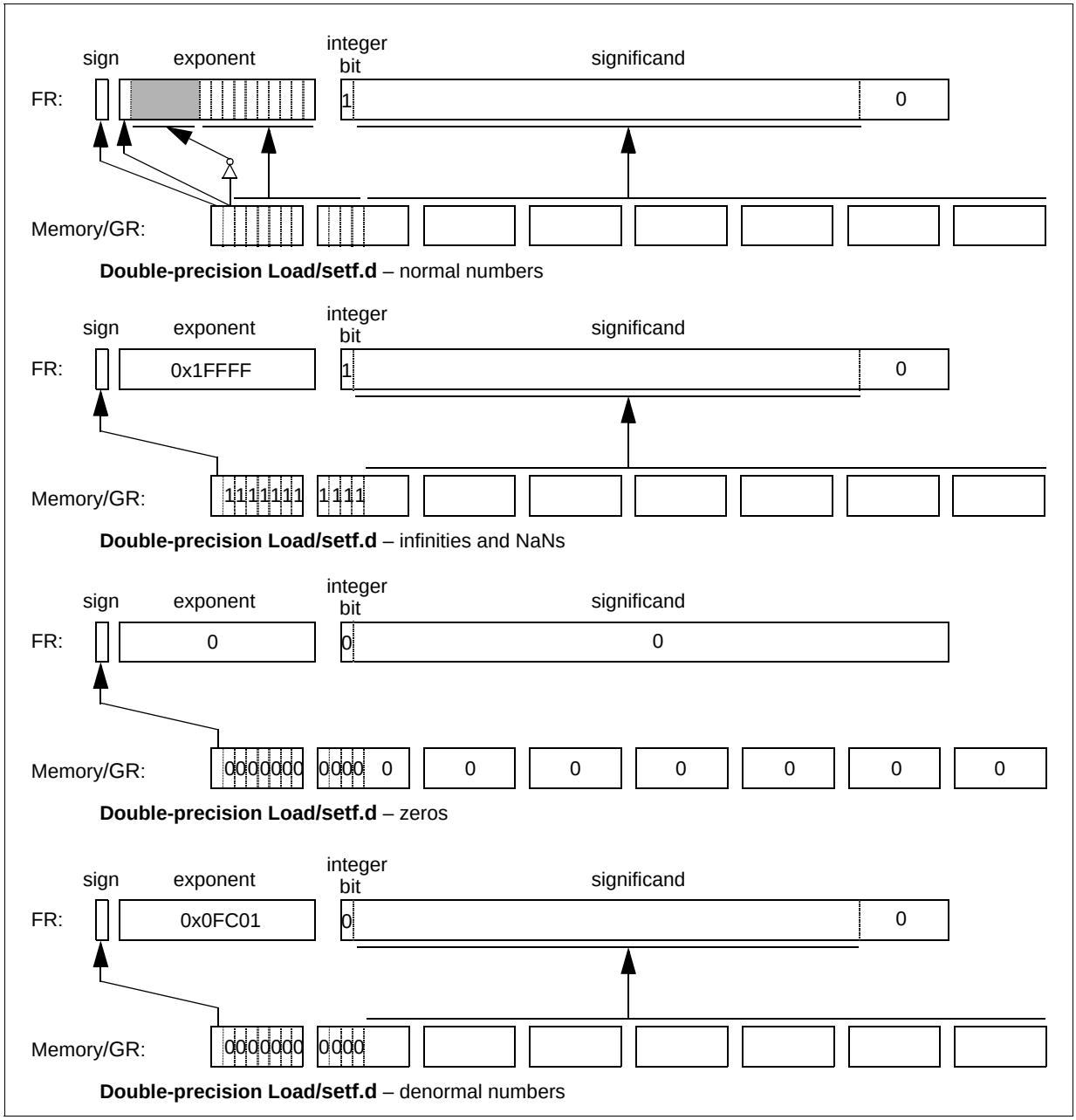


Figure 5-6. Memory to Floating-point Register Data Translation – Double Extended, Integer, Parallel FP and Fill

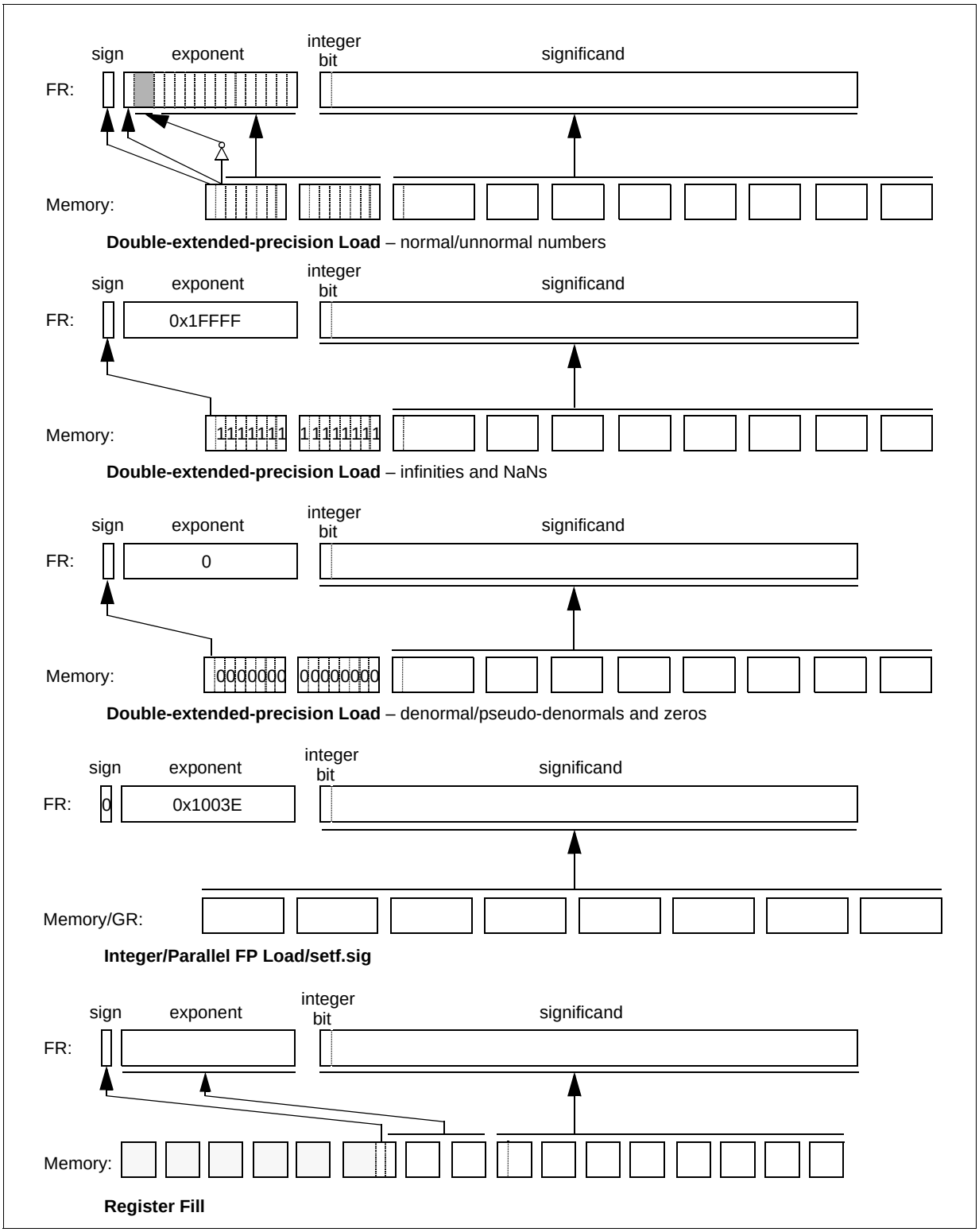


Figure 5-7. Floating-point Register to Memory Data Translation – Single Precision

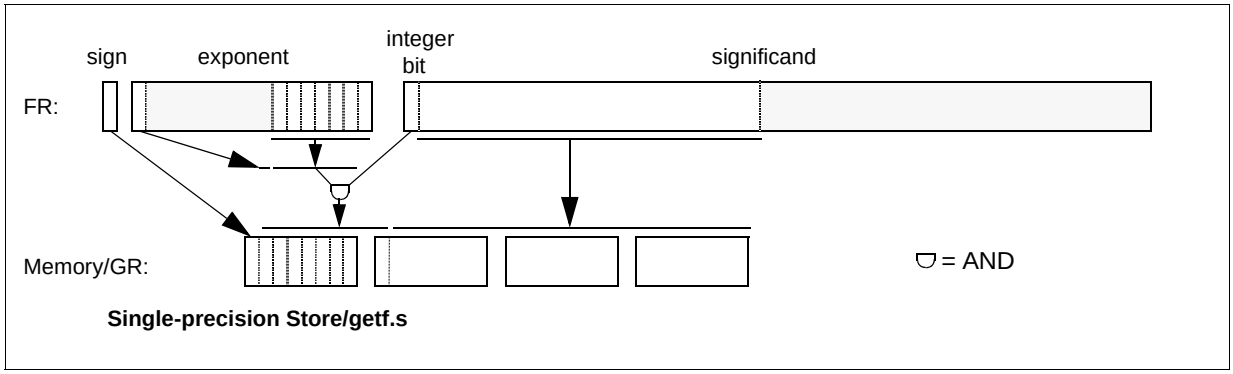


Figure 5-8. Floating-point Register to Memory Data Translation – Double Precision

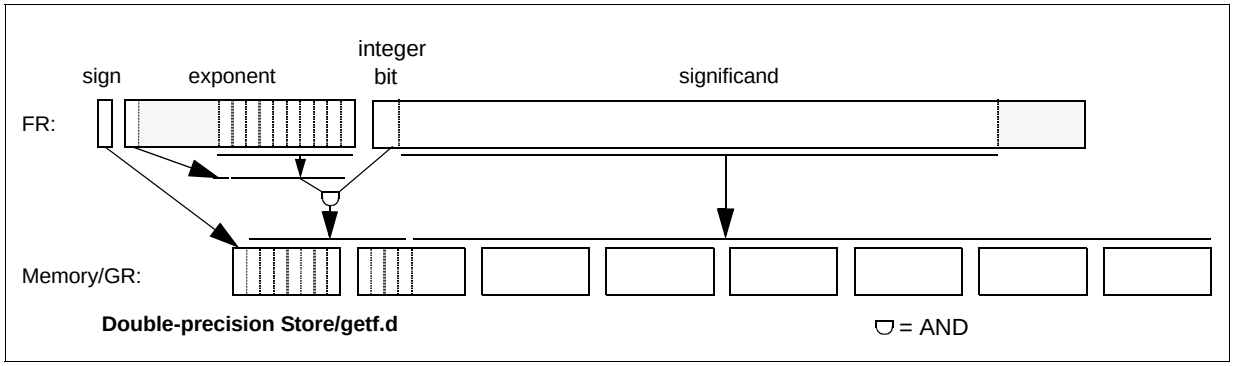
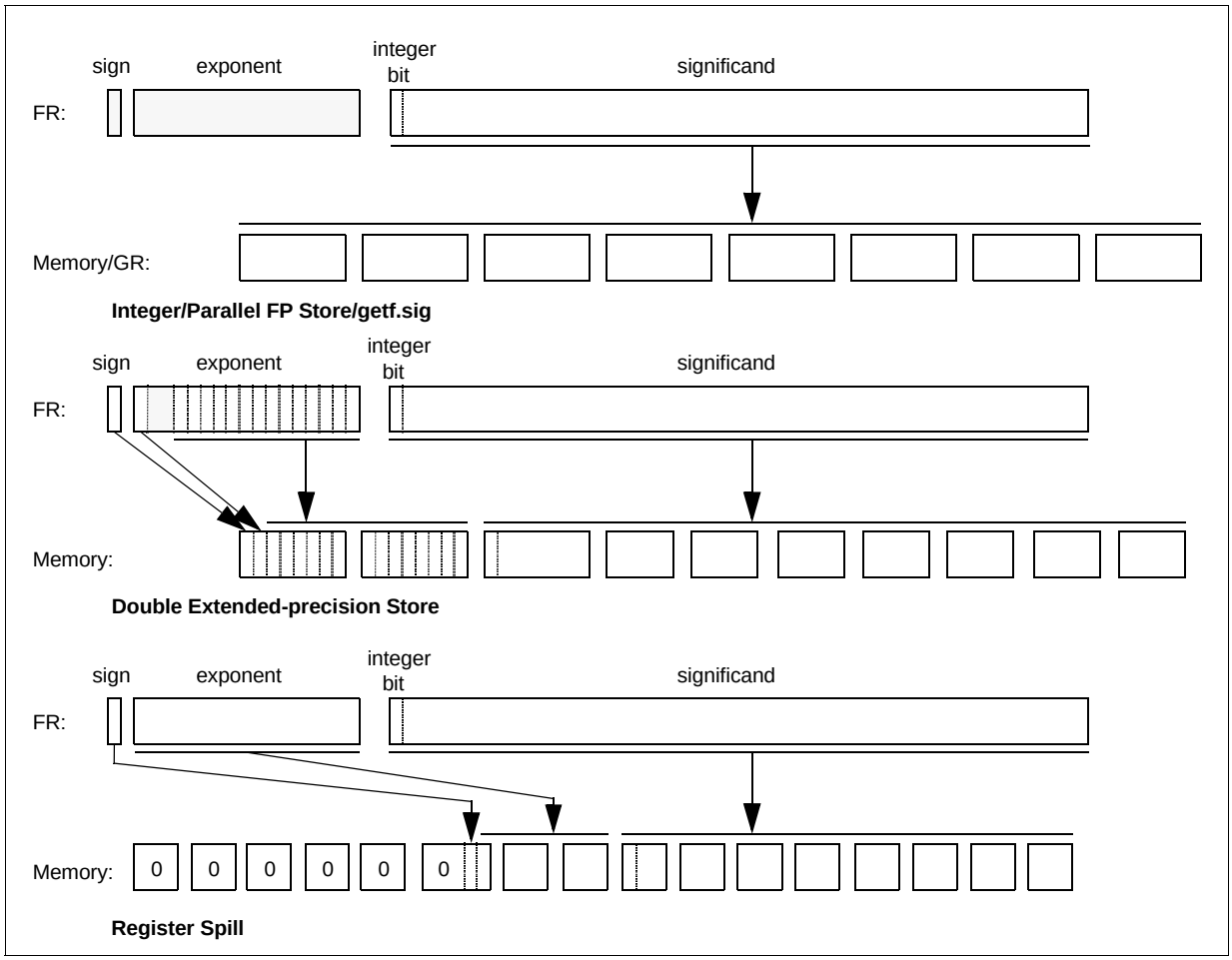
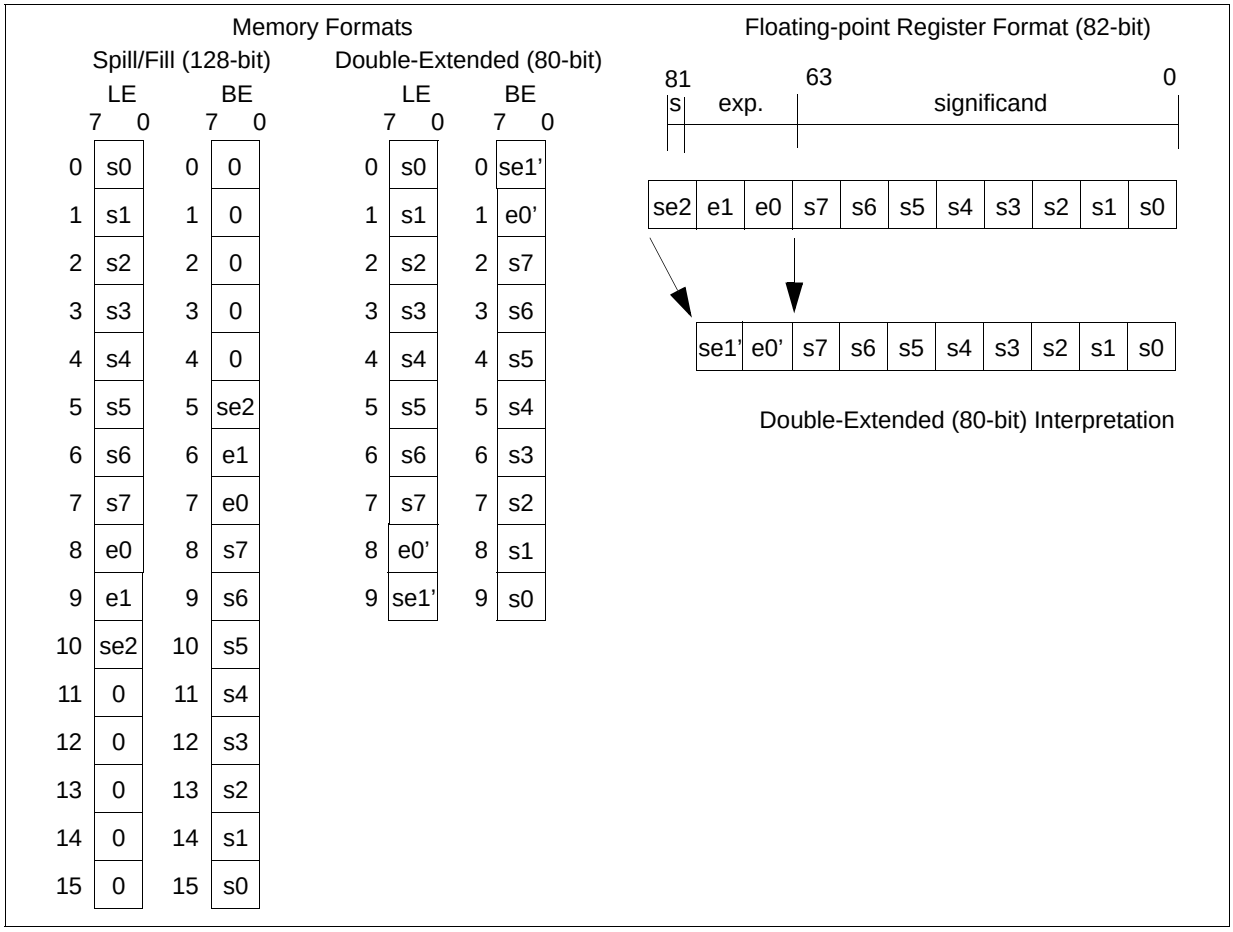


Figure 5-9. Floating-point Register to Memory Data Translation – Double Extended, Integer, Parallel FP and Spill



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, “Byte Ordering”). The byte-ordering for the spill/fill memory and double-extended formats is shown in Figure 5-10.

Figure 5-10. Spill/Fill and Double-extended (80-bit) Floating-point Memory Formats



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, Figure 5-7 and Figure 5-8. 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 (setf)

	General Register		Floating-Point Register (.sig)			Floating-Point Register (.exp)		
Class	**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 (getf)

	Floating-Point Register			General Register (.sig)		General Register (.exp)	
Class	**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	any	any	any	0	significand	0	((sign<<17) | exponent)

5.3.3 Arithmetic Instructions

All 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 1:84..

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 and range of the result. The precision is determined either statically using a.*pc* completer or dynamically using the.*pc* field of the FPSR status field. The range is determined similarly except the.*wre* field of the FPSR status field is also used. Normal (non Parallel FP) arithmetic instructions that do not have a.*pc* completer use the floating-point register format precision and range. See Table 5-6 for details.

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

Table 5-12. Arithmetic Floating-point 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. Arithmetic 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 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 (using the `fcvt.fx` instruction) and then using the `xma` instruction.

The `fmpy` pseudo-operation delivers the IEEE compliant result by rounding the product and without performing the addition inherent in the `fma`. An `fma` with the addend specified as a register other than FR 0, and containing the value +0.0, will not deliver the IEEE compliant multiply result in some cases.

5.3.4 Non-arithmetic Instructions

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.

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 `fneg` pseudo-operation (see Table 5-15) simply reverses the sign bit of the operand and is therefore not equivalent to the IEEE negation operation. For the IEEE negation operation, an `fnma` using FR 1 as the multiplicand and FR 0 as the addend must be used.

Table 5-14 lists the non-arithmetic floating-point instructions and Table 5-15 lists the non-arithmetic pseudo-operation definitions.

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`
Floating-point And	`fand`
Floating-point And Complement	`fandcm`
Floating-point Or	`for`
Floating-point Xor	`fxor`
Floating-point Select	`fselect`

Table 5-15. Non-arithmetic Floating-point Pseudo-operations

Operation	Mnemonic	Operation Used
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`
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

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 main status field 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 branches to recovery code. 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 5-16. 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 multiplicands as signed and unsigned 64-bit integers respectively.

Table 5-17. 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

This section describes the support of the IEEE standard in the areas where specific details are left open to implementation.

5.4.1 Floating-point Interruptions

Floating-point interruptions are precise. The exception reporting and handling occurs on the instruction which causes the interruption. There are three floating-point interruptions: Disabled Floating-Point Register fault, Floating-Point Exception fault, and Floating-Point Exception trap (see Chapter 5, “Interruptions” in Volume 2 for more details).

Exceptions are processed according to a predetermined precedence. Precedence in exception handling means that higher-priority exceptions are flagged first and results are delivered according to the requirements of that exception. Lower-priority exceptions are not flagged even if they occur. For example, dividing an SNaN by zero causes an invalid operation exception (due to the SNaN) and not a zero-divide exception; the exception disabled result is the quieted version of the SNaN, not infinity. However, an IEEE Inexact Floating-Point Exception trap can accompany an IEEE Underflow or Overflow Floating-Point Exception trap.

For instructions that access the floating-point register file, the Disabled Floating-point Register fault has the highest priority.

5.4.1.1 Disabled Floating-point Register Fault

Two bits in the PSR, PSR.dfl and PSR.dfh, (see Section 3.3.2, “Processor Status Register (PSR)” on page 2:22) can be used by an operating system to enable or disable access to two subsets of floating-point registers: FR 2 to FR 31, and FR 32 to FR 127, respectively. The Disabled Floating-Point Register fault occurs when an access (read or write) is made to a FR which has been disabled. Operating systems can use this fault to identify a task as integer or floating-point and optimize the default set of registers which get saved on a task switch. If a mainly integer task is able to use only FR 2 to FR 32 for executing integer multiply and divide operations, then context switch time may be reduced by disabling access to the high floating-point registers.

5.4.1.2 Floating-point Exception Fault

A Floating-Point Exception fault occurs if one of the following four circumstances arises:

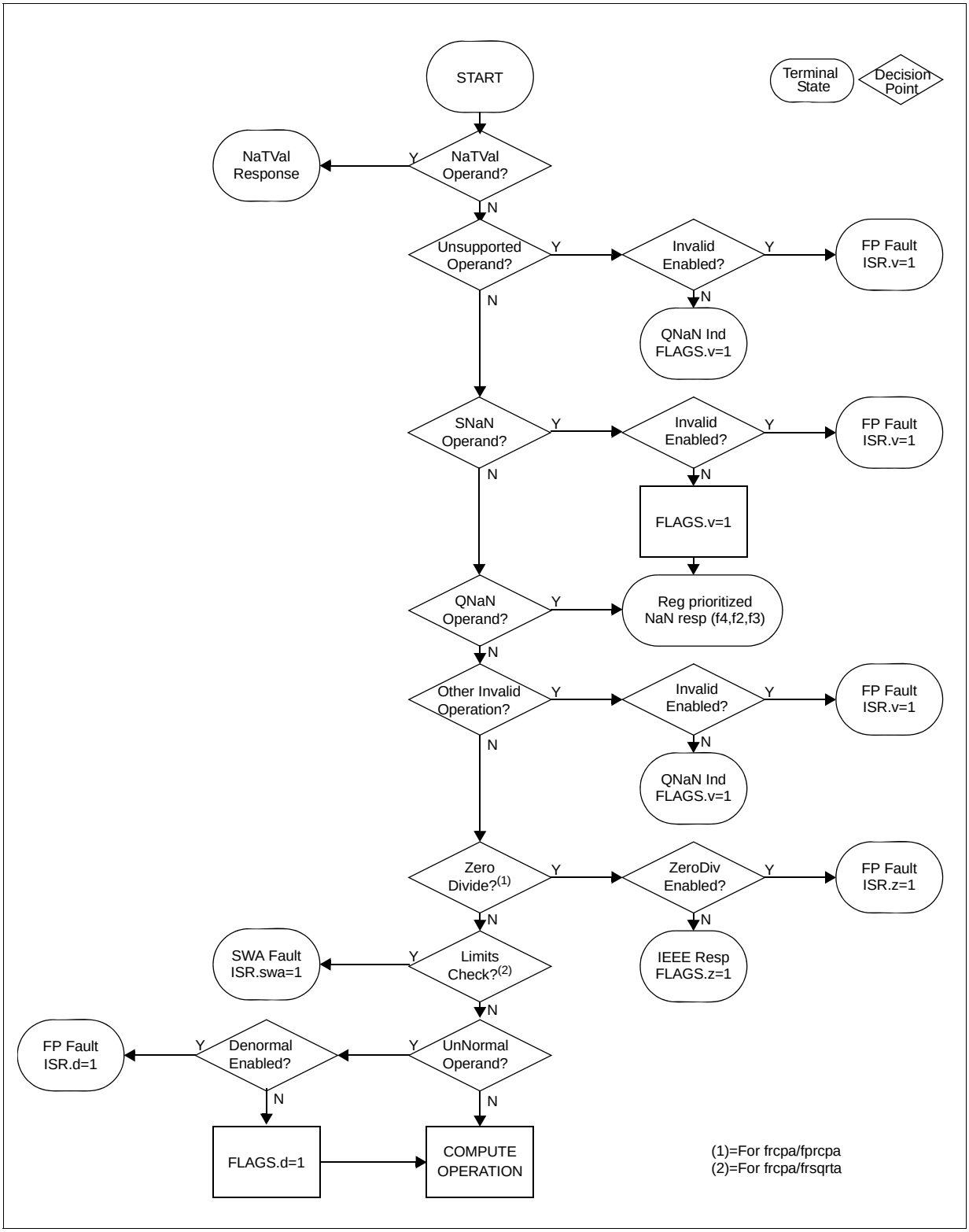
1. The processor requests system software assistance to complete the operation, via the Software Assist fault
2. The IEEE Invalid Operation trap is enabled and this condition occurs
3. The IEEE Zero Divide trap is enabled and this condition occurs
4. The Denormal/Unnormal Operand trap is enabled and an unnormalized operand (denormals are represented as unnormalized numbers in the register file) is encountered by a floating-point arithmetic instruction

If a Floating-Point Exception fault occurs, the only indication of which fault occurred is in the ISR.code. The appropriate status flags are not updated in the FPSR.

There is no requirement that the Software Assist Floating-Point Exception fault ever be signaled (except for certain operands in the `frcpa` and the `frsqrta` instructions), nor is there a mode to force its use. If there is no input NaTVal operand, a processor implementation may signal a Software Assist Floating-Point Exception fault at any time during the operation. In order to ensure maximum floating-point performance, most implementations will not use this exception except in difficult situations such as operations consuming denormal numbers.

The precedence among Floating-point Exception faults for arithmetic operations is depicted in Figure 5-11.

Figure 5-11. Floating-point Exception Fault Prioritization



5.4.1.3 Floating-point Exception Trap

A Floating-point Exception trap occurs if one of the following four circumstances arises:

1. The processor requests system software assistance to complete the operation, via the Software Assist trap
2. The IEEE Overflow trap is enabled and an overflow occurs
3. The IEEE Underflow trap is enabled and an underflow occurs
4. The IEEE Inexact trap is enabled and an inexact result occurs

When an overflow, underflow, or inexact result occurs, the appropriate status flags are updated in the FPSR. If enabled, a Floating-Point Exception trap occurs, and an indication of which enabled trap occurred is stored in ISR.code and the fpa bit in ISR.code (ISR{14}) is set as described in the next paragraph.

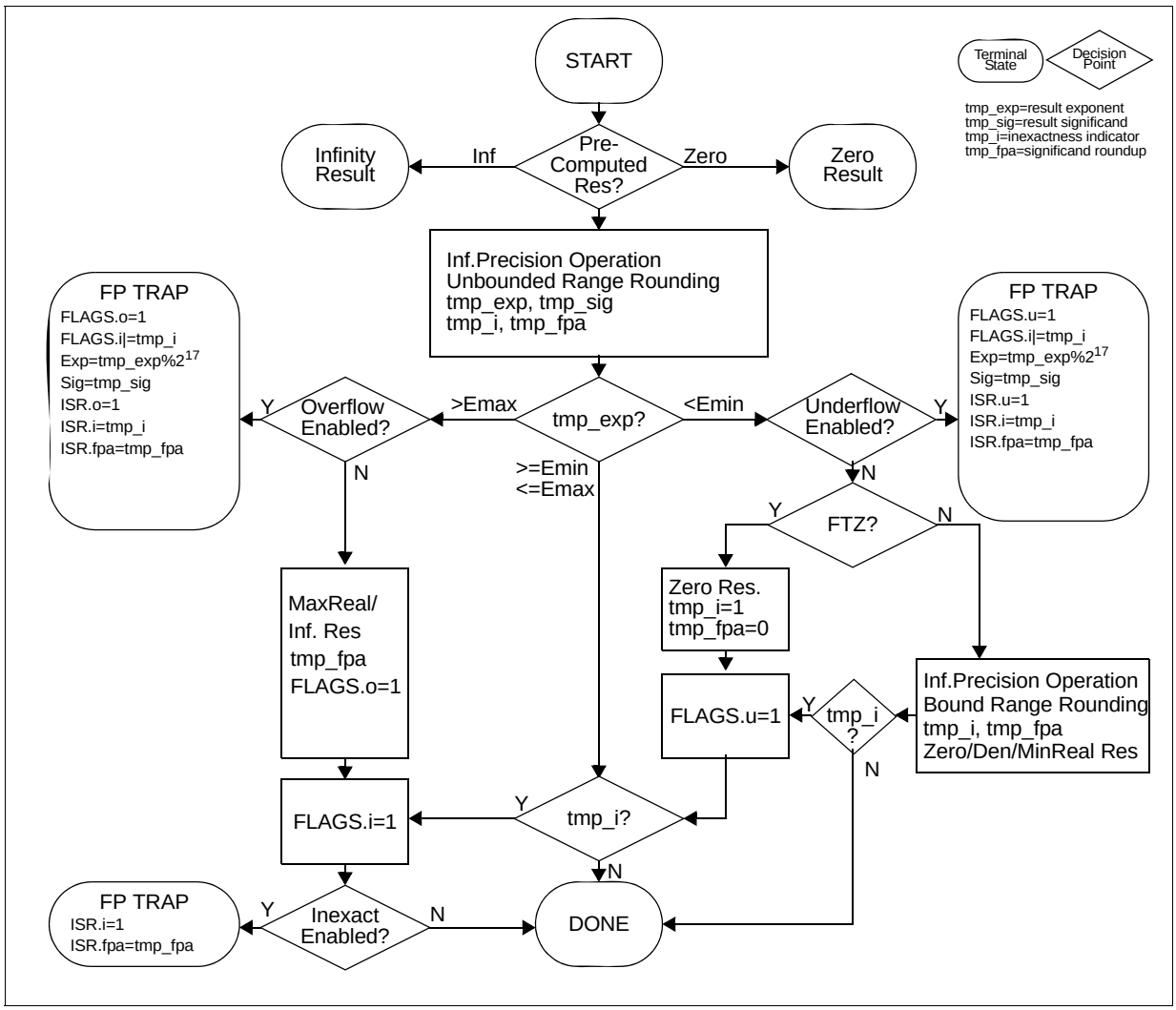
ISR.fpa is set to 1 when the magnitude of the delivered result is greater than the magnitude of the infinitely precise result. It is set to 0 otherwise. The magnitude of the delivered result may be greater if:

- The significand is incremented during rounding, or
- A larger pre-determined value (e.g., infinity) is substituted for the computed result (e.g., when overflow is disabled).

There is no requirement that the Software Assist Floating-Point Exception trap ever be signaled, nor is there a mode to force its use. In order to ensure maximum floating-point performance, most implementations will not use this exception except in difficult situations, such as operations creating denormal numbers. The occurrence of a Software Assist trap is indicated when a trap bit is set in ISR.code, but that trap is disabled. The destination register contains the trap enabled response for that trap.

The precedence among Floating-point Exception traps for arithmetic operations is depicted in Figure 5-12.

Figure 5-12. Floating-point Exception Trap Prioritization



5.4.2 Definition of Overflow

The overflow exception can occur whenever the rounded true result would exceed, in magnitude, the largest finite number in the destination format.

The IEEE Overflow Floating-Point Exception trap disabled response for all normal and Parallel-FP arithmetic instructions is to either return an infinity or the correctly signed maximum finite value for the destination precision. This depends on the rounding mode, the sign of the result, and the operation. An inexact result exception is signaled.

The IEEE Overflow Floating-Point Exception trap enabled response for all normal arithmetic instructions is to return the true biased exponent value MOD 2^{17} and for all Parallel-FP arithmetic instructions is to return the true biased exponent value MOD 2^{8}. The value's significand is rounded to the specified precision and written to the destination register. If the rounded value is different

from the infinitely-precise value, then inexactness is signaled. If the significand was rounded by adding a one to its least significant bit, then bit `fpa` in ISR.code is set to 1. Finally, an interruption due to a Floating-Point Exception trap will occur.

Note that when rounding to single, double, or double-extended real, the overflow trap enabled response for normal (non Parallel FP) arithmetic instructions is not guaranteed to be in the range of a valid single, double, or double-extended real quantity, because it is in 17-bit exponent format.

5.4.3 Definition of Tininess, Inexact and Underflow

Tininess is detected after rounding, and is said to occur when a non-zero result (computed as though the exponent range were unbounded) would lie strictly between $+2^{Emin}$ and -2^{Emin}. See Table 5-1 for the values of Emin for each real type. Creation of a tiny result may cause an exception later (such as overflow upon division because it is so small).

Inexactness is said to occur when the result differs from what would have been computed if both the exponent range and precision were unbounded.

How tininess and inexactness trigger the underflow exception depends on whether the Underflow Floating-Point Exception trap is disabled or enabled. If the trap is disabled then the underflow exception is signaled when the result is both tiny and inexact. If the trap is enabled then the underflow exception is signaled when the result is tiny, regardless of inexactness. Note that in the event that the Underflow Floating-Point Exception trap is disabled and tininess but not inexactness occurs, then neither underflow nor inexactness is signaled, and the result is a denormal.

The IEEE Underflow Floating-Point Exception trap disabled response for all normal and Parallel-FP arithmetic instructions is to denormalize the infinitely precise result and then round it to the destination precision. The result may be a denormal, zero, or a normal. The inexact exception is signaled when appropriate.

The IEEE Underflow Floating-Point Exception trap enabled response for all normal arithmetic instructions is to return the true biased exponent value MOD $2^{17}$and for all Parallel-FP arithmetic instructions is to return the true biased exponent value MOD 2^{8}. The significand is rounded to the specified precision and written to the destination register independent of the possibility of the exponent calculation requiring a borrow. If the rounded value is different from the infinitely-precise value, then inexactness is signaled. If the significand was rounded by adding a one to its least significant bit, then bit `fpa` in ISR.code is set to 1. Finally, an interruption due to a Floating-Point Exception trap will occur.

Note: When rounding to single, double, or double-extended real, the underflow trap enabled response for normal (non Parallel FP) arithmetic instructions is not guaranteed to be in the range of a valid single, double, or double-extended real quantity, because it is in 17-bit exponent format.

When Flush-to-Zero mode is enabled, the behavior for tiny results is different. If an instruction would deliver a tiny result, a correctly signed zero is delivered instead and the appropriate FPSR.sf*x*.u and FPSR.sf*x*.i bits are set. This mode may improve the performance on implementations that do not implement denormal handling in hardware. When the Flush-to-Zero mode is enabled, floating-point exception software assist traps will not occur when producing tiny results.

5.4.4 Integer Invalid Operations

Floating-point to integer conversions which are invalid (in the IEEE sense) signal an Invalid Operation Floating-Point Exception fault. If the IEEE Invalid Operation trap is disabled, then the largest magnitude negative integer is the result, even for unsigned integer operations.

5.4.5 Definition of Arithmetic Operations

Arithmetic operations are those that compute on the operands by treating each operand's encoding as a value, whereas non-arithmetic operations perform bit manipulations on the input operands without regard to the value represented by the encoding (except for NaTVal detection). Non-arithmetic instructions do not cause Floating-point Exception faults or traps, but can cause the Disabled Floating-point Register fault.

5.4.6 Definition 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*.

5.4.7 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

5.4.8 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 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. When computations with higher precision operands produce values beyond the destination precision range, the information provided in the ISR.code allows the true result to be unambiguously determined by software. The correct wrapping count and the appropriate bias amount can also be computed.
- An IEEE style quad-precision real type that is supported in software.

IA-32 Application Execution Model in an Intel® Itanium® System Environment 6

This chapter describes the IA-32 execution model from the perspective of an application programmer using the Itanium architecture, interfacing with IA-32 code, while operating in the Itanium System Environment. The main features covered are:

- IA-32 integer, segment, floating-point, MMX technology, and SSE register state mappings
- Instruction set transitions
- 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 *IA-32 Intel® Architecture Software Developer's Manual* for details regarding IA-32 application programming model.

The Itanium architecture can support 16-bit Real Mode, 16-bit VM86, and 16-bit/32-bit Protected Mode IA-32 applications in the context of an Itanium architecture-based operating system. Whether an IA-32 application is actually supported on specific operating systems is determined by the infrastructure provided by that specific operating system.

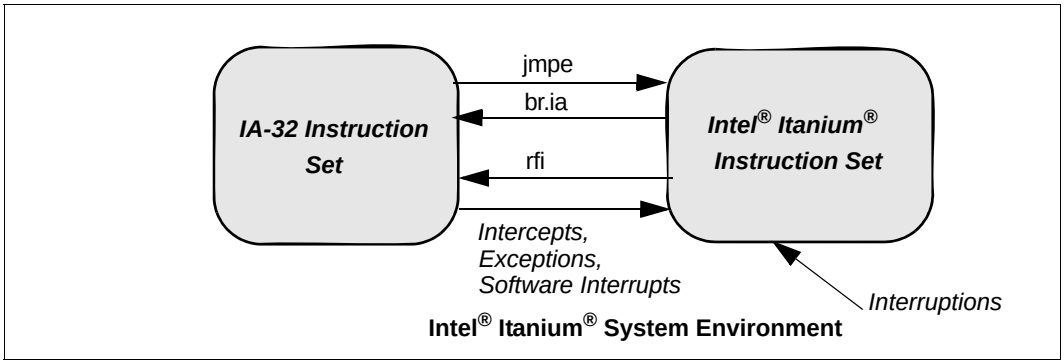
6.1 Instruction Set Modes

The processor can be executing either IA-32 or Itanium instructions at any point in time. PSR.is (defined in Section 3.3.2, "Processor Status Register (PSR)" on page 2:22) specifies the currently executing instruction set, where 1 indicates IA-32 instructions are executing, and 0 indicates Itanium instructions are executing. Three special instructions and interruptions are defined to transition the processor between the IA-32 and the Itanium instruction sets as shown in Figure 6-1.

- `jmpe` (IA-32 instruction) Jump to an Itanium target instruction, and transition to the Itanium instruction set.
- `br.ia` (Itanium instruction) Branch to an IA-32 target instruction, and change the instruction set to IA-32.
- `rfi` (Itanium instruction) "Return from interruption" is defined to return to either an IA-32 or Itanium instruction when resuming from an interruption.
- Interruptions transition the processor to the Itanium 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. Instruction Set Transition Model



6.1.1 Instruction Set Execution in the Intel® Itanium® Architecture

While the processor executes from the Itanium instruction set (PSR.is is 0):

- Itanium instructions are fetched, decoded and executed by the processor.
- Itanium instructions can access the entire Itanium and IA-32 application register state. This includes IA-32 segment descriptors, selectors, general registers, physical floating-point registers, MMX technology registers, and SSE registers. See Section 6.2, “IA-32 Application Register State Model” 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. All computed addresses are virtual addresses.
- 2^{64} virtual addresses can be generated and 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 (PSR.is is 1) within the Itanium 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, and SSE 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, and SSE registers, selectors, EFLAGS, FP registers and FP control registers). Itanium application and control register state is not visible, e.g. branch, predicate, application, control, debug, test, and performance monitor registers.
- 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.
- Virtual addresses are confined to the lower 4G bytes of virtual region 0. Itanium architecture 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 superseded by resources defined in the Itanium architecture. 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` (Itanium instruction) should be consulted for details.

Operating systems can disable instruction set transitions (`jmpe` and `br.ia`) by setting PSR.di to one. If PSR.di is one, execution of `jmpe` or `br.ia` results in a Disabled Instruction Set Transition Fault. System level instruction set transitions due to either `rfi` or an interruption ignore the state of PSR.di (defined in Section 3.3.2, “Processor Status Register (PSR)” on page 2:22).

6.1.3.1 JMPE Instruction

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

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

`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.

Transitions into the Itanium instruction set do not change the privilege level of the processor.

6.1.3.2 Branch to IA Instruction

The `br.ia` instruction is used to unconditionally branch to the IA-32 instruction set. 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.

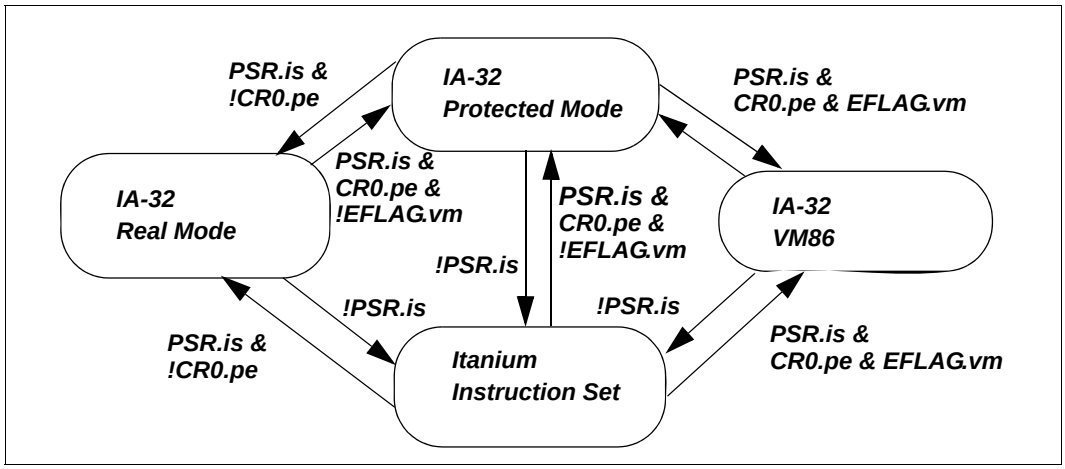
Software should ensure the code segment descriptor and selector are properly loaded before issuing the branch. If the target EIP value exceeds the code segment limit or has a code segment privilege violation, an IA-32 GPFault(0) exception is reported on the target IA-32 instruction.

The processor does not ensure Itanium instruction set generated writes into the IA-32 instruction stream are observed by the processor. For details, see "Self Modifying Code" on page 1:129. Before entering the IA-32 instruction set, Itanium architecture-based 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 left in an undefined state after IA-32 instruction set execution. Software can not rely on the value of these registers across an instruction set transition. For details, see "Register Stack Engine" on page 1:130.

6.1.4 IA-32 Operating Mode Transitions

As described in "IA-32 Instruction Set Execution" on page 1:108, `jmpe`, `br.ia`, and `rfi` instructions and interruptions can transition the processor between the two instruction set modes. Transitions are allowed between the Itanium architecture and all major IA-32 modes. As shown in Figure 6-2, `br.ia` and `rfi` will transition the processor from the Itanium instruction set into IA-32 VM86, Real Mode or Protected Mode. While `jmpe` and interruptions will transition the processor from either IA-32 VM86, Real Mode or Protected Mode into the Itanium instruction set. Mode transitions between IA-32 Real Mode, Protected Mode and VM86 definitions are the same as those defined in the *IA-32 Intel® Architecture Software Developer's Manual*.

Figure 6-2. Instruction Set Mode Transitions



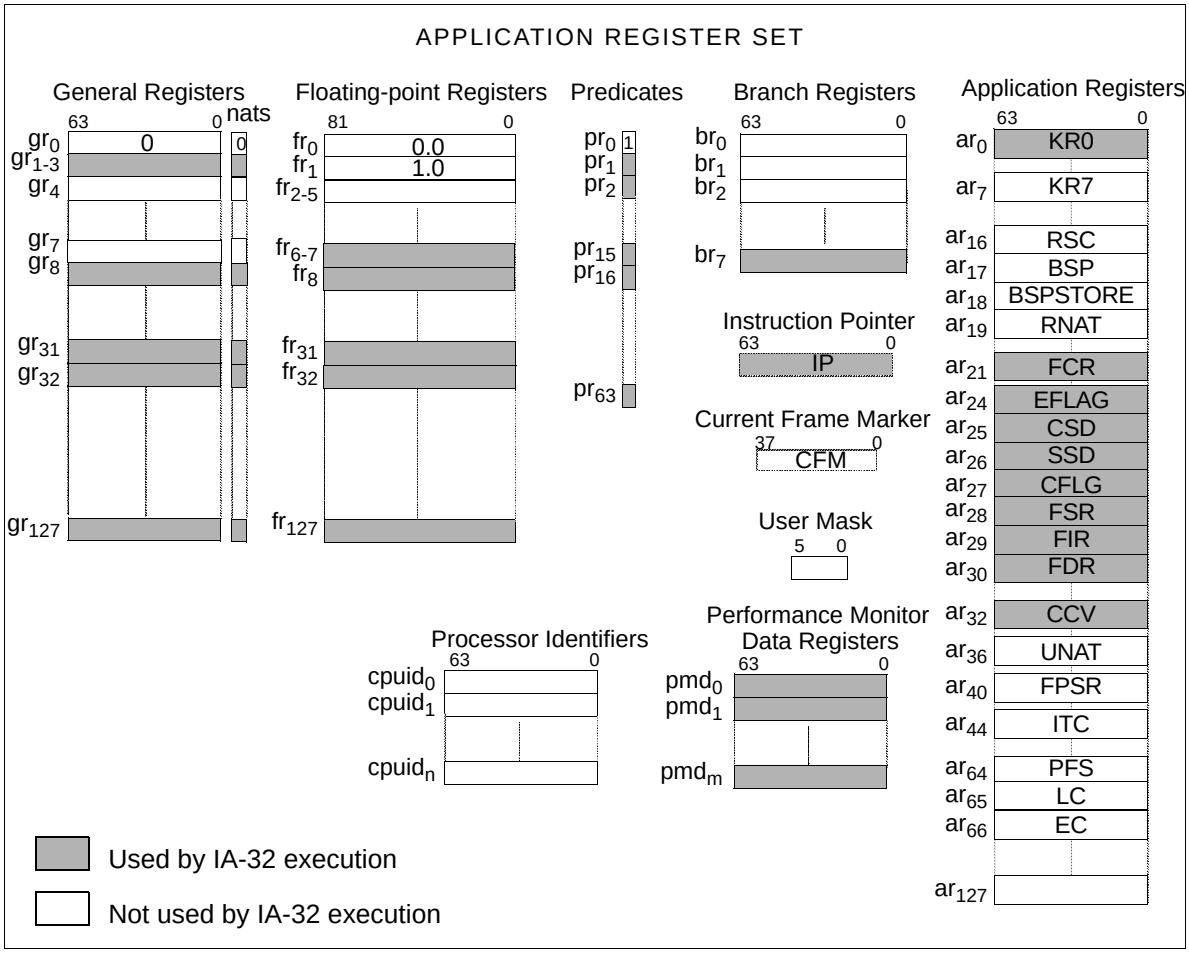
Itanium architecture-based interface code is responsible for setting up and loading a consistent Protected Mode, Real Mode, or VM86 environment (e.g. loading segment selectors and descriptors, etc.) as defined in "Segment Descriptor and Environment Integrity" on page 1:117. The processor applies additional segment descriptor checks to ensure operations are performed in a consistent manner.

6.2 IA-32 Application Register State Model

As shown in Figure 6-3 and Table 6-1, IA-32 general purpose registers, segment selectors, and segment descriptors, are mapped into the lower 32-bits of Itanium general purpose registers GR8 to GR31. The floating-point register stack, MMX technology, and SSE registers are mapped on Itanium floating-point registers FR8 to FR31.

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

Figure 6-3. IA-32 Application Register Model



Some Itanium registers are modified to an undefined state by hardware as a side-effect during IA-32 instruction set execution as noted in Table 6-1 and Figure 6-2. Generally, Itanium system state is not affected by IA-32 instruction set execution. Itanium architecture-based code can reference all registers (including IA-32), 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 Itanium 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 Itanium instruction execution according to Itanium software conventions. Software should follow IA-32 and Itanium calling conventions for these registers.
- **Undefined**: Registers marked as undefined may be 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. Itanium architecture-based 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.

Table 6-1. IA-32 Application Register Mapping

Intel® Itanium® Reg	IA-32 Reg	Convention	Size	Description
General Purpose Integer Registers				
GR0				constant 0
GR1-3		undefined[f]		scratch for IA-32 execution
GR4-7		unmodified		Intel® Itanium® 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		undefined[f]		scratch for IA-32 execution
GR24	ESD	IA-32 state	64	IA-32 segment descriptors (register format)[b]
GR25-26		undefined[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		undefined[d]		IA-32 code execution space

Table 6-1. IA-32 Application Register Mapping (Continued)

Intel® Itanium® Reg	IA-32 Reg	Convention	Size	Description
Process Environment				
IP	IP	shared	64	shared IA-32 and Intel® Itanium® virtual Instruction Pointer
Floating-point Registers				
FR0				constant +0.0
FR1				constant +1.0
FR2-5		unmodified		Intel® Itanium® preserved registers
FR6-7		undefined		IA-32 code execution space
FR8	MM0/FP0	IA-32 state	64/80	IA-32 Intel 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 SSE 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		undefined[f]		IA-32 code execution space
Predicate Registers				
PR0				constant 1
PR1-63		undefined[f]		IA-32 code execution space
Branch Registers				
BR0-5		unmodified		Intel® Itanium® preserved registers
BR6-7		undefined		IA-32 code execution space
Application Registers				
RSC		unmodified		not used for IA-32 execution Intel® Itanium® preserved registers
BSP				
BSPSTORE				
RNAT				
CCV		undefined[f]	64	IA-32 code execution space
UNAT		unmodified		not used for IA-32 execution, Intel® Itanium® preserved register
FPSR.sf0		unmodified		Intel® Itanium® numeric status and controls register
FPSR.sf1,2,3		undefined[f]		IA-32 code execution space.

Table 6-1. IA-32 Application Register Mapping (Continued)

Intel® Itanium® Reg	IA-32 Reg	Convention	Size	Description
FSR	FSW,FTW, MXCSR	IA-32 state	64	IA-32 numeric status and tag word and SSE status
FCR	FCW, MXCSR		64	IA-32 numeric and SSE 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 Intel® Itanium® Interval Timer
PFS		unmodified		not used for IA-32 code execution, Prior EC is preserved in PFM Intel® Itanium® 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.

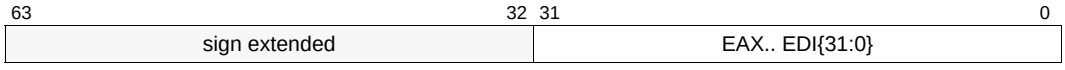
a. On transitions into the IA-32 instruction set the upper 32-bits are ignored. On exit the upper 32-bits are sign extended from bit 31.
b. Segment descriptor formats differ from the iA-32 memory format, see "IA-32 Segment Registers" on page 1:115 for details. Modification of a selector or descriptor does not set the access/busy bit in memory.
c. The GDT/LDT descriptors are NOT protected from modification by Itanium architecture-based user level code
d. All registers in the current and prior registers frames are left in an undefined state after IA-32 execution. Software must preserve these values before entering the IA-32 instruction set.
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 left an undefined state following IA-32 instruction set execution. 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 Itanium general registers GR8 to GR15. Values in the upper 32-bits of GR8 to GR15 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 Itanium calling conventions, the required IA-32 state must be loaded in memory or registers by Itanium architecture-based code before entering the IA-32 instruction set.

Figure 6-4. IA-32 General Registers (GR8 to GR15)



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 Itanium 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 Itanium 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 wraparound 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 6-6. 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 Itanium architecture-based code. IA-32 segment register load instructions unscramble the GDT/LDT memory format into the descriptor register format on a segment register load. Itanium architecture-based software can also directly load descriptor registers provided they are properly unscrambled by software. When Itanium architecture-based software loads these registers, no data integrity checks are performed at that time if illegal values are loaded in any fields. For a complete definition of all bit fields and field semantics refer to the *IA-32 Intel® Architecture Software Developer's Manual.*

Figure 6-5. IA-32 Segment Register Selector Format

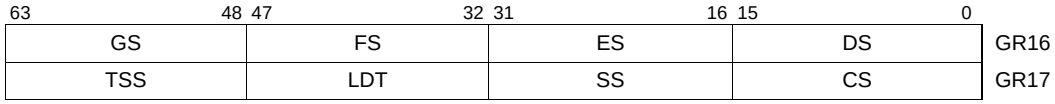


Figure 6-6. IA-32 Code/Data Segment Register Descriptor Format

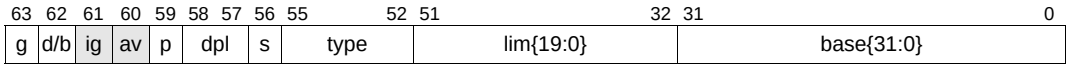


Table 6-2. IA-32 Segment Register Fields

Field	Bits	Description
selector	15:0	Segment Selector value, see the *IA-32 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.

Table 6-2. IA-32 Segment Register Fields (Continued)

Field	Bits	Description
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 *IA-32 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 *IA-32 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.
p	59	Segment Present bit. If 0, and a IA-32 memory reference uses this segment an IA_32_Exception(GPFault) is generated for data segments (CS, DS, ES, FS, GS) and an IA_32_Exception(StackFault) for SS.
av	60	Ignored – This field is ignored by the processor during IA-32 instruction set execution. This field is available for IA-32 software use and there will be no future use for this field. For Itanium instructions, implementations which do not support the `ld16`, `st16` and `cmp8xchg16` instructions can either ignore writes and return zero on reads, or write the value and return the last value written on reads. Implementations which do support these instructions write the value and return the last value written on reads.
ig	61	Ignored – 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 IA-32 software. For Itanium instructions, implementations which do not support the `ld16`, `st16` and `cmp8xchg16` instructions can either ignore writes and return zero on reads, or write the value and return the last value written on reads. Implementations which do support these instructions write the value and return the last value written on reads.
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 Intel® Itanium® 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 Itanium architecture-based code to the required values based on IA-32 and Itanium calling conventions and the segmentation model used.

Itanium architecture-based 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, Itanium architecture-based software can switch to the IA-32 instruction set and perform the required segment load with an IA-32 Mov Sreg instruction. If Itanium architecture-based 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 Itanium 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 Itanium instruction set into the general purpose register file perform no such protection checks, nor are segment Access-bits updated by the processor.

If Itanium architecture-based 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 "IA-32 Environment Runtime Integrity Checks" on page 1:119. On the transitions between IA-32 and Itanium architecture-based 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	VM86 Mode
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 (Continued)

Register	Field	Real Mode	Protected Mode	VM86 Mode
PSR	**cpl**	**0**	**Privilege Level**	**3**
EFLAG	**vm**	**0**	**0**	**1**
CR0	**pe**	**0**	**1**	**1**
LDT,GDT, TSS	selector	N/A	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

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

- Itanium architecture-based software should ensure the stack segment descriptor register's DPL==PSR.cpl.
- For DSD, ESD, FSD and GSD segment descriptor registers, Itanium architecture-based software should ensure DPL>=PSR.cpl.
- For CSD segment descriptor register, Itanium architecture-based 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

Itanium architecture-based 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`).
- Itanium architecture-based 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, Itanium architecture-based 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.
- Itanium architecture-based 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

Itanium architecture-based 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.

- Itanium architecture-based software should ensure PSR.cpl is 0
- Itanium architecture-based 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)
- Itanium architecture-based 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 Runtime Integrity Checks

Processors in the Itanium processor family perform additional runtime checks to verify the integrity of the IA-32 environments. These checks are in addition to the runtime checks defined on IA-32 processors and are high-lighted in Table 6-4. Existing IA-32 runtime checks are listed but not highlighted. Descriptor fields not listed in the table are not checked. As defined in the table, runtime 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 runtime checks are not performed during transitions from the Itanium instruction set to the IA-32 instruction set.

Table 6-4. IA-32 Environment Runtime 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.vmC FLG.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))			

Table 6-4. IA-32 Environment Runtime Integrity Checks (Continued)

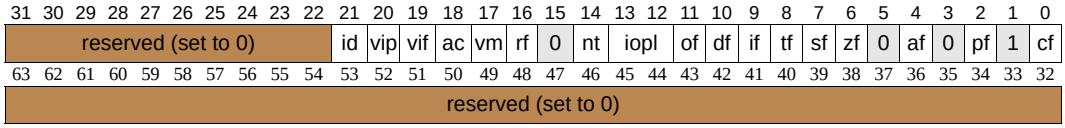
Reference	Resource	Real Mode	Protected Mode	VM86Mode	Fault
all code fetches CS	dpl	ignored		dpl is not 3	Code Fetch Fault (GPFault(0))
	d-bit			is not 16-bit	
	type	ignored (can be exec or data)			
		GPFault if data expand down			
	s, p, a-bits	are not 1			
	g-bit/limit	segment limit violation			
data 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 Itanium instruction execution. See Table 6-5, “IA-32 EFLAGS Register Fields” on page 1:121 for the behavior on IA-32 and Itanium instruction reads/writes to this application register. For details on system flags in the IA-32 EFLAGS register, see Section 10.3.2, “IA-32 System EFLAG Register” on page 2:235.

Figure 6-7. IA-32 EFLAG Register (AR24)



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 Itanium instructions. Flags ID, OF, DF, SF, ZF, AF, PF and CF are defined in the *IA-32 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 *IA-32 Intel® Architecture Software Developer's Manual* for details.
	1	Ignored – For IA-32 instructions, writes are ignored, reads return one. For Itanium instructions, the implementation can either ignore writes and return one on reads; or write the value, and return the last value written on reads.
	3,5, 15	Ignored – For IA-32 instructions, writes are ignored, reads return zero. For Itanium instructions, the implementation can either ignore writes and return zero on reads, or write the value and return the last value written on reads.
pf	2	IA-32 Parity Flag. See the *IA-32 Intel® Architecture Software Developer's Manual* for details.
af	4	IA-32 Aux Flag. See the *IA-32 Intel® Architecture Software Developer's Manual* for details.
zf	6	IA-32 Zero Flag. See the *IA-32 Intel® Architecture Software Developer's Manual* for details.
sf	7	IA-32 Sign Flag. See the *IA-32 Intel® Architecture Software Developer's Manual* for details.
tf	8	See Section 10.3.2, "IA-32 System EFLAG Register" on page 2:235.
if	9	
df	10	IA-32 Direction Flag. See the *IA-32 Intel® Architecture Software Developer's Manual* for details.
of	11	IA-32 Overflow Flag. See the *IA-32 Intel® Architecture Software Developer's Manual* for details.
iopl	13:12	See Section 10.3.2, "IA-32 System EFLAG Register" on page 2:235.
nt	14	
rf	16	
vm	17	
ac	18	
vif	19	
vip	20	
id	21	
	63:22	This field is reserved for IA-32 instructions – reads return zeros and non-zero writes causes IA_32_Exception (General Protection) faults. For Itanium instructions, the implementation can either raise Reserved Register/Field fault on non-zero writes and return zero on reads, or write the value (no Reserved Register/Field fault), and return the last value written on reads.

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 Itanium 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 Itanium 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

Intel® Itanium® Reg	IA-32 Reg	Size (bits)	Description
FR8	ST[(TOS + N)==0]	80	IA-32 numeric register stack Accesses to FR8 - FR15 by Intel® Itanium® instructions 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 SSE control register
FSR (AR28)	FSW,FTW, MXCSR	64	IA-32 numeric and SSE 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 Itanium 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. Itanium architecture-based 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 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 Itanium instructions, loading a single or double denormal results in an un-normalized denormal value placed in the target floating-point register. There are two canonical exponent values in the Itanium architecture which indicate single precision and double precision denormals.

When transferring floating-point values from Itanium 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-32 to Itanium architecture-based code via the floating-point registers, software must ensure the following:

- Single or double precision Itanium denormals must be converted into a normalized double extended precision value expected by IA-32 instructions. Software can convert Itanium 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. The Itanium architecture uses 82 bits (17-bit widest range exponent) for intermediate calculations. Software must ensure all 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`.
- 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 cannot 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 Itanium denormals.

6.2.5.3 IA-32 Floating-point Control Registers

FPSR controls Itanium 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 SSE instructions have separate control and status registers, namely floating-point control register (FCR) and floating-point status register (FSR).

FCR contains the IA-32 FCW bits and all SSE control bits as shown in Figure 6-8.

FSR contains the IA-32 floating-point status flags FSW, FTW, and SSE status fields as shown in Figure 6-9. 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. Itanium architecture-based 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 reads/writes of FSR and FCR by Itanium instructions.

Software must ensure that FCR and FSR are properly loaded for IA-32 numeric execution before entering the IA-32 instruction set. For Itanium instructions accessing ignored fields, the implementation can either ignore writes and return the specified constant on reads, or write the value and return the last value written on reads. For Itanium instructions accessing reserved fields, the implementation can either raise Reserved Register/Field fault on non-zero writes and return zero on reads, or write the value (no Reserved Register/Field fault), and return the last value written on reads.

Figure 6-8. IA-32 Floating-point Control Register (FCR)

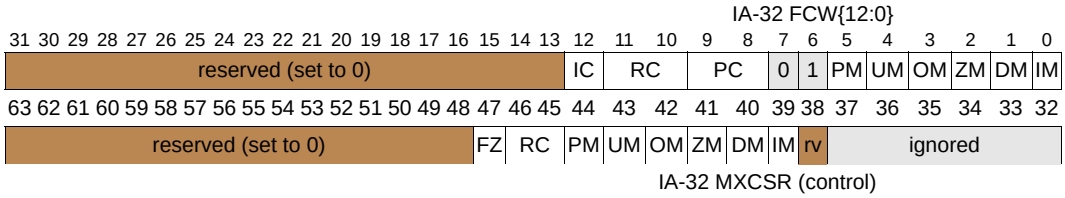


Figure 6-9. IA-32 Floating-point Status Register (FSR)

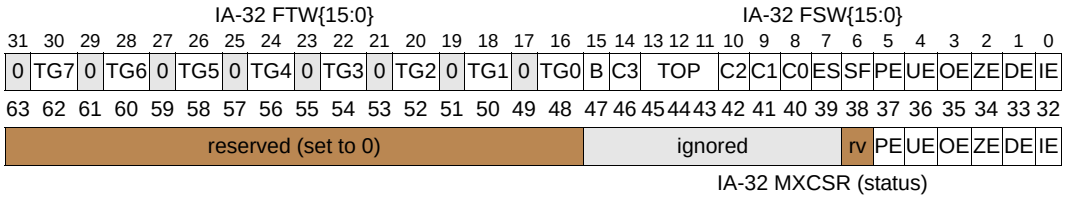


Table 6-7. IA-32 Floating-point Status Register Mapping (FSR)

IA-32 State	Intel® Itanium® State	Bits	IA-32 Usage	Usage in the Intel® Itanium® Architecture
FSW, FTW, MXCSR state in the FSR Register				
FSW.ie	FSR.ie	0	Invalid operation Exception	None of these bits reflect the status of Intel® Itanium® floating-point execution. See the *IA-32 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	SSE Invalid operation Exception	Does not reflect the status of Intel® Itanium® floating-point execution. See *IA-32 Intel® Architecture Software Developer's Manual* for details.
MXCSR.de	FSR.de	33	SSE Denormalized operand Exception	
MXCSR.ze	FSR.ze	34	SSE Zero divide Exception	
MXCSR.oe	FSR.oe	35	SSE Overflow Exception	
MXCSR.ue	FSR.ue	36	SSE Underflow Exception	
MXCSR.pe	FSR.pe	37	SSE 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, "IA-32 Floating-point Environment" for details
b. Tag encodings indicate whether each IA-32 numeric register contains an zero, NaN, Infinity or Denormal are not supported by reads of FSR by Itanium instructions. 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 Itanium architecture-based code, the value returned is a summary of any unmasked pending exceptions contained in the FSR, IE, DE, ZE, OE, UE, and PE 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 1 by Itanium architecture-based 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, and PE.
- When FSR.es is written with inconsistent state with respect to the FSR bits (IE, DE, ZE, OE, and PE), subsequent numeric exceptions may report inconsistent floating-point status bits.

For Itanium instructions, the implementation can either raise Reserved Register/Field faults on non-zero writes to the reserved fields, or write the value and return the last value written on reads. FSR, FDR, and FIR must be preserved across a context switch to generate and accurately report numeric exceptions.

Figure 6-10. Floating-point Data Register (FDR)

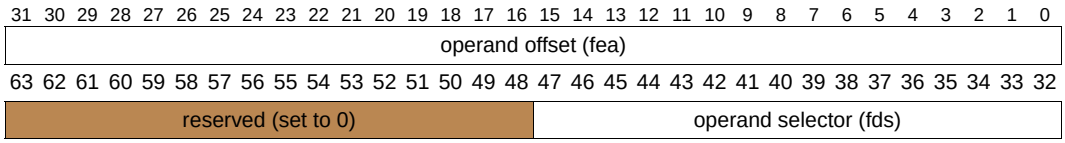
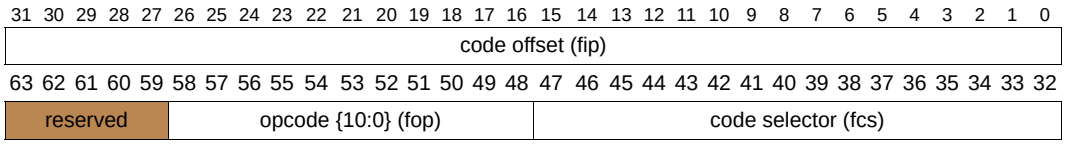


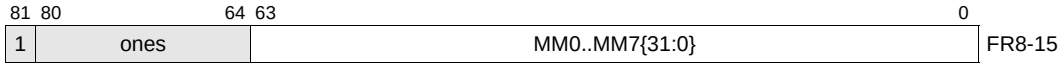
Figure 6-11. Floating-point Instruction Register (FIR)



6.2.6 IA-32 Intel® MMX™ Technology Registers

The eight IA-32 Intel MMX technology registers are mapped on the eight Itanium floating-point 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 6-12. IA-32 Intel® 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 an 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.

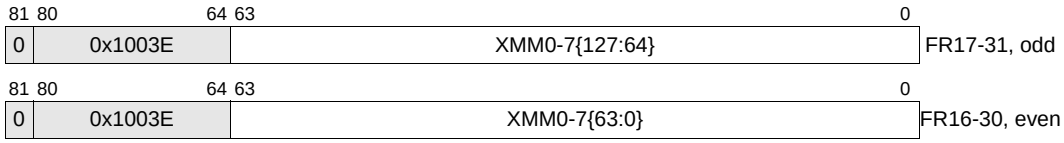
As a result of this mapping, the mantissa of a floating-point value written by either IA-32 or Itanium floating-point instructions will also appear in an IA-32 MMX technology register. An IA-32 MMX technology register will also appear in one of the eight mapped floating-point register's mantissa field.

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

6.2.7 IA-32 SSE Registers

The eight 128-bit IA-32 SSE registers (XMM0-7) are mapped on sixteen physical Itanium floating-point 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 6-13. SSE Registers (XMM0-XMM7)

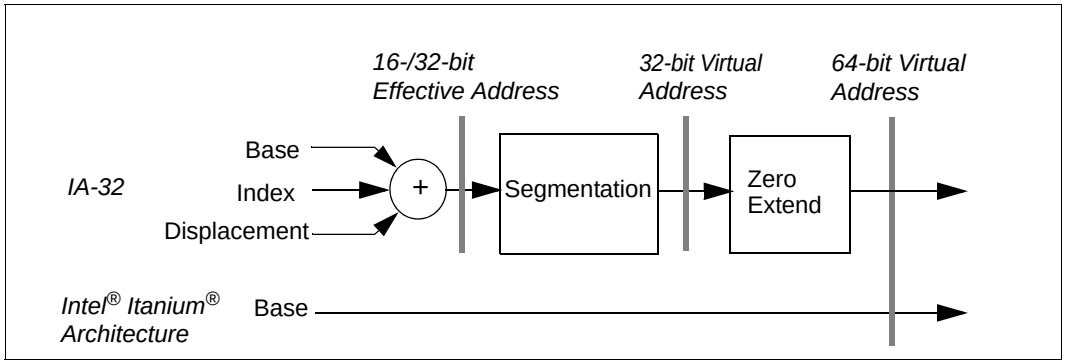


- When a value is written to an SSE register using IA-32 SSE instructions:
  - The exponent field of the corresponding Itanium floating-point register (bits 80-64) is set to 0x1003E and the sign bit (bit 81) is set to 0.
  - The mantissa (bits 63-0) is set to the XMM data value bits{63:0} for even registers and bits{127:64} for odd registers.
- When a SSE register is read using IA-32 SSE instructions:
  - The exponent field of the corresponding Itanium 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 Itanium or IA-32 instruction set are defined to address the same physical memory location. Itanium 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 within virtual region 0. Virtual addresses generated by either instruction set are then translated into physical addresses using memory management mechanisms defined in Chapter 4, "Addressing and Protection" in Volume 2.

Figure 6-14. Memory Addressing Model



6.3.1 Memory Endianess

Memory integer and floating-point (IEEE) data types are binary compatible between the IA-32 and Itanium instruction sets. Itanium architecture-based 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 Itanium 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 *IA-32 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 *IA-32 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 *IA-32 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 within virtual region 0.

If IA-32 code is utilizing a flat segmented model (segment bases are set to zero) then IA-32 and Itanium architecture-based 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 Itanium architecture-based 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.

It is undefined whether the processor will detect a IA-32 self modifying code event for the following conditions; 1) PSR.dt or PSR.it is 0, or 2) there are virtual aliases to different physical addresses between the instruction and data TLBs. To ensure self modifying code works correctly for IA-32 applications, the operating system must ensure that there are no virtual aliases to different physical addresses between the instruction and data TLBs.

When switching from the Itanium instruction set to the IA-32 instruction set, and while executing Itanium 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 Itanium instructions, Itanium architecture-based code must explicitly synchronize the instruction caches with the code sequence defined in "Memory Consistency" on page 1:67. Otherwise the modification may or may not be observed by subsequent IA-32 instructions.

When switching from the IA-32 to the Itanium 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 Itanium instruction fetches see the modification.

6.3.4 Memory Ordering Interactions

IA-32 instructions are mapped into the Itanium memory ordering model as follows:

- All IA-32 stores have ***release*** semantics
- All IA-32 loads have ***acquire*** semantics
- All IA-32 read-modify-write or lock instructions have ***release*** and ***acquire*** semantics (fully fenced).

Instruction set transitions do not automatically fence memory data references. To ensure proper ordering software needs to take into account the following ordering rules.

Transitions from Itanium instruction set to IA-32 instruction set

- All data dependencies are honored, IA-32 loads see the results of all prior Itanium stores
- IA-32 stores (*release*) can not pass any prior Itanium load or store

- IA-32 loads (*acquire*) can pass prior Itanium unordered loads or any prior Itanium store to a different address. Itanium architecture-based software can prevent IA-32 loads from passing prior Itanium loads and stores by issuing an *acquire* operation (or `mf`) before the instruction set transition.

Transitions from IA-32 instruction set to Itanium instruction set

- All data dependencies are honored, Itanium loads see the results of all prior IA-32 stores
- Itanium stores or loads can not pass prior IA-32 loads (*acquire*)
- Itanium unordered stores or any Itanium load can pass prior IA-32 stores (*release*) to a different address. Itanium architecture-based software can prevent Itanium loads and stores from passing prior IA-32 stores by issuing a *release* operation (or `mf`) after the instruction set transition.

6.4 IA-32 Usage of Intel® Itanium® Registers

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

6.4.1 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 via either the `br.ia` or `rfi`. Any dirty registers left in the current and prior register stack frames are left in an undefined state. Software can not rely on the value of these registers across an instruction set transition.

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 ALAT

IA-32 instruction set execution leaves the contents of the ALAT undefined. Software cannot rely on ALAT state being preserved across an instruction set transition. On entry to IA-32 code, existing entries in the ALAT are ignored. For details on the ALAT, refer to Section 4.4.5.2, “Data Speculation and Instructions” on page 1:59.

6.4.3 NaT/NaTVal Response for IA-32 Instructions

If Itanium architecture-based code sets a NaT condition in the integer registers or a NaTVal condition in a floating-point register, MMX technology, or SSE register before switching to the IA-32 instruction set the following conditions can arise:

- When the IA-32 instruction set is entered, NaT values must not be contained in any register defined to contain IA-32 state, otherwise processor operation is model specific and undefined.

Processors may generate a NaT Register Consumption Abort on any IA-32 instruction at any time (including the first IA-32 instruction) for all IA-32 integer, MMX technology, SSE, or FP instructions regardless of whether not that instruction directly (or indirectly) references a register containing a NaT. NaT Register Consumption aborts encountered during IA-32 execution may terminate IA-32 instructions in the middle of execution with architectural state already modified.

- 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 cannot 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 Itanium denormals.

- IA-32 SSE 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.

Part II: Optimization Guide for the Intel® Itanium® Architecture

About the Optimization Guide 1

The second portion of this document explains in detail optimization techniques associated with the Itanium instruction set. It is intended for those interested in furthering their understanding of 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 assembly language programming for the Itanium architecture.

Note: To demonstrate techniques, this guide contains code examples that are not targeted towards a specific processor based on the Itanium architecture, 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

1.1 Overview of the Optimization Guide

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

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

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

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

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

Introduction to Programming for the Intel® Itanium® Architecture 2

2.1 Overview

The Itanium 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, the Itanium architecture does not require that code be written for a specific microarchitecture implementation in order to be functional.

The Itanium architecture 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.

The architecture also enables high floating-point performance and provides direct support for multimedia applications.

Complete descriptions of the syntax and semantics of Itanium instructions can be found in Volume 3: Instruction Set Reference. Though this chapter provides a high level introduction to application level programming, it assumes prior experience with assembly language programming as well as some familiarity with the Itanium application architecture. Optimization is explored in other chapters of this guide.

2.2 Registers

The 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 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 Itanium 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.

2.3 Using Intel® Itanium® Instructions

Itanium instructions are grouped into 128-bit *bundles* of three instructions. Each instruction occupies the first, second, or third *slot* of a bundle. Instruction format, expression of parallelism, and bundle specification are described below.

2.3.1 Format

A basic Itanium 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 Itanium instructions can be accompanied by a qualifying predicate.

mnemonic Specifies a name that uniquely identifies an Itanium 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 Itanium instructions have at least two input source operands.

2.3.2 Expressing Parallelism

The Itanium architecture 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 dependencies. Instruction groups are delimited by *stops* in the assembly source code. Since instruction groups have no RAW or WAW register dependencies, they can be issued without hardware checks for register dependencies 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 dependencies 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 implementations may not have sufficient resources to issue all instructions in an instruction group.

2.3.3 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 processors based on the Itanium architecture to dispatch instructions with simple instruction decoding, and stops enable explicit specification of parallelism.

There are five slot types (M, I, F, B, and L), six 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 slot and one without. Instructions must be placed in slots corresponding to their instruction types based on the template specification, except for A-type instructions that can go in either I or M slots. 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.

2.4 Memory Access and Speculation

The Itanium architecture provides memory access only through register load and store instructions and special semaphore instructions. The architecture also provides extensive support for hiding memory latency via programmer-controlled speculation.

2.4.1 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.

2.4.2 Speculation

Speculation allows a programmer to break data or control dependencies that would normally limit code motion. The two kinds of speculation are called control speculation and data speculation. This section summarizes speculation in the Itanium architecture. See Chapter 3, “Memory Reference” for more detailed descriptions of speculative instruction behavior and application.

2.4.3 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 the Itanium architecture, 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`.

2.4.4 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 assembly sequence:

```
st8 [r55]=r45  // Cycle 0
ld8 r3=[r5] ;; // Cycle 0
shr r7=r3,r87  // Cycle 2
```

The Itanium architecture 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.

2.5 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 the Itanium architecture using qualifying predicates so that branches are removed. The 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 4, “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.

2.6 Architectural 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 Itanium general registers are organized as a logically infinite set of stack frames that are allocated from a finite pool of physical registers.

2.6.1 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`.

2.6.2 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.

2.7 Branches and Hints

Since branches have a major impact on program performance, the Itanium architecture 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.

2.7.1 Branch Instructions

Branching in the Itanium architecture 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. The architecture 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.

2.7.2 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, the Itanium architecture 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 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, the architecture 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, see Chapter 5, “Software Pipelining and Loop Support.”

2.7.3 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.

2.8 Summary

The Itanium architecture 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 3

3.1 Overview

Memory latency is a major factor in determining the performance of integer applications. In order to help reduce the effects of memory latency, the Itanium architecture explicitly supports software pipelining, large register files, and compiler-controlled speculation. This chapter discusses features and optimizations related to compiler-controlled speculation. See Chapter 5, "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 the Itanium architecture, and general concepts and terminology related to data dependencies. The concept of speculation is then introduced, followed by discussions and examples of how speculation is used. The remainder of this chapter describes several important optimizations related to memory access and instruction scheduling.

3.2 Non-speculative Memory References

The Itanium architecture supports non-speculative loads and stores, as well as explicit memory hint instructions.

3.2.1 Stores to Memory

Itanium integer store instructions 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 the Itanium architecture 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).

3.2.2 Loads from Memory

Itanium integer load instructions 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 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.

3.2.3 Data Prefetch Hint

The `lfetch` instruction requests that lines be moved between different levels of the memory hierarchy. Like all hint instructions defined in the Itanium architecture, `lfetch` has no effect on program correctness, and any microarchitecture implementation may choose to ignore it.

3.3 Instruction Dependencies

Data and control dependencies are fundamental factors in optimization and instruction scheduling. Such dependencies 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 dependencies that cannot be broken due to getting a wrong answer (if a data dependency is broken) or raising a fault that should not be raised (if a control dependency is broken). This section describes:

- Background material on memory reference dependencies.
- Descriptions of how dependencies constrain code scheduling on traditional architectures.

Section 3.4 describes memory reference features defined in the Itanium architecture that increase the number of dependencies that can be removed by a compiler.

3.3.1 Control Dependencies

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 dependencies and their effects on optimization.

3.3.1.1 Instruction Scheduling and Control Dependencies

The code below contains a control dependency at the branch instruction:

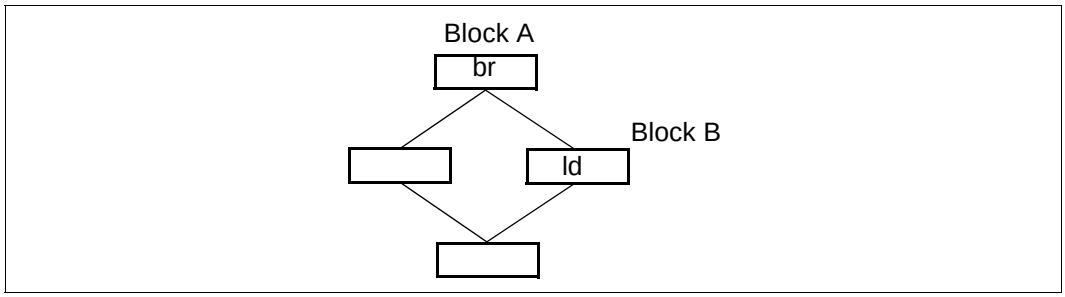
```
  add r7=r6,1            // Cycle 0
  add r13=r25,r27
  cmp.eq p1,p2=r12,r23
(p1)br.cond  some_label ;;

  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 Figure 3-1, the load in block B cannot be moved up because of a conditional branch at the end of block A.

Figure 3-1. Control Dependency Preventing Code Motion



3.3.2 Data Dependencies

A data dependency exists between an instruction that accesses a register or memory location and another instruction that alters the same register or location.

3.3.2.1 Basics of Data Dependency

The following basic terms describe data dependencies between instructions:

Write-after-write (WAW)
: A dependency between two instructions that write to the same register or memory location.

Write-after-read (WAR)
: A dependency between two instructions in which an instruction reads a register or memory location that a subsequent instruction writes.

Read-after-write (RAW)
: A dependency between two instructions in which an instruction writes to a register or memory location that is read by a subsequent instruction.

Ambiguous memory dependencies
: Dependencies 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 dependencies.

Independent memory references
: References by two or more memory instructions that are known not to have conflicting memory accesses.

3.3.2.2 Data Dependency in the Intel® Itanium® Architecture

The Itanium architecture requires the programmer to insert stops between RAW and WAW *register* dependencies 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, implementations based on the Itanium architecture are required to observe *memory*-based dependencies 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 dependency 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 dependency 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]
```

3.3.2.3 Instruction Scheduling and Data Dependencies

The dependency 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 dependency 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 dependency. Stores will cause data dependencies 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;
```

3.4 Using Speculation in the Intel® Itanium® Architecture to Overcome Dependencies

Both data and control dependencies constrain optimization of program code. The Itanium architecture provides support for two basic techniques used to overcome dependencies:

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.

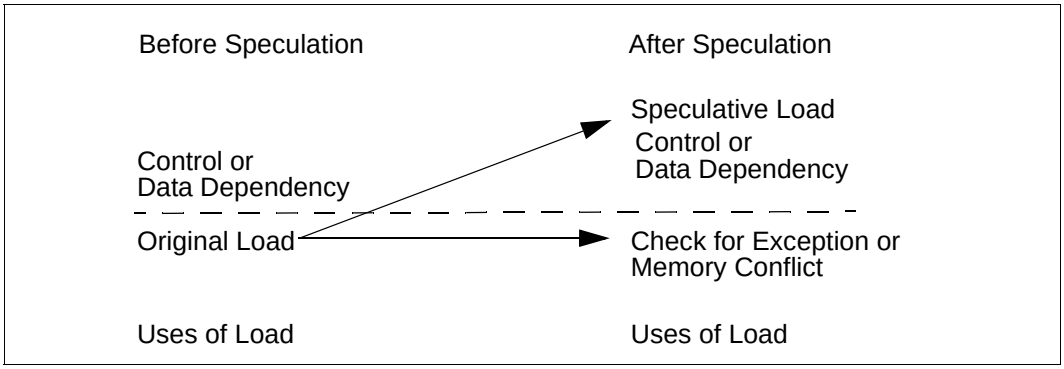
3.4.1 Speculation Model in the Intel® Itanium® Architecture

The limitations imposed by dependencies on instruction scheduling can be solved by separating the loading of data from the exception handling or the acknowledgment of data conflicts. The Itanium architecture 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 Itanium speculative load can be moved above a dependency barrier (shown as a dashed line) as shown in Figure 3-2.

Figure 3-2. Speculation Model in the Intel® Itanium® Architecture



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.

3.4.2 Using Data Speculation in the Intel® Itanium® Architecture

Data speculation in the Itanium architecture 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 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.

3.4.2.1 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.

3.4.2.2 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.

3.4.2.3 Terminology Review

Terms related to speculation, such as *advanced loads* and *check loads*, have well-defined meanings in the Itanium architecture. 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.

3.4.3 Using Control Speculation in the Intel® Itanium® Architecture

The check to determine if control speculation was successful is similar to that for data speculation.

3.4.3.1 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.

3.4.3.2 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 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 3.5.6.

3.4.3.3 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.

3.4.3.4 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.

3.4.4 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.

3.5 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. The Itanium architecture 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.

3.5.1 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 3.5.3 for information on considerations for recovery code placement.

3.5.2 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.

3.5.3 Optimizing Code Size

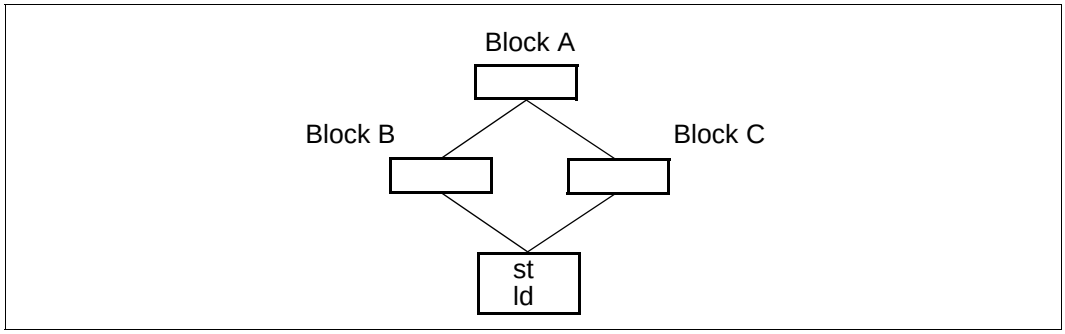
Part of the decision of when to speculate should involve consideration of any possible increases in code size. *Such consideration is not particular to speculation, but to any transformations that cause code to be duplicated, such as loop unrolling, procedure inlining, or tail duplication.* Techniques 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 in Figure 3-3 and the explanation shows how this could arise.

Figure 3-3. Minimizing Code Size During Speculation



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.

3.5.4 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 dependency 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 dependencies 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.

3.5.5 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).

3.5.6 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
```

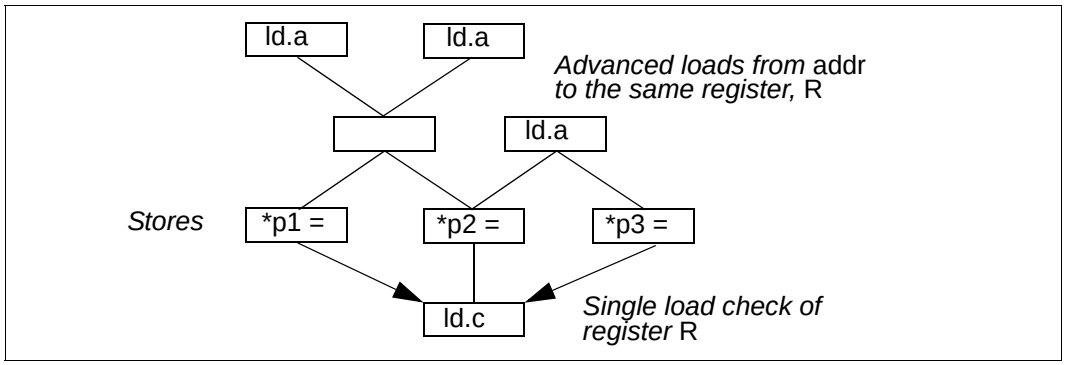
```
// Other instructions

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 Figure 3-4 demonstrates where this technique might be applied.

Figure 3-4. Using a Single Check for Three Advanced Loads



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`.

3.6 Summary

The examples in this chapter show where the Itanium architecture can take advantage of existing techniques like dynamic profiling and disambiguation. Special architectural 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 the 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 4

4.1 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.

4.2 Predication

Predication allows the compiler to convert control dependencies into data dependencies. This section describes several sources of branch-related performance considerations, followed by a summary of predication mechanism, followed by a series of descriptions of optimizations and techniques based on predication.

4.2.1 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.

4.2.1.1 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 Itanium architecture-based 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.

4.2.1.2 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 dependencies, 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 Itanium architecture-based 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.

4.2.2 Predication in the Intel® Itanium® Architecture

Now that the performance implications of branching have been described, this section overviews predication in the Itanium architecture – the primary mechanism used by optimizations described in this section.

Almost all Itanium 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:, "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, the architecture 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
```

4.2.3 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.

4.2.3.1 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 Itanium architecture-based assembly code 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.

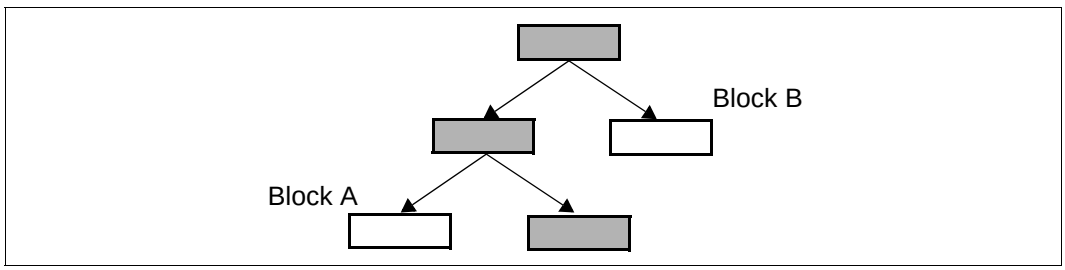
In this case, predication saves an average of three cycles.

4.2.3.2 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 Figure 4-1. Although blocks A and B are not on the main trace, suppose they are executed a significant number of times.

Figure 4-1. Flow Graph Illustrating Opportunities for Off-path Predication



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.

4.2.3.3 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 dependency height. This is possible because predicating an instruction replaces a control dependency with a data dependency. If the data dependency is less constraining than the control dependency, such a transformation may improve the instruction schedule.

Given the Itanium architecture-based 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).

4.2.3.4 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 dependencies, 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.

4.2.3.5 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 control or data speculation. Therefore, when the use of predication yields the same critical path length as data or control speculation, it is almost always preferable to use predication.

4.2.4 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.

4.2.4.1 Unbalanced Execution Paths

The simple conditional below has an unbalanced flow-dependency 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 Itanium architecture-based 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.

4.2.4.2 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.

4.2.4.3 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.

4.2.4.4 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.

4.2.4.5 Overlapping Resource Usage

Before performing if-conversion, the programmer must consider the execution resources consumed by predicated blocks in addition to considering flow-dependency 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 below.

4.2.4.6 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 dependency heights of the branch paths are equal.

4.2.5 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 dependency 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-dependeny 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.

4.3 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. The Itanium architecture 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.

4.3.1 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 implementations based on the Itanium architecture, 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, the Itanium architecture has special *parallel compare* instructions to optimize expressions that have `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, the architecture 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:, "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 dependencies 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` `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.

4.3.2 Reducing Critical Path with Multiway Branches

While there are no special instructions to support branches with multiple conditions and multiple targets, the Itanium architecture has implicit support by allowing multiple consecutive B-slot instructions within an instruction group.

An example uses a basic block with four possible successors. The following Itanium architecture-based 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.

4.3.3 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.

4.3.3.1 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.

The Itanium architecture 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 4.3.1.

4.3.3.2 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 defined in the architecture, 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.

4.3.4 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.

4.3.4.1 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.

4.4 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 I-cache accesses and branch prediction.

Two types of branch-related hints are defined by the Itanium 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 I-cache 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 static hints to be tailored for usage patterns that may not be fully known at compilation time or when the binaries are first distributed.

4.5 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 the performance of the Itanium architecture. Unfortunately, discussion of compiler algorithms of this nature are far beyond the scope of this document.

Software Pipelining and Loop Support 5

5.1 Overview

The Itanium architecture 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. Specific loop support features of the Itanium architecture are then introduced. The remainder of this chapter describes the programming and optimization of various type of loops.

5.2 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 the architecture.

The Itanium architecture 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.

5.3 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 loop support features in the Itanium architecture that are explored later in this chapter. The loop above will be used as a running example in the next few sections.

5.3.1 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 dependencies. The code below shows the loop from our example on page 1:179 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 1:179, 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.

5.3.2 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 1:179 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 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.

5.4 Loop Support Features in the Intel® Itanium® Architecture

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 loop support features in the Itanium architecture 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.

5.4.1 Register Rotation

Register rotation renames registers by adding the register number to the value of a register rename 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, its 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 1:179 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 one 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.

5.4.2 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`.

5.4.3 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.

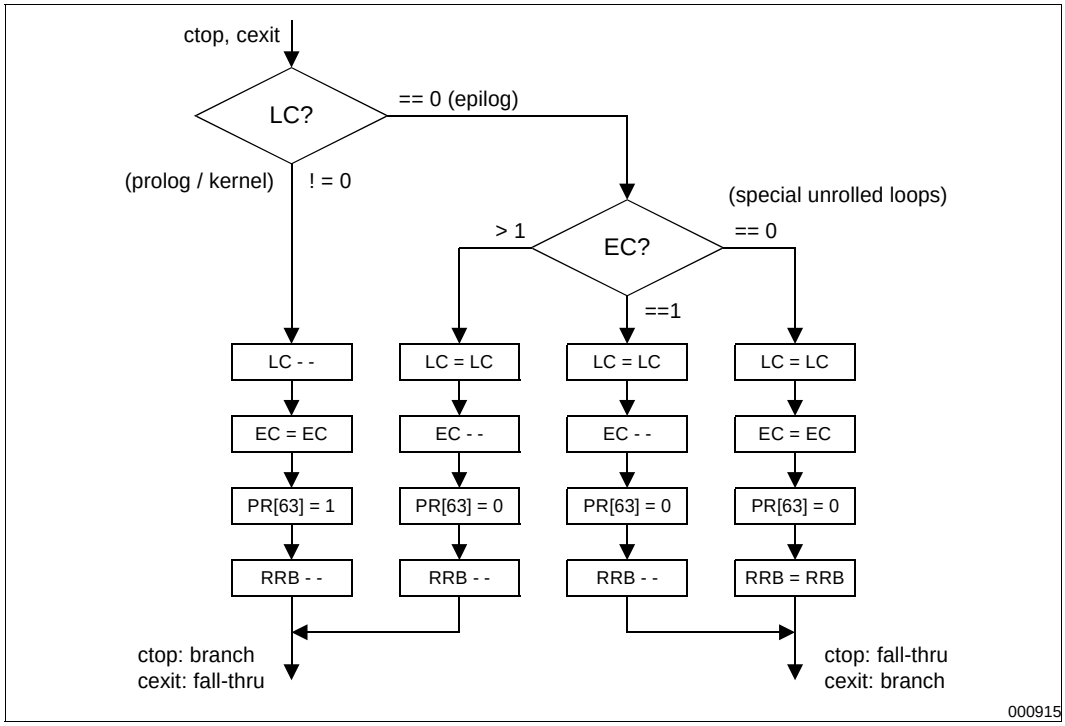
5.4.3.1 Counted Loop Branches

Figure 5-1 shows a flowchart for modulo-scheduled counted loop branches.

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.

Figure 5-1. `ctop` and `cexit` Execution Flow



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.

5.4.3.2 Counted Loop Example

A conceptual view of a pipelined iteration of the example counted loop on page 1:179 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. Table 5-1 shows a trace of the execution of this loop.

Table 5-1. `ctop` Loop Trace

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.

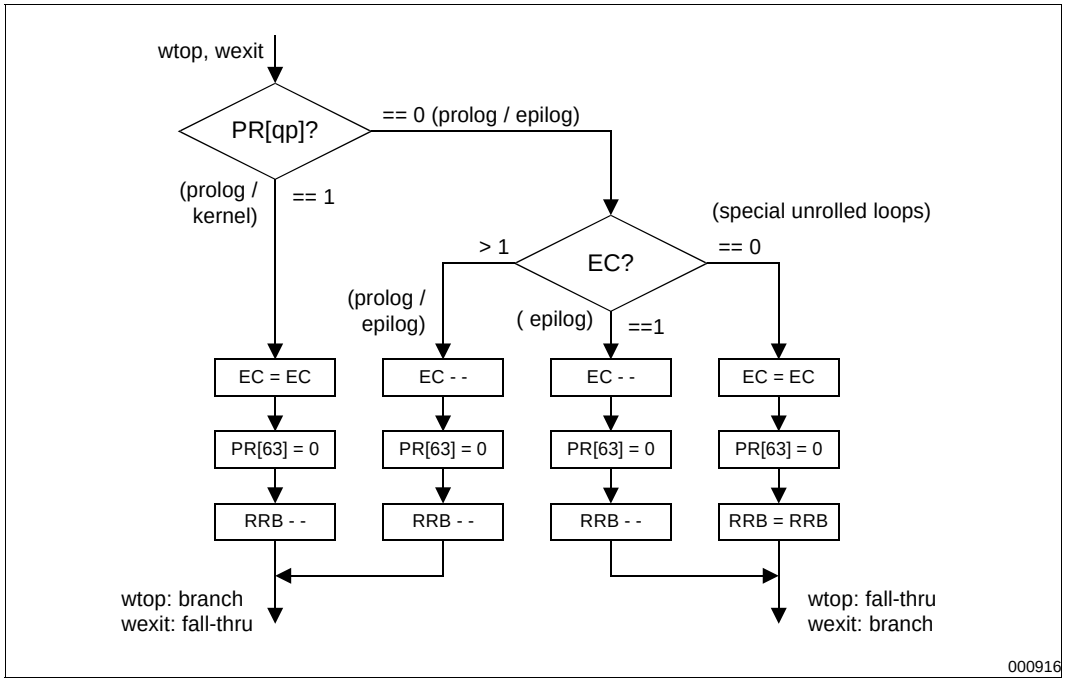
It is desirable to allocate variables that are loop variant to the rotating portion of the register file whenever possible to preserve space in the static portion for loop invariant variables. Induction variables that are post incremented must be allocated to the static portion of the register file.

5.4.3.3 While Loop Branches

Figure 5-2 shows the flowchart for while loop branches.

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.

Figure 5-2. `wtop` and `wexit` Execution Flow



5.4.4 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 register rename base that is decremented.

Induction Variable
: Value that is incremented (or decremented) once per source iteration by the same amount.

5.5 Optimization of Loops in the Intel® Itanium® Architecture

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 dependency 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.

5.5.1 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 the Itanium architecture, 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 1:188 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 Table 5-2.

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.

Table 5-2. `wtop` Loop Trace

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.

5.5.2 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;;
```

5.5.3 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.

5.5.3.1 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 dependency 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.

5.5.3.2 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.

5.5.4 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.

5.5.5 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 3, “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.

5.5.5.1 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;;
```

5.5.5.2 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.

5.5.6 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 one 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 considerations 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.

5.5.7 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 that 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
```

5.5.8 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 that, 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 dependency 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 that 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.sr40, 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 1:185 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.

5.5.9 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 register rotation feature in the Itanium architecture 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;;
```

5.6 Summary

The examples in this chapter show how features in the Itanium architecture 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 6

6.1 Overview

The Itanium floating-point architecture is fully ANSI/IEEE-754 standard compliant and 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, architectural 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 these features.

6.2 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.

6.2.1 Execution Latency

Loops often contain recurrence relationships. Consider the tri-diagonal elimination kernel from the Livermore Fortran Kernel suite.

```
DO 5 i = 2, N
   5X[i] = Z[i] * (Y[i] - X[i-1])
```

The dependency 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 dependency 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
   1X[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.

6.2.2 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).

6.2.3 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 dependencies. 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.

6.2.4 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.

6.3 Floating-point Features in the Intel® Itanium® Architecture

This section highlights architectural features that reduce the impact of the performance limiters described in Section 6.2 using illustrative examples.

6.3.1 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.

The Itanium architecture 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), two 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, 2
      DO 1 j = 1, P, 2
          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.s0   f9  = f5, f7, f9      // *,+ on C[i,j]
  fma.s0 f10  f6, f7, f10        // *,+ on C[i+1,j]
  fma.s0 f11 = f5, f8, f11   // *,+ on C[i,j+1]
  fma.s0  f12 = f6, f8, 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
  1A[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 5, “Software Pipelining and Loop Support” for details on how to rewrite loops using this transformation.

6.3.1.1 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.

6.3.2 Multiply-Add Instruction

The Itanium architecture 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 dependency 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 dependency between the two statements on variable `B5(k+KB5I))` and a loop-carried dependency 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.

6.3.3 Software Divide/Square Root Sequence

To perform division or square root operations on the Itanium architecture, 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.

6.3.3.1 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

6.3.3.2 Double Precision – Square Root

Square Root (Max Throughput)[a] (14 Instructions, 10 Groups)	Square Root (Min Latency)[b] (17 Instructions, 9 Groups)
frsqrta.s0 f7,p6=f6;; (p6) fma.s1 f8=f10,f7,f0 (p6) fma.s1 f7=f6,f7,f0;; (p6) fnma.s1 f9=f7,f8,f10;; (p6) fma.s1 f8=f9,f8,f8 (p6) fma.s1 f7=f9,f7,f7;; (p6) fnma.s1 f9=f7,f8,f10;; (p6) fma.s1 f8=f9,f8,f8 (p6) fma.s1 f7=f9,f7,f7;; (p6) fnma.s1 f9=f7,f8,f10;; (p6) fma.s1 f8=f9,f8,f8 (p6) fma.d.s1 f7=f9,f7,f7;; (p6) fnma.s1 f9=f7,f7,f6;; (p6) fma.d.s0 f7=f9,f8,f7;;	frsqrta.s0 f7,p6=f6;; (p6) fma.s1 f8=f9,f7,f0 (p6) fma.s1 f7=f6,f7,f0;; (p6) fnma.s1 f9=f7,f8,f9;; (p6) fma.s1 f10=f11,f9,f10 (p6) fma.s1 f11=f9,f9,f0 (p6) fma.s1 f12=f13,f9,f12;; (p6) fma.s1 f10=f11,f10,f9 (p6) fma.s1 f11=f11,f11,f0 (p6) fma.s1 f9=f9,f12,f14;; (p6) fma.s1 f12=f10,f7,f7 (p6) fma.s1 f7=f7,f11,f0 (p6) fma.s1 f10=f11,f9,f10;; (p6) fma.d.s1 f7=f9,f7,f12 (p6) fma.s1 f8=f10,f8,f8;; (p6) fnma.s1 f9=f7,f7,f6;; (p6) fma.d.s0 f7=f9,f8,f7;;

a. The following value is assumed preset: f10=1/2.
b. The following values are assumed preset: f9=1/2, f10=3/2, f11=5/2, f12=63/8, f13=231/16, f14=35/8.

For divide, 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 its flags are all discarded.

When divide and square-root operations 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.

6.3.4 Computational Models

The Itanium architecture 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, higher performance, or both.

The register file format is uniform for the three 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 is 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 IA-32 FP computational style (range=extended, precision=single/double/extended) can also achieved using the status field bits.

6.3.5 Multiple Status Fields

The FPSR is divided into one main (architectural) status field and three additional identical status fields. These additional status fields could be used to performance advantage.

First, divide and square-root sequences (described in Section 6.3.3) 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 or 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
 415IF( j5-n+II   ) 430,425,425
 420IF( j5-n+LB   ) 435,415,415
 425IF( PLAN(j5)-R) 445,480,440
 430IF( PLAN(j5)-S) 445,480,440
 435IF( PLAN(j5)-T) 445,480,440
 440IF( ZONE(j4-1)) 455,485,470
 445IF( ZONE(j4-1)) 470,485,455
 450k3= 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
 455m= m+1
      IF( m-ZONE(1) ) 465,465,460
460m= 1
465IF( 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 results (and flags) of the speculatively computed operations (of the conditional after statement 450) can be discarded.

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. One such 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.

6.3.6 Other Features

The Itanium architecture offers a number of other architectural constructs to enhance the performance of different computational situations.

6.3.6.1 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 for divide) 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.EQ. NATVAL OR
      A.EQ. SNAN OR A.EQ. QNAN OR
      A.EQ. NEG_INF OR A.EQ. POS_INF OR
      A.LT. 0.0D0) THEN
      WRITE (*, "INVALID INPUT OPERAND")
  ELSE
      WRITE (*, "SQUARE-ROOT = ", SQRT(A))
  ENDIF
```

The above conditional can be determined by two fclass instructions as indicated below:

```
  fclass.m   p1, p2 = f2, 0x1E3;;   // Detect NaTVal, NaN, +Inf or -Inf
(p2)fclass.m p1, p2 = f2, 0x01A     // Detect -Norm or -Unorm
```

The resultant complimentary predicates (p1 and p2) can be used to control the `ELSE` and `THEN` statements respectively.

6.3.6.2 Min/Max/AMin/AMax

The Itanium architecture 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(b, a)` 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 instructions are designed to mimic the C statement behavior so that they can be generated by the compiler. They are also not commutative. 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
24IF(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 ;;
```

NaNs in the array (X) will be ignored.

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 ;;
```

NaNs in the array (X) will reset the minimum value.

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. In the next iteration, the new array value (loaded in f6) will become the new minimum.

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.

6.3.6.3 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 2^{63}).

Conversions from signed integers to floating-point and from floating-point to signed or unsigned integers are accomplished by fcvt.xf and fcvt.fx/fcvt.fxu instructions respectively. However, since signed integers are converted directly to their canonical floating-point representations, they do not need to be normalized after conversion.

6.3.6.4 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. The Itanium architecture 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.

6.3.7 Memory Access Control

Recognizing the trend of growing memory access latency, and the implementation costs of high bandwidth, the Itanium architecture incorporates many architectural features to help manage the memory hierarchy and increase performance. As described in Section 6.2, memory latency and bandwidth are significant performance limiters in floating-point applications. The architecture offers features to address both these limitations.

In order to enhance the core bandwidth to the floating-point register file, the architecture defines load-pair instructions. In order to mitigate the memory latency, explicit and implicit data prefetch instructions are defined. In order to maximize the utilization of caches, the architecture 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, the architecture defines machine hints to trigger relevant instruction prefetches.

6.3.7.1 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 that can issue 2 FP instructions per cycle, provides sufficient bandwidth from the second level cache (L2) 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 L2. A common example of such a case is a simple double precision dot product – DDOT:

```
  DO 1 I = 1, N
1C = C + A(I) * B(I)
```

The inner loop consists of two loads (for `A` and `B`) and a multiply-add (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]
      C4 = C4 + A[I+3] * B[I+3]
      C5 = C5 + A[I+4] * B[I+4]
      C6 = C6 + A[I+5] * B[I+5]
      C7 = C7 + A[I+6] * B[I+6]
1 C8 = C8 + A[I+7] * B[I+7]
  C = C1 + C2 + C3 + C4 + C5 + C6 + C7 + C8
```

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.

6.3.7.2 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 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.

6.3.7.3 Allocation Control

Since data accesses have different locality attributes (temporal/non-temporal, spatial/non-spatial), The Itanium architecture 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).

6.4 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 floating-point support in the Itanium architecture beyond the software-pipelining support described in Chapter 5, “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.

Index

Symbols

A

Index

B

C

Index

Index

D

Index

Index

Index

G

H

I

Index

Index

Index

J

K

L

Index

M

Index

N

O

P

Index

Index

Q

R

Index

S

Index

T

U

V

Index

W

X

Z