Intel® Itanium™ Architecture Software Developer's Manual

Volume 2: System Architecture

Revision 2.0

December 2001

Document Number: 245318-003

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

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

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

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

Intel® processors based on the Itanium architecture may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

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

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling1-800-548-4725, or by visiting Intel's website at http://www.intel.com.

Contents

Part I: System Architecture Guide

Part II: System Programmer's Guide

Figures

Part I: System Architecture Guide

Part II: System Programmer's Guide

Tables

Part I: System Architecture Guide

Part II: System Programmer’s Guide

Part I: System 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-based operating systems running IA-32 or Itanium-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™-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-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 multi-processing 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-based operating system 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-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-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 and IA-32 instruction sets, 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 Streaming SIMD Extension Instruction Reference” provides a detailed description of all IA-32 Streaming SIMD Extension 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 *Intel Architecture Software Developer's Manual*.

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

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

Itanium-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™ 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. This document has been re-titled and replaces the *Intel® Itanium™ Architecture Software Developer's Manual, Volume 4: Itanium™ Processor Programmer's Guide.*
- ***Intel® Architecture Software Developer's Manual*** – This set of manuals describes the Intel 32-bit architecture. They are readily available from the Intel Literature Department by calling 1-800-548-4725 and requesting Document Numbers 243190, 243191and 243192.
- ***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-based operating system.
- ***Itanium™ Processor Family System Abstraction Layer Specification*** – This document (Document number 245359) specifies requirements to develop platform firmware for Itanium-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 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). 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).

Date of Revision	Revision Number	Description
		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). 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.

Intel® Itanium™ System Environment 2

As described in Section 2.1 "Operating Environments" in Volume 1, the Itanium architecture features two full operating system environments: the **IA-32 System Environment** supports IA-32 operating systems, and the **Itanium System Environment** supports Itanium-based operating systems. The architectural model also supports a mixture of IA-32 and Itanium-based application code within an Itanium-based operating system.

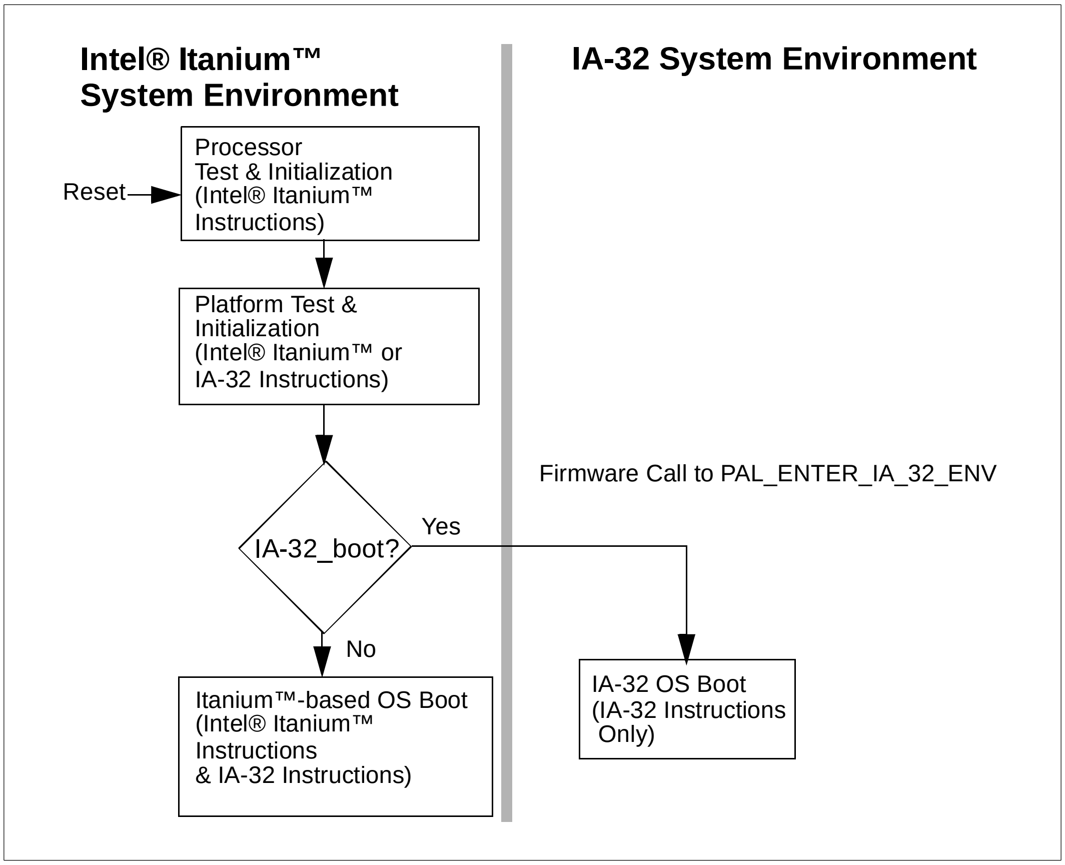
The system environment determines the set of processor system resources seen by the operating system. These resources include: virtual memory management, physical memory attributes, external interrupt mechanisms, exception and interrupt delivery, machine check architectures, debug, performance monitoring, control registers, and the set of privileged instructions.

The choice of system environment is made when a processor boots, and is described in “Processor Boot Sequence”. Section 2.2 in this chapter defines the Itanium System Environment.

2.1 Processor Boot Sequence

Figure 2-1 shows the defined boot sequence. Unlike IA-32 processors, which power up in 32-bit Real Mode, processors in the Itanium processor family power up in the Itanium System Environment running Itanium-based code. Processor initialization, testing, memory, and platform initialization/testing are performed by processor firmware. Mechanisms are provided to execute Real Mode IA-32 boot BIOSs and device drivers during the boot sequence. After the boot sequence, a determination is made by boot software to continue executing in Itanium System Environment (for example to boot an Itanium-based operating systems) or to enter the IA-32 operating system environment through the PAL_ENTER_IA_32_ENV firmware call. Refer to Chapter 11, "Processor Abstraction Layer" for details.

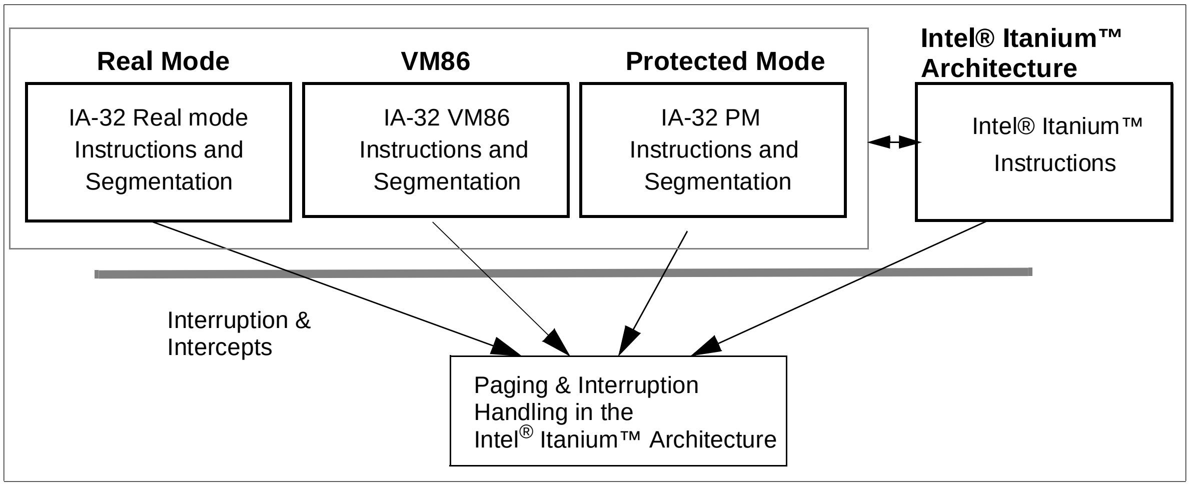
Figure 2-1. System Environment Boot Flow


Intel® Itanium™ System Environment
IA-32 System Environment
Reset
Processor Test & Initialization (Intel® Itanium™ Instructions)
Platform Test & Initialization (Intel® Itanium™ or IA-32 Instructions)
IA-32_boot?
Yes
No
Firmware Call to PAL_ENTER_IA_32_ENV
Itanium™-based OS Boot (Intel® Itanium™ Instructions & IA-32 Instructions)
IA-32 OS Boot (IA-32 Instructions Only)

2.2 Intel® Itanium™ System Environment Overview

The Itanium system environment is designed to support execution of Itanium-based operating systems running IA-32 or Itanium-based applications. IA-32 applications can interact with Itanium-based operating systems, applications and libraries within this environment. Both IA-32 application level code and Itanium instructions can be executed by the operating system and user level software. The entire machine state, including the IA-32 general registers and floating-point registers, segment selectors and descriptors is accessible to Itanium-based code. As shown in Figure 2-2, all major IA-32 operating modes are fully supported.

Figure 2-2. Intel® Itanium™ System Environment


Real Mode
VM86
Protected Mode
Intel® Itanium™ Architecture
IA-32 Real mode Instructions and Segmentation
IA-32 VM86 Instructions and Segmentation
IA-32 PM Instructions and Segmentation
Intel® Itanium™ Instructions
Interruption & Intercepts
Paging & Interruption Handling in the Intel® Itanium™ Architecture

In the Itanium system environment, Itanium architecture operating system resources supersede all IA-32 system resources. Specifically, the IA-32 defined set of control, test, debug, machine check registers, privilege instructions, and virtual paging algorithms are replaced by the Itanium architecture system resources. When IA-32 code is running on an Itanium-based operating system, the processor directly executes all performance critical but non-sensitive IA-32 application level instructions. Accesses to sensitive system resources (interrupt flags, control registers, TLBs, etc.) are intercepted into the Itanium-based operating system. Using this set of intervention hooks, an Itanium-based operating system can emulate or virtualize an IA-32 system resource for an IA-32 application, OS, or device driver.

The Itanium system architecture features are presented in the following chapters:

- Chapter 3 describes system resources.
- Chapter 4 describes the virtual memory architecture.
- Chapter 5 defines the interrupt and exception architecture.
- Chapter 6 describes the register stack engine.
- Chapter 7 describes debug and performance monitoring hooks.
- Chapter 8 describes interruption handler entry points.

Additional support for IA-32 applications in the Itanium system environment is defined by chapters:

- Chapter 9 describes IA-32 interruption handler entry points.
- Chapter 10, "Itanium™-based Operating System Interaction Model with IA-32 Applications"describes how IA-32 applications interact with Itanium-based operating systems.

System State and Programming Model 3

This chapter describes the architectural state visible only to an operating system and defines system state programming models. It covers the functional descriptions of all the system state registers, descriptions of individual fields in each register, and their serialization requirements. The virtual and physical memory management details are described in Chapter 4, “Addressing and Protection”. Interruptions are described in Chapter 5, “Interruptions”.

Note: Unless otherwise noted, references to “interruption” in this chapter refer to IVA-based interruptions. See “Interruption Definitions” on page 1:79.

3.1 Privilege Levels

Four privilege levels, numbered from 0 to 3, are provided to control access to system instructions, system registers and system memory areas. Level 0 is the most privileged and level 3 the least privileged. Application instructions and registers can be accessed at any privilege level. System instructions and registers defined in this chapter can only be accessed at privilege level 0; otherwise, a Privilege Operation fault is raised. The processor maintains a Current Privilege Level (CPL) in the cpl field of the Processor Status Register (PSR). CPL can only be modified by controlled entry and exit points managed by the operating system. Virtual memory protection mechanisms control memory accesses based on the Privilege Level (PL) of the virtual page and the CPL.

3.2 Serialization

For all application and system level resources, apart from the control register file, the processor ensures values written to a register are observed by instructions in subsequent instruction groups. This is termed **data dependency**. For example, writes to general registers, floating-point and application registers are observed by subsequent reads of the same register. (See “Control Registers” on page 1:24 for control register serialization requirements.) For modifications of application level resources with side effects, the side effects are ensured by the processor to be observed by subsequent instruction groups. This is termed **implicit serialization**. Application registers (ARs), with the exception of the Interval Time Counter, the User Mask, when modified by sum, rum, and mov to psr.um, and the Current Frame Marker (CFM), are implicitly serialized. PMD registers have special serialization requirements as described in “Generic Performance Counter Registers” on page 1:138. All other application-level resources (GRs, FRs, PRs, BRs, IP, CPUID) have no side effects and so need not be serialized.

To avoid serialization overhead in privileged operating system code, system register resources are not implicitly serialized. The processor does not ensure modification of registers with side effects are observed by subsequent instruction groups. For system register resources other than control registers, the processor ensures data dependencies are honored (reads see the results of prior writes to the same register). See Section 3.3.3 and Table 3-3 on page 2:24 for control register serialization requirements. This approach simplifies hardware and allows for more efficient software operations.

For example, during a low level context switch where there is no immediate use of loaded system registers, these registers can be loaded without any serialization overhead. To ensure side effects are observed before a dependent instruction is fetched or executed, two serialization operations are provided: **instruction serialization** and **data serialization**.

3.2.1 Instruction Serialization

Instruction serialization ensures that modifications to processor resources are observed before subsequent instruction group fetches are re-initiated. Software must use an instruction serialization operation before any instruction group that is dependent upon the modified system resource. Resource side effects may be observed at any point before the explicit serialization operation.

Modification of the following system resources (if the modification affects instruction fetching) require instruction serialization: RR, PKR, ITR, ITC, IBR, PMC, PMD, PSR bits as defined in "Processor Status Register (PSR)" on page 1:18 and Control Registers as defined in "Control Registers" on page 1:24.

The instructions Return from Interruption (`rfi`) and Instruction Serialize (`srlz.i`) perform explicit instruction serialization.

An interruption performs an implicit instruction serialization operation, so the first instruction group in the interruption handler will observe the serialized state.

```
Instruction Serialization Example:

mov ibr[reg]= reg    // move to instruction debug register
;;                   // end of instruction group
srlz.i               // ensure subsequent instruction fetches observe
                     // modification
;;                   // end of instruction group
inst                 // dependent instruction
```

Note: The serializing instruction, the instruction to be serialized, and any operations dependent on the serialization must be in three separate instruction groups.

3.2.2 Data Serialization

Data serialization ensures that modifications to processor resources affecting both execution and data memory accesses are observed. Software must issue a data serialize operation prior to the instruction dependent upon the modified resource. Data serialization can be issued within the same instruction group as the dependent instruction. Resource side effects may be observed at any point before the explicit serialization operation.

Modification of the following system resources require data serialization: RR, PKR, DTR, DTC, DBR, PMC, PMD, PSR bits as defined in "Processor Status Register (PSR)" on page 1:18 and Control Registers as defined in "Control Registers" on page 1:24.

The control registers are different from the general registers and other registers. Most control registers require an explicit data serialization between the writing of a control register and the reading of that same control register. (See Table 3-3 on page 2:24 for serialization requirements for specific control registers.)

The Data Serialize (`srlz.d`) instruction performs explicit data serialization. Instruction serialization operations (`rfi`, `srlz.i`, and interruptions) also perform a data serialization operation.

```
Data Serialization Example:

mov rr[reg] = reg    //move into region register
;;                   //end of instruction group
srlz.d               //serialize region register modification
ld                   //perform a dependent load
```

The serializing instruction and the instruction to be serialized (the one writing the resource) must be in two different instruction groups. Operations dependent on the serialization and the serialization can be in the same instruction group, but the `srlz` instruction must be before the dependent instruction slot.

3.2.3 Definition of In-flight Resources

When the value of a resource that requires an explicit instruction or data serialization is changed by one or more writers, that resource is said to be **in-flight** until the required serialization is performed. There can be multiple in-flight values if multiple writers have occurred since the last serialization.

An instruction that reads an in-flight resource will see one of the in-flight values or the state prior to any of the unserialized writers. However, whether such a reader sees the original or one of the in-flight values is not predictable.

For a reader of an in-flight resource, this definition includes (but is not limited to) the following possible outcomes:

- The reader of an in-flight resource may see the most-recently-serialized value or any of the in-flight values each time it is executed - seeing the value from a particular writer one time does not guarantee that the same writer's value will be seen by that reader the next time.
- Multiple readers of an in-flight resource may see different values - each may see the most-recently-serialized value or any of the in-flight values, independent of what other readers may see.
- If a single execution of an instruction reads an in-flight resource more than once during its execution, each read may see a different value.

Thus, the only way to guarantee that the latest value is seen by a reader is to perform the required serialization.

3.3 System State

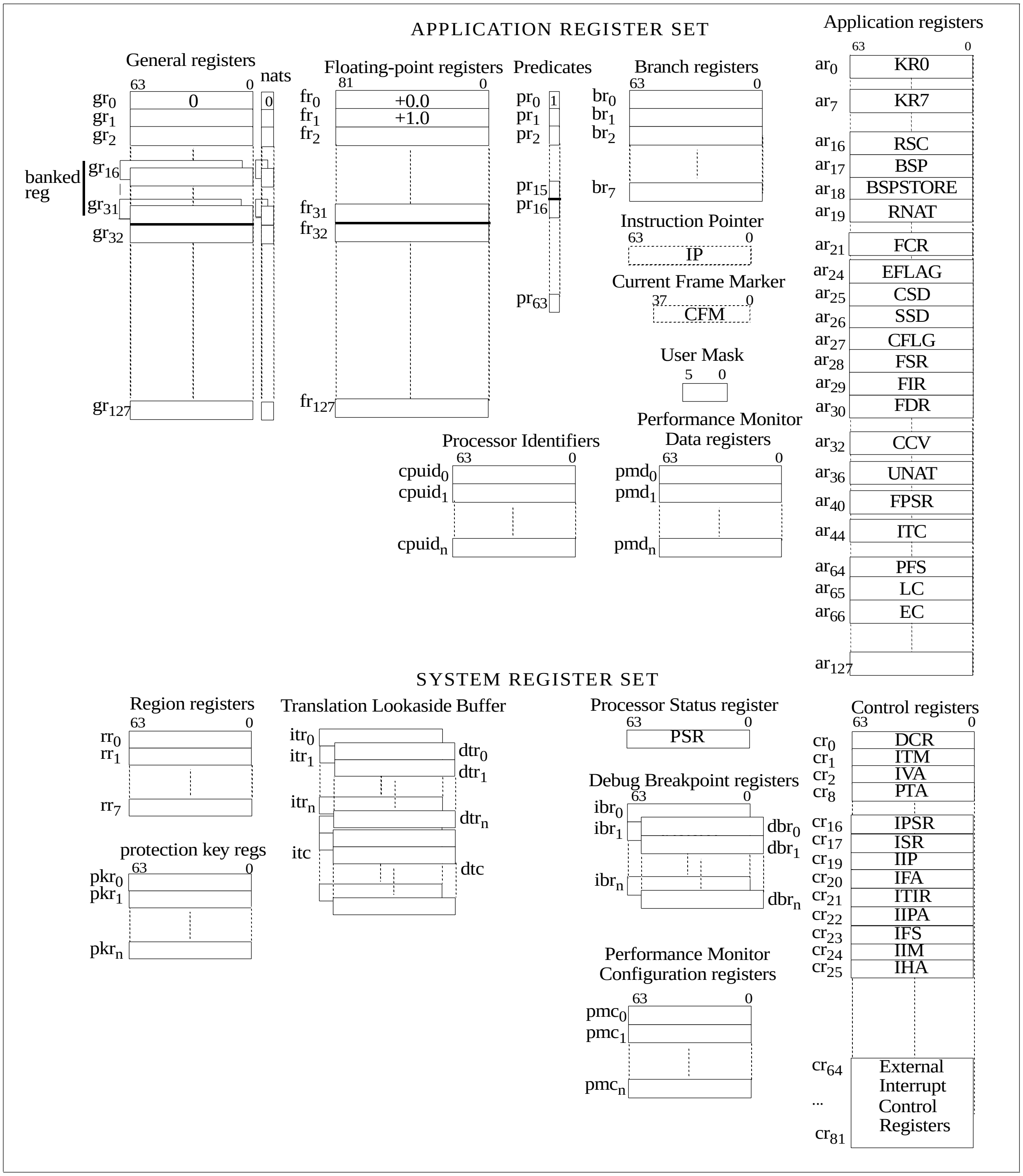
The architecture provides a rich set of system register resources for process control, interruptions handling, protection, debugging, and performance monitoring. This section gives an overview of these resources.

3.3.1 System State Overview

Figure 3-1 shows the set of all defined privileged system register resources. Application state as defined in "Application Register State" on page 1:19 is also accessible.

- **Processor Status Register (PSR)** – 64-bit register that maintains control information for the currently running process. See "Processor Status Register (PSR)" on page 1:18 for complete details.
- **Control Registers (CR)** – This register name space contains several 64-bit registers that capture the state of the processor on an interruption, enable system-wide features, and specify global processor parameters for interruptions and memory management. See "Control Registers" on page 1:24 for complete information.
- **Interrupt Registers** – These registers provide the capability of masking external interrupts, reading external interrupt vector numbers, programming vector numbers for internal processor asynchronous events and external interrupt sources. For complete information, see "Interrupts" on page 1:97.
- **Interval Timer Facilities** – A 64-bit interval timer is provided for privileged and non-privileged use and as a time base for performance measurements. Timing facilities are defined in detail in "Interval Time Counter and Match Register (ITC – AR44 and ITM – CR1)" on page 1:27.
- **Debug Breakpoint Registers (DBR/IBR)** – 64-bit Data and 64-bit Instruction Breakpoint Register pairs (DBR, IBR) can be programmed to fault on reference to a range of virtual and physical addresses generated by either Itanium or IA-32 instructions. See "Debugging" on page 1:133 for details. The minimum number of DBR register pairs and IBR register pairs is 4 in any implementation. On some implementations, a hardware debugger may use two or more of these register pairs for its own use; see "Data and Instruction Breakpoint Registers" on page 1:134 for details.
- **Performance Monitor Configuration/Data Registers (PMC/PMD)** – Multiple performance monitors can be programmed to measure a wide range of user, operating system, or processor performance values. Performance monitors can be programmed to measure performance values from either IA-32 or Itanium instructions. Performance monitors are defined in "Performance Monitoring" on page 1:137. The minimum number of generic PMC/PMD register pairs in any implementation is 4.
- **Banked General Registers** – A set of 16 banked 64-bit general purpose registers, GR 16-GR 31, are available as temporary storage and register context when operating in low level interruption code. See "Banked General Registers" on page 1:34 for complete details.
- **Region Registers (RR)** – Eight 64-bit region registers specify the identifiers and preferred page sizes for multiple virtual address spaces. Refer to "Region Registers (RR)" on page 1:48 for complete information.
- **Protection Key Registers (PKR)** – At least sixteen 64-bit protection key registers contain protection keys and read, write, execute permissions for virtual memory protection domains. Please see the processor specific documentation for further information on the number of Protection Key Registers implemented on the Itanium processor. Refer to "Protection Keys" on page 1:48 for details.

Figure 3-1. System Register Model

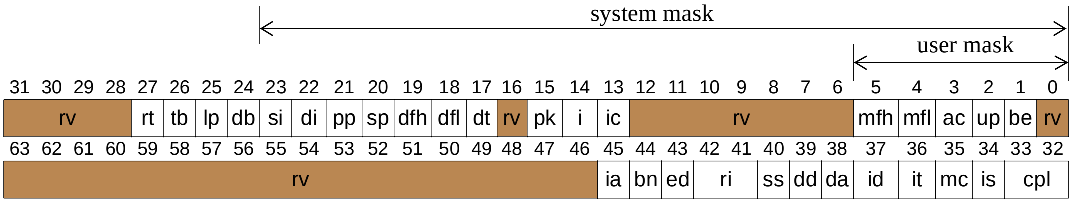


- **Translation Lookaside Buffer (TLB)** – Holds recently used virtual to physical address mappings. The TLB is divided into Instruction (ITLB), Data (DTLB), Translation Registers (TR) and Translation Cache (TC) sections. See "Translation Lookaside Buffer (TLB)" on page 1:39 for complete details. Translation Registers are software managed portions of the TLB and the Translation Cache section of the TLB is directly managed by the processor.

3.3.2 Processor Status Register (PSR)

The PSR maintains the current execution environment. The PSR is divided into four overlapping sections (See Figure 3-2): user mask bits (PSR{5:0}), system mask bits (PSR{23:0}), the lower half (PSR{31:0}), and the entire PSR (PSR{63:0}). PSR fields are defined in Table 3-2 along with serialization requirements for modification of each field and the state of the field after an interruption.

Figure 3-2. Processor Status Register (PSR)



The PSR instructions and their serialization requirements are defined in Table 3-1. These instructions explicitly read or write portions of the PSR. Other instructions also read and write portions of the PSR as described in Table 3-2 and Table 5-2.

Table 3-1. Processor Status Register Instructions

Mnemonic	Description	Operation	Instr. Type	Serialization Required
`sum` *`imm`*	Set user mask from immediate	PSR{5:0} ← PSR{5:0} | *imm*	M	implicit
`rum` *`imm`*	Reset user mask from immediate	PSR{5:0} ← PSR{5:0} & ~*imm*	M	implicit
`mov psr.um =` r_2	Move to user mask	PSR{5:0} ← GR[r_2]	M	implicit
`mov` r_1 `= psr.um`	Move from user mask	GR[r_1] ←PSR{5:0}	M	none
`ssm` *`imm`*	Set system mask from immediate	PSR{23:0} ← PSR{23:0} | *imm*	M	data/inst[a]
`rsm` *`imm`*	Reset system mask from immediate	PSR{23:0} ← PSR{23:0} &~*imm*	M	data/inst[a]
`mov psr.l =` r_2	Move to lower PSR	PSR{31:0} ← GR[r_2]	M	data/inst[a]
`mov` r_1 `= psr`	Move from PSR	GR[r_1] ←PSR{36:35,31:0}[b]	M	none
`bsw.0, bsw.1`	Bank switch	PSR{44} ← 0 or 1	B	implicit
`rfi`	Return From Interruption	PSR{63:0} ← IPSR	B	implicit

a. Based upon the resource being serialized, use data or instruction serialization.
b. All other bits of the PSR read as zero.

The user mask, PSR{5:0}, can be set and cleared by the Set User Mask (`sum`), Reset User Mask (`rum`) and Move to User Mask (`mov psr.um=`) instructions at any privilege level. For user mask modifications by `sum, rum` and `mov`, the processor ensures all side effects are observed before subsequent instruction groups.

The system mask, PSR{23:0}, can be set and cleared by the Set System Mask (`ssm`) and Reset System Mask (`rsm`) instructions. Software must issue the appropriate serialization operation before dependent instructions. The system mask instructions are privileged.

The lower half of the PSR, PSR{31:0}, can be written with the Move to Lower PSR (`mov psr.l=`) instruction. Software must issue the appropriate serialization operation before dependent instructions. The Move to Lower PSR instruction is privileged.

The PSR can be read with the Move from PSR (`mov =psr`) instruction. Only PSR{36:35} and PSR{31:0} are written to the target register by Move from PSR. PSR{63:37} and PSR{34:32} can only be read after an interruption by reading the state in IPSR. The entire PSR is updated from IPSR by the Return from Interruption (`rfi`) instruction. An `rfi` also implicitly serializes the PSR. Both Move from PSR and Return from Interruption are privileged.

Table 3-2. Processor Status Register Fields

Field	Bit	Description	Interruption State	Serialization Required
User Mask = PSR{5:0}				
rv	0	reserved		
be	1	Big-Endian – When 1, data memory references are big-endian. When 0, data memory references are little endian. This bit is ignored for IA-32 data references, which are always performed little-endian. Instruction fetches are always performed little endian.	DCR.be	data[a]
up	2	User Performance monitor enable – When 1, performance monitors configured as user monitors are enabled to count events (including IA-32). When 0, user configured monitors are disabled. See "Performance Monitoring" on page 1:137 for details.	unchanged	data[a] inst[b]
ac	3	Alignment Check – When 1, all unaligned data memory references result in an Unaligned Data Reference fault. When 0, unaligned data memory references may or may not result in a Unaligned Data Reference fault. See "Memory Datum Alignment and Atomicity" on page 1:77 for details. Unaligned semaphore references also result in a Unaligned Data Reference fault, regardless of the state of PSR.ac. For IA-32 instructions, if PSR.ac is 1 an unaligned IA-32 data memory reference raises an IA-32_Exception(AlignmentCheck) fault. When 0, additional IA-32 control bits as defined in Section 10.6.7 also generate alignment checks.	0	data[a]
mfl	4	Lower (f2 .. f31) floating-point registers written – This bit is set to one when an Intel® Itanium™ instruction completes that uses register f2..f31 as a target register. This bit is sticky and only cleared by an explicit write of the user mask. When leaving the IA-32 instruction set, PSR.mfl is set to 1 if PSR.dfl is 0, otherwise PSR.mfl is unmodified.	unchanged	data[a]
mfh	5	Upper (f32 .. f127) floating-point registers written – This bit is set to one when an Intel® Itanium™ instruction completes that uses register f32..f127 as a target register. This bit is sticky and only cleared by an explicit write of the user mask. PSR.mfh is unmodified by IA-32 instruction set execution.	unchanged	data[a]
System Mask = PSR{23:0}				

Table 3-2. Processor Status Register Fields (Continued)

Field	Bit	Description	Interruption State	Serialization Required
ic	13	Interruption Collection – When 1 and an interruption occurs, the current state of the processor is loaded in IIP, IPSR, IIM and IFS; and additional registers defined in "Interruption Vector Descriptions" on page 1:147. When 0, IIP, IPSR, IIM and IFS are not modified on an interruption (see "Writing of Interruption Resources by Vector" on page 1:148 for details). When 0, speculative load exceptions result in deferred exception behavior, regardless of the state of the DCR and ITLB deferral bits. Processor operation is undefined if PSR.ic is 0 and a transition is made to execute IA-32 code.	0	inst/data[c]
i	14	Interrupt Bit – When 1 and executing Intel® Itanium™ instructions, unmasked pending external interrupts will interrupt the processor by transferring control to the external interrupt handler. When 0, pending external interrupts do not interrupt the processor. The effect of clearing PSR.i via Reset System Mask (rsm) instructions is observed by the next instruction. Toggling PSR.i from one to zero via Move to PSR.l requires data serialization. When executing IA-32 instructions, external interrupts are enabled if PSR.i and (CFLG.if is 0 or EFLAG.if is 1). NMI interrupts are enabled if PSR.i is 1 regardless of EFLAG.if.	0	clear: implicit serialization set: data[d]
pk	15	Protection Key enable – When 1 and PSR.it is 1, instruction references (including IA-32) check for valid protection keys. When 1 and PSR.dt is 1, data references (including IA-32) check for valid protection keys. When 1 and PSR.rt is 1, protection key checks are enabled for register stack references. When 0, neither instruction, data, nor register stack references are checked for valid protection keys. When PSR.dt, PSR.rt or PSR.it are 0, PSR.pk is ignored for the corresponding reference.	unchanged	inst/data[e]
rv	12:6, 16	reserved		
dt	17	Data address Translation – When 1, virtual data addresses are translated and access rights checked. When 0, data accesses use physical addressing. PSR.dt must be 1 when entering IA-32 code, otherwise processor operation is undefined.	unchanged/0[j]	data
dfl	18	Disabled Floating-point Low register set – When 1, a read or write access to f2 through f31 results in a Disabled Floating-Point Register fault. When 1, all IA-32 FP, Intel® MMX2 and Intel® MMX™ instructions raise a Disabled FP Register fault (regardless whether the instruction actually references f2-31).	0	data
dfh	19	Disabled Floating-point High register set – When 1, a read or write access to f32 through f127 results in a Disabled Floating-Point Register fault. When 1, a Disabled FP Register fault is raised on the first IA-32 target instruction following a `br.ia` or `rfi`, regardless whether f32-127 are referenced.	0	data

Table 3-2. Processor Status Register Fields (Continued)

Field	Bit	Description	Interruption State	Serialization Required
sp	20	Secure Performance monitors – Controls the ability of non-privileged code (including IA-32 code) to read non-privileged performance monitors. See Table 7-5 on page 2:140 for values returned by PMD read instructions. Also, when 0, PSR.up can be modified by user mask instructions; otherwise, PSR.up is unchanged by user mask instructions. When 1 or CFLG.pce is 0, non-privileged IA-32 performance monitor reads (via `rdpmc`) raise an IA-32_Exception(GPFault).	0	data
pp	21	Privileged Performance monitor enable – When 1, monitors configured as privileged monitors are enabled to count events (including IA-32 events). When 0, privileged monitors are disabled. See “Performance Monitoring” on page 1:137 for details.	DCR.pp	inst/data[e]
di	22	Disable Instruction set transition – When 1, attempts to switch instruction sets via the IA-32 `jmpe` or `br.ia` instructions results in a Disabled Instruction Set Transition fault. This bit doesn’t restrict instruction set transitions due to interruptions or `rfi`.	0	data
si	23	Secure Interval timer – When 1, the Interval Time Counter (ITC) register is readable only by privileged code, non-privileged reads result in a Privilege Operation fault. When 0, ITC is readable at any privilege level. System software can secure the ITC from non-privileged IA-32 access by setting either PSR.si or CFLG.tsd to 1. When secured, an IA-32 rdtsc (read time stamp counter) instruction at any privilege level other than the most privileged raises an IA-32_Exception(GPfault)	0	data
PSR.l = PSR{31:0}				
db	24	Debug Breakpoint fault – When 1, data and instruction address breakpoints are enabled and can cause an Data/Instruction Debug fault. When 1, IA-32 instruction address breakpoints are enabled and can cause an IA-32_Exception(Debug) fault.When 1, IA-32 data address breakpoints are enabled and can cause an IA-32_Exception(Debug) Trap.When 0, address breakpoint faults and traps are disabled.	0	inst/data[e]
lp	25	Lower Privilege transfer trap – When 1, a Lower Privilege Transfer trap occurs whenever a taken branch lowers the current privilege level (numerically increases). This bit is ignored during IA-32 instruction set execution.	0	data
tb	26	Taken Branch trap – When 1, the successful completion of a taken branch results in a Taken Branch trap. `rfi` and interruptions can not raise a Taken Branch trap. When 1, successful completion of a taken IA-32 branch results in an IA-32_Exception(Debug) trap.	0	data

Table 3-2. Processor Status Register Fields (Continued)

Field	Bit	Description	Interruption State	Serialization Required
rt	27	Register stack Translation – When 1, register stack accesses are translated and access rights are checked. When 0, register stack accesses use physical addressing. PSR.dt is ignored for register stack accesses. The register stack engine must be in enforced lazy mode (RSC.mode = 00) when modifying this bit; otherwise, processor behavior is undefined. During IA-32 instruction execution this bit is ignored and the register stack is disabled.	unchanged	data
rv	31:28	reserved		
PSR{63:0}				
cpl[f]	33:32	Current Privilege Level –The current privilege level of the processor (including IA-32). Controls accessibility to system registers, instructions and virtual memory pages. A value of 0 is most privileged, a value of 3 is least privileged. Written by the `rfi`, `epc`, and `br.ret` instructions. PSR.cpl is unchanged by the `jmpe` and `br.ia` instructions. PSR.cpl cannot be updated by any IA-32 instructions.	0	rfi[g]
is	34	Instruction Set – When 0, Intel® Itanium™ instructions are executing. When 1, IA-32 instructions are executing. Written by the `rfi` and `br.ia` instructions and the IA-32 `jmpe` instruction.	0	rfi[g], br.ia[h]
mc	35	Machine Check abort mask – When 1, machine check aborts are masked. When 0, machine check aborts can be delivered (including IA-32 instruction set execution). Processor operation is undefined if PSR.mc is 1 and a transition is made to execute IA-32 code.	unchanged/1[i]	rfi[g]
it	36	Instruction address Translation – When 1, virtual instruction addresses are translated and access rights checked. When 0, instruction accesses use physical addressing. PSR.it must be 1 when entering IA-32 code, otherwise processor operation is undefined.	unchanged/0[j]	rfi[g]
id	37	Instruction Debug fault disable – When 1, Instruction Debug faults are disabled on the first restart instruction in the current bundle.[k] When PSR.id is 1 or EFLAG.rf is 1, IA-32 instruction debug faults are disabled for one IA-32 instruction. PSR.id and EFLAG.rf are set to 0 after the successful execution of each IA-32 instruction.	0	rfi[g]
da	38	Disable Data Access and Dirty-bit faults – When 1, Data Access and Dirty-Bit faults are disabled on the first restart instruction in the current bundle or for the first mandatory RSE reference following the `rfi`.[k] IA-32 Access/Dirty-bit faults are not affected by PSR.da.[l]	0	rfi[g]
dd	39	Data Debug fault disable – When 1, Data Debug faults are disabled on the first restart instruction in the current bundle or for the first mandatory RSE reference.[k] IA-32 Data Debug traps are not affected by PSR.dd.[l]	0	rfi[g]
ss	40	Single Step enable – When 1, a Single Step trap occurs following the successful execution of the first restart instruction in the current bundle. Instruction slots 0, 1, and 2 can be single stepped. When 1 or EFLAG.tf is 1, an IA-32_Exception(Debug) trap is taken after each IA-32 instruction.	0	rfi[g]

Table 3-2. Processor Status Register Fields (Continued)

Field	Bit	Description	Interruption State	Serialization Required
ri	42:41	Restart Instruction – Set on an interruption, indicating the next instruction in the bundle to be executed. When the next instruction is the L+X instruction of an MLX, this field is set to the value 1. When restarting instructions with `rfi,` this field specifies which instruction(s) in the bundle are restarted. The specified and subsequent instructions are restarted, all instructions prior to the restart point are ignored. 0 – restart execution at instruction slot 0 1 – restart execution at instruction slot 1 2 – restart execution at instruction slot 2 3 – reserved Except at an interruption and for the first restart instruction following an `rfi`, the value of this field is undefined. This field is set to 0 after any interruption from the IA-32 instruction set and is ignored when IA-32 instructions are restarted.	instruction pointer	rfi[g]
ed	43	Exception Deferral – When 1, if the first restart instruction in the current bundle is a speculative load, the operation is forced to indicate a deferred exception by setting the load target register to NaT or NaTVal. No memory references are performed, however any address post increments are performed. If the operation is a speculative advanced load, the ALAT entry corresponding to the load address and target register is purged. If the operation is an `lfetch` instruction, memory promotion is not performed, however any address post increments are performed. When 0, exception deferral is not forced on restarted speculative loads. If the first restart instruction is not a speculative load or `lfetch` instruction, this bit is ignored.[kl]	0	rfi[g]
bn	44	register Bank – When 1, registers GR16 to GR31 for bank 1 are accessible. When 0, registers GR16 to GR31 for bank 0 are accessible. Written by `rfi` and `bsw` instructions.	0	implicit[m]
ia	45	Disable Instruction Access-bit faults – When 1, Instruction Access-Bit faults are disabled on the first restart instruction in the current bundle.[k] IA-32 Access-bit faults are not affected by PSR.ia.[l]	0	rfi[g]
rv	63:46	reserved		

a. User mask bits are implicitly serialized if accessed via user mask instructions; `sum`, `rum`, and move to User Mask. If modified with system mask instructions; `rsm`, `ssm` and move to PSR.l, software must explicitly serialize to ensure side effects are observed before dependent instructions.

b. User mask modification serialization is implicit only for monitoring data execution events. Software should issue instruction serialization operations before monitoring instruction events to achieve better accuracy.

c. Requires instruction serialization to guarantee that VHPT walks initiated on behalf of an instruction reference observe the new value of this bit. Otherwise, data serialization is sufficient to guarantee that the new value is observed.

d. The effect of masking external interrupts with `rsm` is observed by the next instruction. However, the processor does not ensure unmasking interruptions with ssm is immediately observed. Software can issue a data serialization operation to ensure the effects of setting PSR.i are observed before a given point in program execution.

e. Requires instruction or data serialization, based on whether the dependent “use” is an instruction fetch access or data access.
f. CPL can be modified due to interruptions, Return From Interruption (`rfi`), Enter Privilege Code (`epc`), and Branch Return (`br.ret`) instructions.
g. Can only be modified by the Return From Interruption (`rfi`) instruction. `rfi` performs an explicit instruction and data serialization operation.
h. Modification of the PSR.is bit by a `br.ia` instruction set is implicitly instruction serialized.
i. PSR.mc is set to 1 after a machine check abort or INIT; otherwise, unmodified on interruptions.
j. After an interruption this bit is normally unchanged, however after a PAL-based interruption this bit is set to 0.
k. This bit is set to 0 after the successful execution of each instruction in a bundle except for `rfi` which may set it to 1.
l. This bit is ignored when restarting IA-32 instructions and set to zero when `br.ia` or `rfi` successfully complete and before the first IA-32 instruction starts execution.
m. After an interruption, `rfi`, or `bsw` the processor ensures register accesses are made to the new register bank. For interruptions, `rfi` and `bsw`, the processor ensures all register accesses and outstanding loads prior to the bank switch operate on the prior register bank.

3.3.3 Control Registers

Table 3-3 defines all registers in the control register name space along with serialization requirements to ensure side effects are observed by subsequent instructions. However, reads of a control register must be data serialized with prior writes to the same register. The serialization required column only refers to the side effects of the data value.

Writes to read-only registers (IVR, IRR0-3) result in an Illegal Operation fault, accesses to reserved registers result in a Illegal Operation fault. Accesses can only be performed by `mov` to/from instructions defined in Table 3-4 at privilege level 0; otherwise, a Privileged Operation fault is raised.

Table 3-3. Control Registers

	Register	Name	Description	Serialization Required
Global Control Registers	CR0	DCR	Default Control Register	inst/data
	CR1	ITM	Interval Timer Match register	data[a]
	CR2	IVA	Interruption Vector Address	inst[a]
	CR3-CR7		reserved	
	CR8	PTA	Page Table Address	inst/data[b]
	CR9-15		reserved	
Interruption Control Registers	CR16	IPSR	Interruption Processor Status Register	implied[d]
	CR17	ISR	Interruption Status Register	implied[c]
	CR18		reserved	
	CR19	IIP	Interruption Instruction Pointer	implied[d]
	CR20	IFA	Interruption Faulting Address	implied[d]
	CR21	ITIR	Interruption TLB Insertion Register	implied[d]
	CR22	IIPA	Interruption Instruction Previous Address	implied[c]
	CR23	IFS	Interruption Function State	implied[d,e]
	CR24	IIM	Interruption Immediate register	implied[c]
	CR25	IHA	Interruption Hash Address	implied[c]
Reserved	CR26-63		reserved	

Table 3-3. Control Registers (Continued)

	Register	Name	Description	Serialization Required
Interrupt Control Registers	CR64	LID	Local Interrupt ID	data[a]
	CR65	IVR	External Interrupt Vector Register (read only)	data[a]
	CR66	TPR	Task Priority Register	data[a]
	CR67	EOI	End Of External Interrupt	data[a]
	CR68	IRR0	External Interrupt Request Register 0 (read only)	data[a]
	CR69	IRR1	External Interrupt Request Register 1 (read only)	data[a]
	CR70	IRR2	External Interrupt Request Register 2 (read only)	data[a]
	CR71	IRR3	External Interrupt Request Register 3 (read only)	data[a]
	CR72	ITV	Interval Timer Vector	data[a]
	CR73	PMV	Performance Monitoring Vector	data[a]
	CR74	CMCV	Corrected Machine Check Vector	data[a]
	CR75-79		reserved	reserved
	CR80	LRR0	Local Redirection Register 0	data[a]
	CR81	LRR1	Local Redirection Register 1	data[a]
Reserved	CR82-127		reserved	reserved

a. Serialization is needed to ensure external interrupt masking, new interval timer match values or new interruption table addresses are observed before a given point in program execution.

b. Serialization is needed to ensure new values in PTA are visible to the hardware Virtual Hash Page Table (VHPT) walker before a dependent instruction fetch or data access.

c. These registers are modified by the processor on an interruption or by an explicit move to these registers. There are no side effects when written.

d. These registers are implied operands to the rfi and/or TLB insert instructions. The processor ensures writes in previous instruction groups are observed by rfi and/or TLB insert instructions in subsequent instruction groups. These registers are also modified by the processor on an interruption, subsequent reads return the results of the interruption. There are no other side effects.

e. IFS written by a `cover` instruction followed by a move-from IFS is implicitly serialized.

Table 3-4. Control Register Instructions

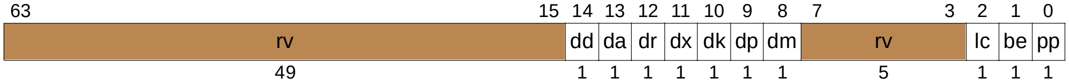
Mnemonic	Description	Operation	Format
`mov` $cr_3 = r_2$	Move to control register	CR[r_3] ← GR[r_2]	M
`mov` $r_1 = cr_3$	Move from control register	GR[r_1] ← CR[r_3]	M
`srlz.i, rfi`	Serialize instruction references	Ensure side effects are observed by the instruction fetch stream	M
`srlz.d`	Serialize data references	Ensure side effects are observed by the execute and data streams	M

3.3.4 Global Control Registers

3.3.4.1 Default Control Register (DCR – CR0)

The DCR specifies default parameters for PSR values on interruption, some additional global controls, and whether speculative load faults can be deferred. Figure 3-3 and Table 3-5 define and describe the DCR fields.

Figure 3-3. Default Control Register (DCR – CR0)



63 15	14	13	12	11	10	9	8	7 3	2	1	0
rv	dd	da	dr	dx	dk	dp	dm	rv	lc	be	pp
49	1	1	1	1	1	1	1	5	1	1	1

Table 3-5. Default Control Register Fields

Field	Bit	Description	Serialization Required
pp	0	Privileged Performance monitor default – On interruption, DCR.pp is loaded into PSR.pp.	data
be	1	Big-Endian default – When 1, Virtual Hash Page Table (VHPT) walker accesses are performed big-endian; otherwise, little-endian. On interruption, DCR.be is loaded into PSR.be.	inst
lc	2	IA-32 Lock Check enable – When 1, and an IA-32 atomic memory reference is defined as requiring a read-modify-write operation external to the processor under an external bus lock, an IA-32_Intercept(Lock) is raised. (IA-32 atomic memory references are defined to require an external bus lock for atomicity when the memory transaction is made to non-write-back memory or are unaligned across an implementation-specific non-supported alignment boundary.) When 0, and an IA-32 atomic memory reference is defined as requiring a read-modify-write operation external to the processor under external bus lock, the processor may either execute the transaction as a series of non-atomic transactions or perform the transaction with an external bus lock, depending on the processor implementation. Intel® Itanium™ semaphore accesses ignore this bit. All unaligned Intel® Itanium™ semaphore references generate an Unaligned Data Reference fault. All aligned Intel® Itanium™ semaphore references made to memory that is neither write-back cacheable nor a NaTPage result in an Unsupported Data Reference fault.	data
dm	8	Defer TLB Miss faults only (VHPT data, Data TLB, and Alternate Data TLB faults) – When 1, and a TLB miss is deferred, lower priority Debug faults may still be delivered. A TLB miss fault, deferred or not, precludes concurrent Page not Present, Key Miss, Key Permission, Access Rights, or Access Bit faults. This bit is ignored by IA-32 instructions.	data
dp	9	Defer Page not Present faults only – When 1, and a Page not Present fault is deferred, lower priority Debug faults may still be delivered. A Page not Present fault, deferred or not, precludes concurrent Key Miss, Key Permission, Access Rights, or Access Bit faults. This bit is ignored by IA-32 instructions.	data
dk	10	Defer Key Miss faults only – When 1, and a Key Miss fault is deferred, lower priority Access Bit, Access Rights or Debug faults may still be delivered. A Key Miss fault, deferred or not, precludes concurrent Key Permission faults. This bit is ignored by IA-32 instructions.	data
dx	11	Defer Key Permission faults only – When 1, and a Key Permission fault is deferred, lower priority Access Bit, Access Rights or Debug faults may still be delivered. This bit is ignored by IA-32 instructions.	data
dr	12	Defer Access Rights faults only – When 1, and an Access Rights fault is deferred, lower priority Access Bit or Debug faults may still be delivered. This bit is ignored by IA-32 instructions.	data
da	13	Defer Access Bit faults only – When 1, and an Access Bit fault is deferred, lower priority Debug faults may still be delivered. This bit is ignored by IA-32 instructions.	data
dd	14	Defer Debug faults – When 1, Data Debug faults on speculative loads are deferred. This bit is ignored by IA-32 instructions.	data
rv	7:3, 63:15	reserved	reserved

For the DCR exception deferral bits, when the bit is 1, and a speculative load results in the specified fault condition, and the speculative load's code page exception deferral bit (ITLB.ed) is 1, the exception is deferred by setting the speculative load target register to NaT or NaTVal. Otherwise, the specified fault is taken on the speculative load. For a description of faults on speculative loads see "Deferral of Speculative Load Faults" on page 1:88.

Since DCR.be also controls byte ordering of VHPT references that are the result of instruction misses, DCR.be requires instruction serialization. Other DCR bits require data serialization only.

3.3.4.2 Interval Time Counter and Match Register (ITC – AR44 and ITM – CR1)

The Interval Time Counter (ITC) and Interval Timer Match (ITM) register support fine-grained time stamps and elapsed time notification, see Figure 3-4 and Figure 3-5.

Figure 3-4. Interval Time Counter (ITC – AR44)

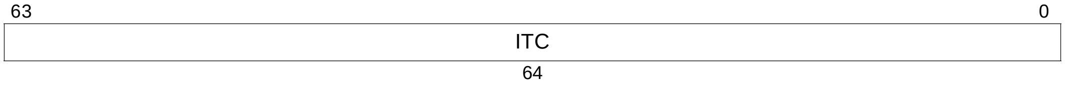
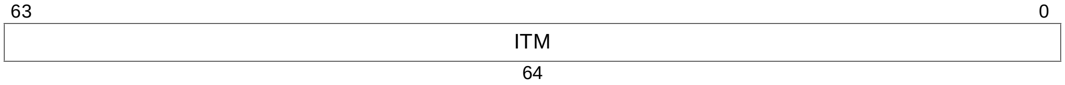


Figure 3-5. Interval Timer Match Register (ITM – CR1)



The ITC is a free-running 64-bit counter that counts up at a fixed relationship to the processor clock; 64-bit overflow conditions occur without notification. The ITC counting rate is not affected by power management mechanisms. The ITC can be read at any privilege level if PSR.si is zero. The timer can be secured from non-privileged access by setting PSR.si to 1. When secured, a read of the ITC by non-privileged code results in a Privileged Register fault. Writes to the ITC can only be performed at privilege level 0; otherwise, a Privileged Register fault is raised.

The IA-32 Time Stamp Counter (TSC) is equivalent to ITC. The ITC can be read by the IA-32 `rdtsc` (read time stamp counter) instruction. System software can secure the ITC from non-privileged IA-32 access by setting either PSR.si or CFLG.tsd to 1. When secured, an IA-32 read of the ITC at any privilege level other than the most privileged raises an IA-32_Exception(GPfault).

When the value in the ITC is equal to the value in the ITM an Interval Timer Interrupt is raised. Once the interruption is taken by the processor and serviced by software, the ITC may not necessarily be equal to the ITM. The ITM is accessible only at privilege level 0; otherwise, a Privileged Operation fault is raised.

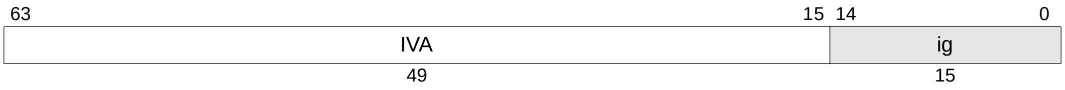
The interval counter can be written, for initialization purposes, by privileged code. The ITC is not architecturally guaranteed to be synchronized with any other processor's interval time counter in an multiprocessor system, nor is it synchronized with the wall clock. Software must calibrate interval timer ticks to wall clock time and periodically adjust for drift.

Modification of the ITC or ITM is not necessarily serialized with respect to instruction execution. Software can issue a data serialization operation to ensure the ITC or ITM updates and possible side effects are observed by a given point in program execution. Software must accept a level of sampling error when reading the interval timer due to various machine stall conditions, interruptions, bus contention effects, etc. Please see the processor specific documentation for further information on the level of sampling error of the Itanium processor.

3.3.4.3 Interruption Vector Address (IVA – CR2)

The IVA specifies the location of the interruption vector table in the virtual address space, or the physical address space if PSR.it is 0, see Figure 3-6. The size of the vector table is 32K bytes and is 32K byte aligned. The lower 15 bits of the IVA are ignored when written, reads return zeros. All upper 49 address bits of IVA must be implemented regardless of the size of the physical and virtual address space. If an unimplemented virtual or physical address (see "Unimplemented Address Bits" on page 1:61) is loaded into IVA, and an interruption occurs, processor behavior is unpredictable. See "IVA-based Interruption Vectors" on page 1:96 for a description of an interruption table layout.

Figure 3-6. Interruption Vector Address (IVA – CR2)



3.3.4.4 Page Table Address (PTA – CR8)

The PTA anchors the Virtual Hash Page Table (VHPT) in the virtual address space. See "Virtual Hash Page Table (VHPT)" on page 1:51 for a complete definition of the VHPT. Operating systems must ensure that the table is aligned on a natural boundary; otherwise, processor operation is undefined. See Figure 3-7 and Table 3-6 for the PTA field definitions.

Figure 3-7. Page Table Address (PTA – CR8)

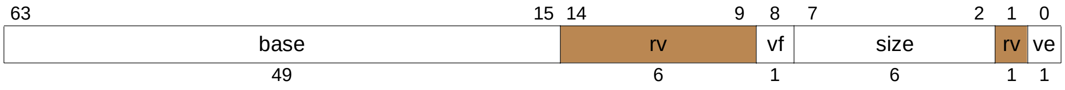


Table 3-6. Page Table Address Fields

Field	Bits	Description
ve	0	VHPT Enable – When 1, the processor is enabled to walk the VHPT.
size	7:2	VHPT Size – VHPT table size in power of 2 increments, table size is 2^{size} bytes. Size generates a mask that is logically AND'ed with the result of the VHPT hash function. Minimum VHPT table size is 32K bytes; otherwise, a Reserved Register/Field fault is raised (see "Virtual Hash Page Table (VHPT)" on page 1:51). The maximum size is 2^{61} bytes for long format VHPTs, and 2^{52} bytes for short format VHPTs.
vf	8	VHPT Format – When 0, 8-byte short format entries are used, when 1, 32-byte long format entries are used.
base	63:15	VHPT Base virtual address – Defines the starting virtual address of the VHPT table. Base is logically OR'ed with the hash index produced by the VHPT hash function when referencing the VHPT. Base must be on 2^{size} boundary otherwise processor operation is undefined. All base address bits of PTA must be implemented regardless of the size of the physical and virtual address space. If an unimplemented virtual address (see "Unimplemented Address Bits" on page 1:61) is used by the processor as a page table base, all VHPT walks generate an Instruction/Data TLB miss (see "Translation Searching" on page 1:57).
rv	1, 14:9	reserved

3.3.5 Interruption Control Registers

Registers CR16 - CR25 record information at the time of an interruption (including from the IA-32 instruction set) and are used by handlers to process the interruption.

The interruption control registers can only be read or written while PSR.ic is 0; otherwise, an Illegal Operation fault is raised. These registers are only guaranteed to retain their values when PSR.ic is 0. When PSR.ic is 1, the processor does not preserve their contents. IIPA has special behavior in case of an `rfi` to a fault. Refer to "Interruption Instruction Previous Address (IIPA – CR22)" on page 1:32.

3.3.5.1 Interruption Processor Status Register (IPSR – CR16)

On an interruption and if PSR.ic is 1, the IPSR receives the value of the PSR. The IPSR, IIP and IFS are used to restore processor state on a Return From Interruption (`rfi`). The IPSR has the same format as PSR, see "Processor Status Register (PSR)" on page 1:18 for details.

3.3.5.2 Interruption Status Register (ISR – CR17)

The ISR receives information related to the nature of the interruption, and is written by the processor on all interruption events regardless of the state of PSR.ic, except for Data Nested TLB faults. The ISR contains information about the excepting instruction and its properties such as whether it was doing a read, write, execute, speculative, or non-access operation, see Figure 3-8 and Table 3-7. Multiple bits may be concurrently set in the ISR, for example, a faulting semaphore operation will set both ISR.r and ISR.w, and faults on speculative loads will set ISR.sp and ISR.r. Additional fault or trap specific information is available in ISR.code and ISR.vector. Refer to Section 8.2 "ISR Settings" for complete definition of the ISR field settings.

Figure 3-8. Interruption Status Register (ISR – CR17)

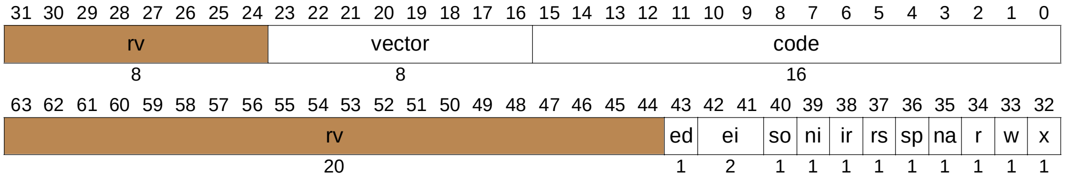


Table 3-7. Interruption Status Register Fields

Field	Bits	Description
code	15:0	Interruption Code – 16 bit code providing additional information specific to the current interruption. For IA-32 specific exceptions and software interrupts, contains the IA-32 interruption error code or zero.
vector	23:16	IA-32 exception/interception vector number. For IA-32 exceptions and software interrupts, contains the IA-32 vector number (e.g., GPFault has a vector number of 13). See Chapter 9, "IA-32 Interruption Vector Descriptions" for details.
x	32	Execute exception – Interruption is associated with an instruction fetch (including IA-32).
w	33	Write exception – Interruption is associated with a write operation. Both ISR.r and ISR.w are set for IA-32 read-modify-write instructions.
r	34	Read exception – Interruption is associated with a read operation. Both ISR.r and ISR.w are set for IA-32 read-modify-write instructions.

Table 3-7. Interruption Status Register Fields (Continued)

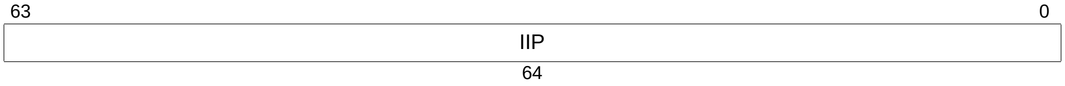
Field	Bits	Description
na	35	Non-access exception – See Section 5.5.2. This bit is always 0 for interruptions taken in the IA-32 instruction set.
sp	36	Speculative load exception – Interruption is associated with a speculative load instruction. This bit is always 0 for interruptions taken in the IA-32 instruction set.
rs	37	Register Stack – Interruption is associated with a mandatory RSE fill or spill. This bit is always 0 for interruptions taken in the IA-32 instruction set.
ir	38	Incomplete Register frame – The current register frame is incomplete when the interruption occurred. This bit is always 0 for interruptions taken in the IA-32 instruction set.
ni	39	Nested Interruption – Indicates that PSR.ic was 0 or in-flight when the interruption occurred. This bit is always 0 for interruptions taken in the IA-32 instruction set.
so	40	IA-32 Supervisor Override – Indicates the fault occurred during an IA-32 instruction set supervisor override condition (the processor was performing a data memory accesses to the IDT, GDT, LDT or TSS segments) or an IA-32 data memory access at a privilege level of zero. This bit is always 0 for interruptions taken while executing Intel® Itanium™ instructions.
ei	42:41	Excepting Instruction – 0 – exception due to instruction in slot 0 1 – exception due to instruction in slot 1 2 – exception due to instruction in slot 2 For faults and external interrupts, ISR.ei is equal to IPSR.ri. For traps, ISR.ei defines the slot of the excepting instruction. Traps on the L+X instruction of an MLX set ISR.ei to 2. This field is always 0 for interruptions taken in the IA-32 instruction set.
ed	43	Exception Deferral – this bit is set to the value of the TLB exception deferral bit (TLB.ed) for the instruction page containing the faulting instruction. If a translation does not exist or instruction translation is disabled, or if the interruption is caused by a mandatory RSE spill or fill, ISR.ed is set to 0. This bit is always 0 for interruptions taken in the IA-32 instruction set.
rv	31:24, 63:44	reserved

3.3.5.3 Interruption Instruction Bundle Pointer (IIP – CR19)

On an interruption and if PSR.ic is 1, the IIP receives the value of IP. IIP contains the virtual address (or physical if instruction translations are disabled) of the next instruction bundle or the IA-32 instruction to be executed upon return from the interruption. For IA-32 instruction addresses, IIP is zero extended to 64-bits and specifies a byte granular address. For traps and interrupts, IIP points to the next instruction to execute. For faults, IIP points to the faulting instruction. As shown in Figure 3-9, all 64-bits of the IIP must be implemented regardless of the size of the physical and virtual address space supported by the processor model (see "Unimplemented Address Bits" on page 1:61). IIP also receives byte-aligned IA-32 instruction pointers. The IIP, IPSR and IFS are used to restore processor state on a Return From Interruption instruction (`rfi`). See "Interruption Vector Descriptions" on page 1:147 for usages of the IIP.

An `rfi` to Itanium-based code (IPSR.is is 0) ignores IIP{3:0}, an `rfi` to IA-32 code (IPSR.is is 1) ignores IIP{63:32}. Ignored bits are assumed to be zero.

Figure 3-9. Interruption Instruction Bundle Pointer (IIP – CR19)



Control transfers to unimplemented addresses (see “Unimplemented Address Bits” on page 1:61) result in an Unimplemented Instruction Address trap. When this trap is delivered, IIP is written as follows:

- If the trap is taken for an unimplemented virtual address, IIP is written with the implemented virtual address bits IP{63:61} and IP{IMPL_VA_MSB:0} only. Bits IIP{60:IMPL_VA_MSB+1} are set to IP{IMPL_VA_MSB}, i.e., sign-extended.
- If the trap is taken for an unimplemented physical address, IIP is written with the physical addressing memory attribute bit IP{63} and the implemented physical address bits IP{IMPL_PA_MSB:0} only. Bits IIP{62:IMPL_PA_MSB+1} are set to 0.

When an `rfi` is executed with an unimplemented address in IIP (an unimplemented virtual address if IPSR.it is 1, or an unimplemented physical address if IPSR.it is 0), and an Unimplemented Instruction Address trap is taken, an implementation may optionally leave IIP unchanged (preserving the unimplemented address in IIP).

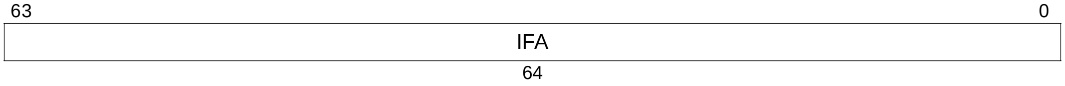
Note: Since IP{3:0} are always 0 when executing Itanium-based code, IIP{3:0} will always be 0 when any interruption is taken from Itanium-based code, with the exception of an Unimplemented Instruction Address trap on an `rfi`, where IIP may optionally be preserved as whatever value it held before executing the `rfi`.

3.3.5.4 Interruption Faulting Address (IFA – CR20)

On an interruption and if PSR.ic is 1, the IFA receives the virtual address (or physical address if translations are disabled) that raised a fault. IFA reports the faulting address for both instruction and data memory accesses (including IA-32). For faulting data references (including IA-32), IFA points to the first byte of the faulting data memory operand. IFA reports a byte granular address. For faulting instruction references (including IA-32), IFA contains the 16-byte aligned bundle address (IFA{3:0} are zero) of the faulting instruction. For faulting IA-32 instructions, IIP points to the first byte of the IA-32 instruction, and is byte granular. In the event of an IA-32 instruction spanning a virtual page boundary, IA-32 instruction fetch faults are reported as either (1) for faults on the first page, IFA is set to the bundle address (IFA{3:0}=0) of the faulting instruction and IIP points to the first byte of the faulting instruction, or (2) for faults on the second page, IFA contains the bundle address of the second virtual page and IIP points to the first byte of the faulting IA-32 instruction.

The IFA also specifies a translation’s virtual address when a translation entry is inserted into the instruction or data TLB. See “Interruption Vector Descriptions” on page 1:147 and “Translation Insertion Format” on page 1:44 for usages of the IFA. As shown in Figure 3-10, all 64-bits of the IFA must be implemented regardless of the size of the virtual and physical space supported by the processor model (see “Unimplemented Address Bits” on page 1:61).

Figure 3-10. Interruption Faulting Address (IFA – CR20)



3.3.5.5 Interruption TLB Insertion Register (ITIR – CR21)

The ITIR receives default translation information from the referenced virtual region register on a virtual address translation fault. See “Interruption Vector Descriptions” on page 1:147 for the fault conditions that set the ITIR. The ITIR provides additional virtual address translation parameters on an insertion into the instruction or data TLB. See “Translation Instructions” on page 1:50 for ITIR usage information. Figure 3-11 and Table 3-8 define the ITIR fields.

Figure 3-11. Interruption TLB Insertion Register (ITIR)

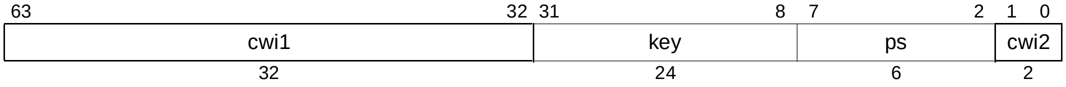


Table 3-8. ITIR Fields

Field	Bits	Description
cwi1, cwi2	63:32, 1:0	On a read these fields may return zeros or the value last written to them. If a non-zero value is written, a subsequent TLB insert will raise a Reserved Register Field fault depending on other parameters to the insert. See “Translation Insertion Format” on page 2:44.
ps	7:2	Page Size – On a TLB insert, specifies the size of the virtual to physical address mapping. On an instruction or data translation fault, this field is set to the accessed region’s page size (RR.ps).
key	31:8	protection Key – On a TLB insert specifies a protection key that uniquely tags translations to a protection domain. On an instruction or data translation fault, Key is set to the accessed Region Identifier (RR.rid).

3.3.5.6 Interruption Instruction Previous Address (IIPA – CR22)

For Itanium instructions, IIPA records the last successfully executed instruction bundle address. For IA-32 instructions, IIPA records the byte granular virtual instruction address zero extended to 64-bits of the faulting or trapping IA-32 instruction. In the case of a fault, IIPA does not report the address of the last successfully executed IA-32 instruction, but rather the address of the faulting IA-32 instruction. IIPA preserves bits 3:0 for byte aligned IA-32 instruction addresses.

The IIPA can be used by software to locate the address of the instruction bundle or IA-32 instruction that raised a trap or the instruction executed prior to a fault or interruption. In the case of a branch related trap, IIPA points to the instruction bundle which contained the branch instruction that raised the trap, while IIP points to the target of the branch.

When an instruction successfully executes without a fault, and the PSR.ic bit was 1 prior to instruction execution, it becomes the “last successfully executed instruction”. On interruptions, IIPA contains the address of the last successfully executed instruction bundle or IA-32 instruction, if PSR.ic was 1 prior to the interruption. If no such instruction exists, e.g., in case of an `rfi` to a fault, the contents of IIPA remain unchanged.

When PSR.ic is one, accesses to IIPA cause an Illegal Operation fault. When PSR.ic is zero, IIPA is not updated by hardware and can be read and written by software. This permits low-level code to preserve IIPA across interruptions.

If the PSR.ic bit is explicitly cleared, e.g., by using `rsm`, then the contents of IIPA are undefined. Only when the PSR.ic bit is cleared by an interruption is the value of IIPA defined. It may point at the instruction which caused a trap, or at the instruction just prior to a faulting instruction, at an earlier instruction that became defined by some prior interruption, or by a move to IIPA instruction when PSR.ic was zero.

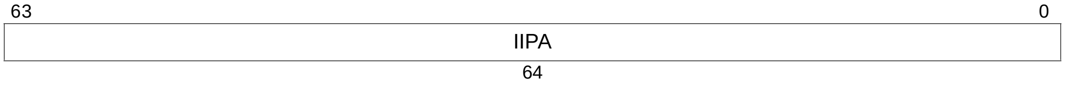
If the PSR.ic bit is explicitly set, e.g., by using `ssm`, then the contents of IIPA are undefined until the PSR.ic bit has been serialized and an interruption occurs.

During instruction set transitions the following boundary cases exist:

- On faults taken on the first IA-32 instruction after a `br.ia` or `rfi`, IIPA records the faulting IA-32 instruction address.
- On `br.ia` traps, IIPA records the address of the trapping instruction bundle.
- On faults taken on the first Itanium instruction after leaving the IA-32 instruction set, due to a `jmpe` or interruption, IIPA contains the address of the `jmpe` instruction or the interrupted IA-32 instruction.
- On `jmpe` Data Debug, Single Step and Taken Branch traps, IIPA contains the address of the `jmpe` instruction.

As shown in Figure 3-12, all 64-bits of the IIPA must be implemented regardless of the size of the physical and virtual address space supported by the processor model (see “Unimplemented Address Bits” on page 1:61).

Figure 3-12. Interruption Instruction Previous Address (IIPA – CR22)



3.3.5.7 Interruption Function State (IFS – CR23)

The IFS register is used to reload the current register stack frame (CFM) on a Return From Interruption (`rfi`). If the IFS is accessed while PSR.ic is 1, an Illegal Operation fault is raised. The IFS can only be accessed at privilege level 0; otherwise, a Privileged Operation fault is raised. The IFS.v bit is cleared on interruption if PSR.ic is 1. All other fields are undefined after an interruption. If PSR.ic is 0, the `cover` instruction copies CFM to IFS.ifm and sets IFS.v to 1. See Figure 3-13 and Table 3-9 for the IFS field definitions.

Figure 3-13. Interruption Function State (IFS – CR23)

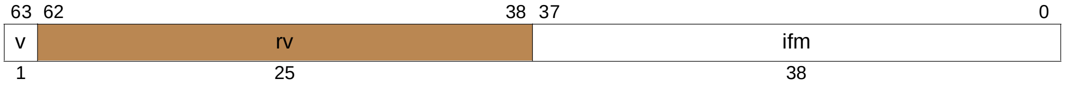


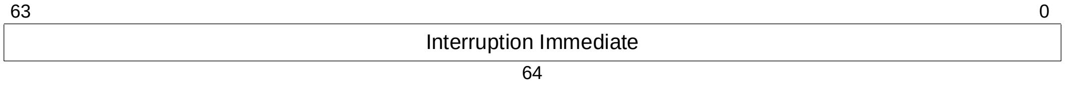
Table 3-9. Interruption Function State Fields

Field	Bits	Description
ifm	37:0	Interruption Frame Marker
v	63	Valid bit, cleared to 0 on interruption if PSR.ic is 1.
rv	62:38	reserved

3.3.5.8 Interruption Immediate (IIM – CR24)

If PSR.ic is 1, the IIM (Figure 3-14) records the zero-extended immediate field encoded in `chk.a`, `chk.s`, `fchkf` or `break` instruction faults. The `break.b` instruction always writes a zero value and ignores its immediate field. The IA-32_Intercept vector writes all 64-bits of IIM to indicate the cause of the intercept. See Table 8-1 on page 2:148 for the value of IIM in other situations. For the purpose of resource dependency, IIM is written as a result of the fault, not by the instruction itself.

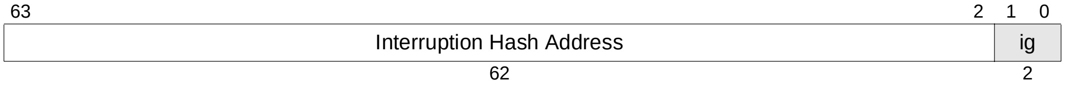
Figure 3-14. Interruption Immediate (IIM – CR24)



3.3.5.9 Interruption Hash Address (IHA – CR25)

The IHA (Figure 3-15) is loaded with the address of the Virtual Hash Page Table (VHPT) entry the processor referenced or would have referenced to resolve a translation fault. The IHA is written on interruptions by the processor when PSR.ic is 1. Refer to "VHPT Hashing" on page 1:54 for complete details. See Table 8-1 on page 2:148 for the value of IHA in other situations. All upper 62 address bits of IHA must be implemented regardless of the size of the virtual address space supported by the processor model (see "Unimplemented Address Bits" on page 1:61). The virtual address written to IHA by the processor is guaranteed to be an implemented virtual addresses on all processor models; however, if the address referenced by the VHPT is an unimplemented virtual address, the value of IHA is undefined.

Figure 3-15. Interruption Hash Address (IHA – CR25)



3.3.6 External Interrupt Control Registers

The external interrupt control registers (CR64-81) are defined in "External Interrupt Control Registers" on page 1:104. They are used to prioritize and deliver external interrupts, send inter-processor interrupts to other processors and assign interrupt vectors for locally generated processor interrupts.

3.3.7 Banked General Registers

Banked general registers (see Figure 3-16) provide immediate register context for low-level interruption handlers (e.g., speculation and TLB miss handlers). Upon interruption, the processor switches 16 general purpose registers (GR16 to GR31) to register bank 0, register bank 1 contents are preserved.

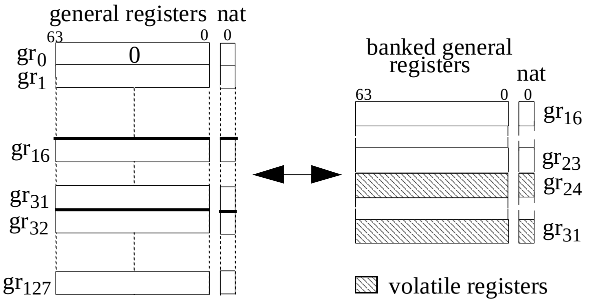
When PSR.bn is 1, bank 1 for registers GR16 to GR31 is selected; when 0, bank 0 for registers GR16 to GR31 is selected. Banks are switched in the following cases:

- an interruption selects bank 0,
- `rfi` switches to the bank specified by IPSR.bn, or

- `bsw` switches to the specified bank.

On an interruption or bank switch, the processor ensures all prior register accesses (reads and writes) are performed to the prior register bank. Data values in banked registers are preserved across bank switches and both banks maintain NaT values when loaded from general registers. Registers from both banks cannot be addressed at the same time. However, non-banked general registers (GR0-15, and GR32-127) are accessible regardless of the state of PSR.bn.

Figure 3-16. Banked General Registers



The ALAT register target tracking mechanism (see "Data Speculation" on page 1:57) does not distinguish the two register banks; from the ALAT's perspective GR16 in bank 0 is the same register as GR16 in bank 1.

Operating systems should ensure that IA-32 and Itanium-based application code is executed within register bank 1. If IA-32 or Itanium-based application code executes out of register bank 0, the application register state (including IA-32) will be lost on any interruption. During interruption processing the operating system uses register bank 0 as the initial working register context.

Usage of these additional registers is determined by software conventions. However, registers GR24 to GR31, of bank 0, are not preserved when PSR.ic is 1; operating system code can not rely on register values being preserved unless PSR.ic is 0. While PSR.ic is 1, processor-specific firmware may use these registers for machine check or firmware interruption handling at any point regardless of the state of PSR.i. If PSR.ic is 0, GR24 to GR31 can be used as scratch registers for low-level interruption handlers. Registers GR16 to GR23 are always preserved; operating system code can rely on the values being preserved.

Addressing and Protection 4

This chapter defines operating system resources to translate 64-bit virtual addresses into physical addresses, 32-bit virtual addressing, virtual aliasing, physical addressing, memory ordering and properties of physical memory. Register state defined to support virtual memory management is defined in Chapter 3, while Chapter 5 provides complete information on virtual memory faults.

Note: Unless otherwise noted, references to "interruption" in this chapter refer to IVA-based interruptions. See "Interruption Definitions" on page 2:79.

The following key features are supported by the virtual memory model.

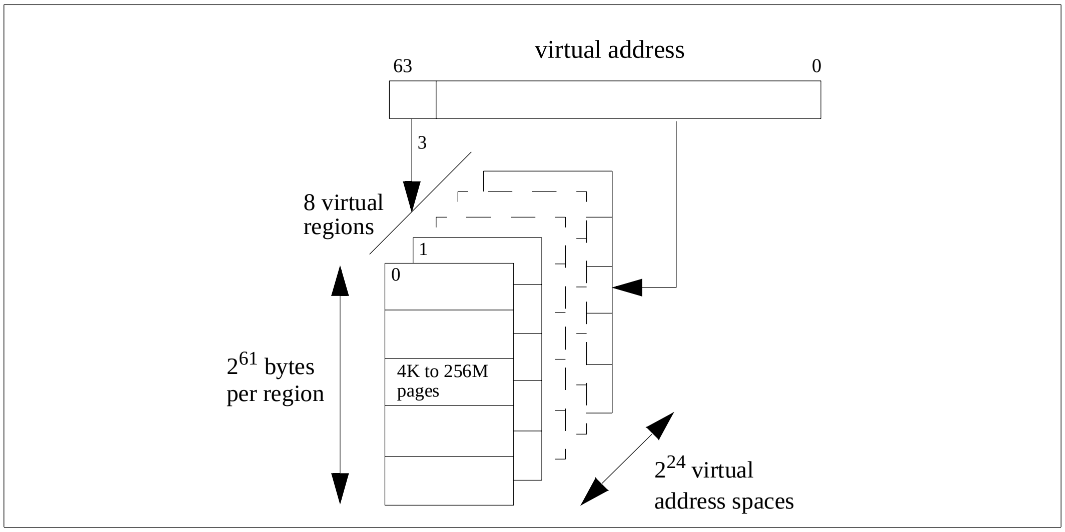
- Virtual Regions are defined to support contemporary operating system Multiple Address Space (MAS) models of placing each process within a unique address space. Region identifiers uniquely tag virtual address mappings to a given process.
- Protection Domain mechanisms support the Single Address Space (SAS) model, where processes co-exist within the same virtual address space.
- Translation Lookaside Buffer (TLB) structures are defined to support high-performance paged virtual memory systems. Software TLB fill and protection handlers are utilized to defer translation policies and protection algorithms to the operating system.
- A Virtual Hash Page Table (VHPT) is designed to augment the performance of the TLB. The VHPT is an extension of the processor's TLB that resides in memory and can be automatically searched by the processor. A particular operating system page table format is not dictated. However, the VHPT is designed to mesh with two common translation structures: the virtual linear page table and hashed page table. Enabling of the VHPT and the size of the VHPT are completely under software control.
- Sparse 64-bit virtual addressing is supported by providing for large translation arrays (including multiple levels of hierarchy similar to a cache hierarchy), efficient translation miss handling support, multiple page sizes, pinned translations, and mechanisms to promote sharing of TLB and page table resources.

4.1 Virtual Addressing

As seen by Itanium-based application programs, the virtual addressing model is fundamentally a 64-bit flat linear virtual address space. 64-bit general registers are used as pointers into this address space. IA-32 32-bit virtual linear addresses are zero extended into the 64-bit virtual address space.

As shown in Figure 4-1, the 64-bit virtual address space is divided into eight 2^{61} byte virtual regions. The region is selected by the upper 3-bits of the virtual address. Associated with each virtual region is a region register that specifies a 24-bit region identifier (unique address space number) for the region. Eight out of the possible 2^{24} virtual address spaces are concurrently accessible via the 8 region registers. The region identifier can be considered the high order address bits of a large 85-bit global address space for a single address space model, or as a unique ID for a multiple address space model.

Figure 4-1. Virtual Address Spaces



By assigning sequential region identifiers, regions can be coalesced to produce larger 62-, 63- or 64-bit spaces. For example, an operating system could implement a 62-bit region for process private data, 62-bit region for I/O, and a 63-bit region for globally shared data. Default page sizes and translation policies can be assigned to each virtual region.

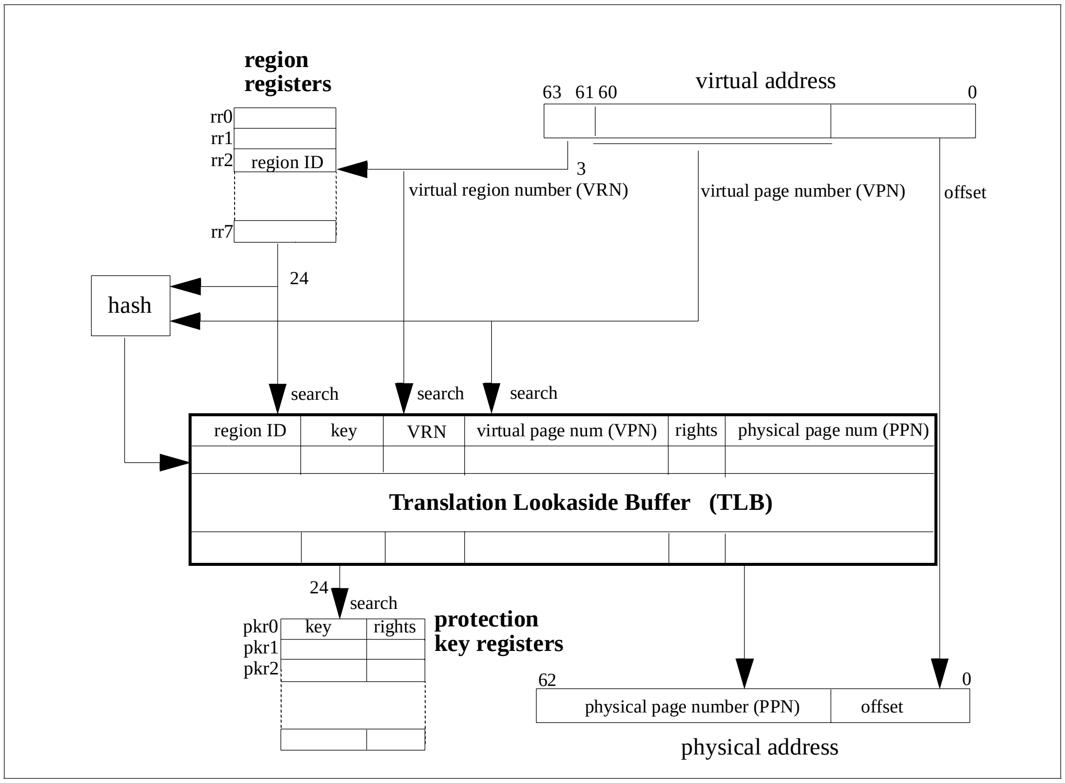
Figure 4-2 shows the process of mapping a virtual address into a physical address. Each virtual address is composed of three fields: the Virtual Region Number, the Virtual Page Number, and the page offset. The upper 3-bits select the Virtual Region Number (VRN). The least-significant bits form the page offset. The Virtual Page Number (VPN) consists of the remaining bits. The VRN bits are not included in the VPN. The page offset bits are passed through the translation process unmodified. Exact bit positions for the page offset and VPN bits vary depending on the page size used in the virtual mapping.

On a memory reference (any reference other than an insert or purge), the VRN bits select a Region Identifier (RID) from 1 of the 8 region registers, the TLB is then searched for a translation entry with a matching VPN and RID value. The VRN may optionally be used when searching for a matching translation on memory references (references other than inserts and purges - see Section 4.1.1.4). If a matching translation entry is found, the entry's physical page number (PPN) is concatenated with the page offset bits to form the physical address. Matching translations are qualified by page-granular privilege level access right checks and optional protection domain checks by verifying the translation's key is contained within a set of protection key registers and read, write, execute permissions are granted.

If the required translation is not resident in the TLB, the processor may optionally search the VHPT structure in memory for the required translation and install the entry into the TLB. If the required entry cannot be found in the TLB and/or VHPT, the processor raises a TLB Miss fault to request that the operating system supply the translation. After the operating system installs the translation in the TLB and/or VHPT, the faulting instruction can be restarted and execution resumed.

Virtual addressing for instruction references are enabled when PSR.it is 1, data references when PSR.dt is 1, and register stack accesses when PSR.rt is 1.

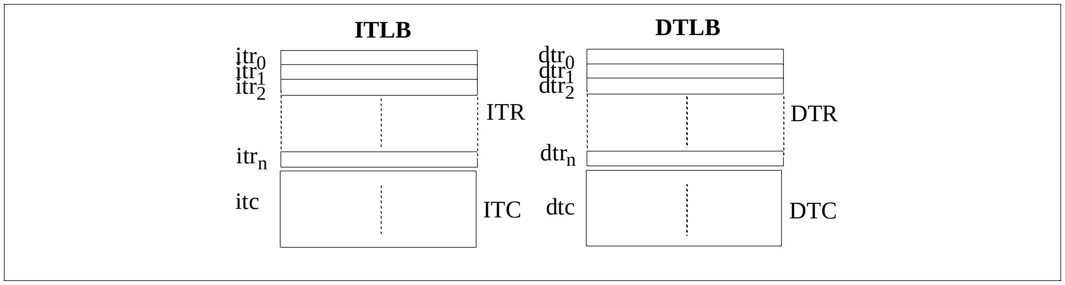
Figure 4-2. Conceptual Virtual Address Translation for References



4.1.1 Translation Lookaside Buffer (TLB)

The processor maintains two architectural TLBs as shown in Figure 4-3, the Instruction TLB (ITLB) and Data TLB (DTLB). Each TLB services translation requests for instruction and data memory references (including IA-32), respectively. The Data TLB also services translation requests for references by the RSE and the VHPT walker. The TLBs are further divided into two sub-sections; Translation Registers (TR) and Translation Cache (TC).

Figure 4-3. TLB Organization



In the remainder of this document, the term TLB refers to the combined instruction, data, translation register, and translation cache structures.

The TLB is a local processor resource; installation of a translation or local processor purges do not affect other processor's TLBs. Global TLB purges are provided to purge translations from all processors within a TLB coherence domain in a multiprocessor system.

4.1.1.1 Translation Registers (TR)

The Translation Register (TR) section of the TLB is a fully-associative array defined to hold translations that software directly manages. Software can explicitly insert a translation into a TR by specifying a register slot number. Translations are removed from the TRs by specifying a virtual address, page size and a region identifier. Translation registers allow the operating system to "pin" critical virtual memory translations in the TLB. Examples include I/O spaces, kernel memory areas, frame buffers, page tables, sensitive interruption code, etc. Instruction fetches for interruption handlers are performed using virtual addresses; therefore, virtual address ranges containing software translation miss routines and critical interruption sequences should be pinned or else additional TLB faults may occur. Other virtual mappings may be pinned for performance reasons.

Entries are placed into a specific TR slot with the Insert Translation Register (`itr`) instruction. Once a translation is inserted, the processor will not replace the translation to make room for other translations. Local translations can only be removed by software issuing the Purge Translation Register (`ptr`) instruction.

TR inserts and purges may cause other TR and/or TC entries to be removed (refer to Section 4.1.1.4 for details). Prior to inserting a TR entry, software must ensure that no overlapping translation exists in any TR (including the one being written); otherwise, a Machine Check abort may be raised, or the processor may exhibit other undefined behavior. Translation register entries may be removed by the processor due to hardware or software errors. In the presence of an error, the processor can remove TR entries; notification is raised via a Machine Check abort.

There are at least 8 instruction and 8 data TR slots implemented on all processor models. Please see the processor specific documentation for further information on the number of translation registers implemented on the Itanium processor. Translation registers support all implemented page sizes and must be implemented in a single-level fully-associative array. Any register slot can be used to specify any virtual address mapping. Translation registers are not directly readable.

In some processor models, translation registers are physically implemented as a subsection of the translation cache array. Valid TR slots are ignored for purposes of processor replacement on an insertion into the TC. However, invalid TR slots (unused slots) may be used as TC entries by the processor. As a result, software inserts into previously invalid TR entries may invalidate a TC entry in that slot.

Implementations may also place a floating boundary between TR and TC entries within the same structure where any entry above the boundary is considered a TC and any entry below the boundary a TR. To maximize TC resources, software should allocate contiguous translation registers starting at slot 0 and continuing upwards.

4.1.1.2 Translation Cache (TC)

The Translation Cache (TC) is an implementation-specific structure defined to hold the large working set of dynamic translations for memory references (including IA-32). Please see the processor specific documentation for further information on Itanium processor TC implementation details. The processor directly controls the replacement policy of all TC entries.

Entries are installed by software into the translation cache with the Insert Data Translation Cache (`itc.d`) and Insert Instruction Translation Cache (`itc.i`) instructions. The Purge Translation Cache Local (`ptc.l`) instruction purges all ITC/DTC entries in the local processor that match the specified virtual address range and region identifier. Purges of all ITC/DTC entries matching a specified virtual address range and region identifier among all processors in a TLB coherence domain can be globally performed with the Purge Translation Cache Global (`ptc.g`, `ptc.ga`) instruction. The TLB coherence domain covers at least the processors on the same local bus on which the purge was broadcast. Propagation between multiple TLB coherence domains is platform dependent. Software must handle the case where a purge does not propagate to all processors in a multiprocessor system. Translation cache purges do not invalidate TR entries.

All the entries in a local processor's ITC and DTC can be purged of all entries with a sequence of Purge Translation Cache Entry (`ptc.e`) instructions. A `ptc.e` does not propagate to other processors.

In all processor models, the translation cache has at least 1 instruction and 1 data entry in addition to the specified 8 instruction and 8 data translation registers. Implementations are free to implement translation cache arrays of larger sizes. Implementations may also choose to implement additional hierarchies for increased performance. At least one translation cache level is required to support all implemented page sizes. Additional hierarchy levels may or may not be performance optimized for the preferred page size specified by the virtual region, may be set-associative or fully associative, and may support a limited set of page sizes. Please see the processor specific documentation for further information on the Itanium processor implementation details of the translation cache.

The translation cache is managed by both software and hardware. In general, software cannot assume any entry installed will remain, nor assume the lifetime of any entry since replacement algorithms are implementation specific. The processor may discard or replace a translation at any point in time for any reason (subject to the forward progress rules below). TC purges may remove more entries than explicitly requested. In the presence of a processor hardware error, the processor may remove TC entries and optionally raise a Corrected Machine Check Interrupt.

In order to ensure forward progress for Itanium-based code, the following rules must be observed by the processor and software.

- Software may insert multiple translation cache entries per TLB fault, provided that only the last installed translation is required for forward progress.
- The processor may occasionally invalidate the last TC entry inserted. The processor must guarantee visibility of the last inserted TC entry to all references while PSR.ic is zero. The processor must eventually guarantee visibility of the last inserted TC entry until an `rfi` sets PSR.ic to 1 and at least one instruction is executed with PSR.ic equal to 1, and completes without a fault or interrupt. The last inserted TC entry may be occasionally removed before this point, and software must be prepared to re-insert the TC entry on a subsequent fault. For example, eager or mandatory RSE activity, speculative VHPT walks, or other interruptions of the restart instruction may displace the software-inserted TC entry, but when software later re-inserts the same TC entry, the processor must eventually complete the restart instruction to ensure forward progress, even if that restart instruction takes other faults which must be handled before it can complete. If PSR.ic is set to 1 by instructions other than `rfi`, the processor does not guarantee forward progress.
- If software inserts an entry into the TLB with an overlapping entry (same or larger size) in the VHPT, and if the VHPT walker is enabled, forward progress is not guaranteed. See "VHPT Searching" on page 2:52.

- Software may only make references to memory with physical addresses or with virtual addresses which are mapped with TRs, or to addresses mapped by the just-inserted translation, between the insertion of a TC entry, and the execution of the instruction with PSR.ic equal to 1 which is dependent on that entry for forward progress. Software may also make repeated attempts to execute the same instruction with PSR.ic equal to 1. If software makes any other memory references than these, the processor does not guarantee forward progress.
- Software must not defeat forward progress by consistently displacing a required TC entry through a global or local translation cache purge.

IA-32 code has more stringent forward progress rules that must be observed by the processor and software. IA-32 forward progress rules are defined in Section 10.6.3.

The translation cache can be used to cache TR entries if the TC maintains the instruction vs. data distinction that is required of the TRs. A data reference cannot be satisfied by a TC entry that is a cache of an instruction TR entry, nor can an instruction reference be satisfied by a TC entry that is a cache of a data TR entry. This approach can be useful in a multi-level TLB implementation.

4.1.1.3 Unified Translation Lookaside Buffers

Some processor models may merge the ITC and DTC into a unified translation cache. The minimum number of unified entries is 2 (1 for instruction, and 1 for data). Processors may service instruction fetch memory references with TC entries originally installed into the DTC and service data memory references with translations originally installed in the ITC. To ensure consistent operation across processor implementations, software is recommended to not install different translations into the ITC or DTC for the same virtual region and virtual address. ITC inserts may remove DTC entries. DTC inserts may remove ITC entries. TC purges remove ITC and DTC entries.

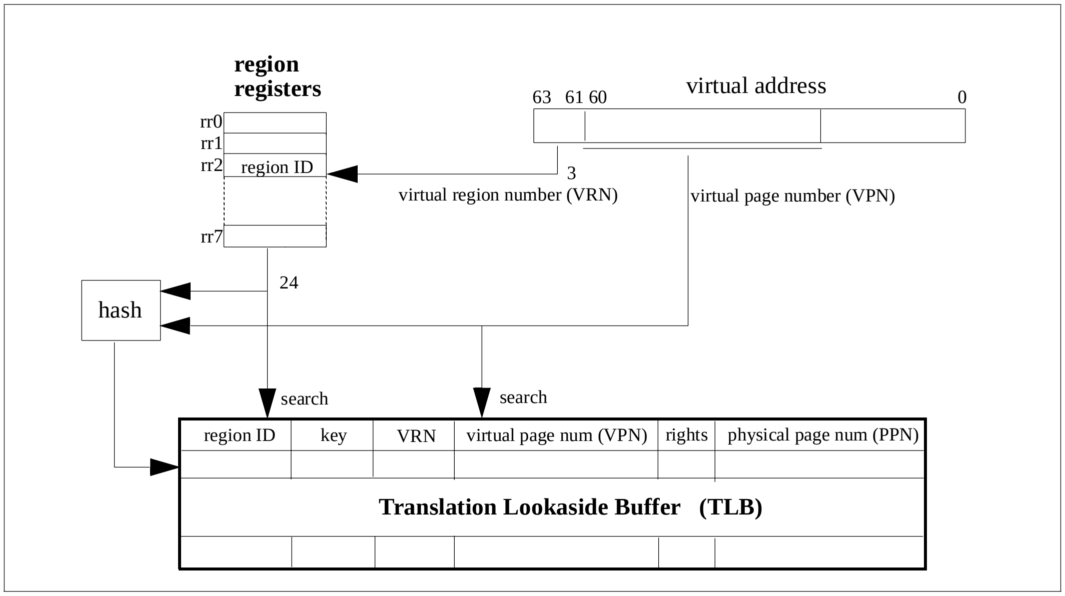
Instruction and data translation registers cannot be unified. DTR entries cannot be used by instruction references and ITR entries cannot be used by data references. ITR inserts and purges do not remove DTR entries. DTR inserts and purges do not remove ITR entries.

4.1.1.4 Purge Behavior of TLB Inserts and Purges

Translations contained in the translation caches (TC) and translation registers (TR) are maintained in a consistent state by ensuring that TLB insertions remove existing overlapping entries before new TR or TC entries are installed. Similarly, TLB purges that partially or fully overlap with existing translations may remove all overlapping entries. In this context, “overlap” refers to two translations with the same region identifier (but not necessarily identical virtual region numbers), and with partially or fully overlapping virtual address ranges (determined by the virtual address and the page size). Examples are: two 4K-byte pages at the same virtual address, or an 8K-byte page at virtual address 0x2000 and a 4K-byte page at 0x3000.

As described in Section 4.1, each TLB may contain a VRN field, and virtual address bits {63:61} may be used as part of the match for memory references (references other than inserts and purges). This binding of a translation to the VRN implies that a lookup of a given virtual address (region identifier/VPN pair) in either the translation cache or translation registers may result in a TLB miss if a memory reference is made through a different VRN (even if the region identifiers in the two region registers are identical). Some processor models may also omit the VRN field of the TLB, causing the TLB search on memory references to find an entry independent of VRN bits. However, all processor models are required, during translation cache purge and insert operations, to purge all possible translations matching the region identifier and virtual address regardless of the specified VRN.

Figure 4-4. Conceptual Virtual Address Searching for Inserts and Purges



A processor may overpurge translation cache entries; i.e., it may purge a larger virtual address range than required by the overlap. Since page sizes are powers of 2 in size and aligned on that same power of 2 boundary, purged entries can either be a superset of, identical to, or a subset of the specified purge range.

Table 4-1 defines the purge behavior of the different TLB insert and purge instructions.

Table 4-1. Purge Behavior of TLB Instructions

TLB Instructions	Translation Cache		Translation Registers	
	Instruction	Data	Instruction	Data
`itc.i`	Must purge[a]	May purge[b]	Machine Check[c]	Must not purge[d]
`itr.i`	Must purge	May purge	Machine Check	Must not purge
`itc.d`	May purge	Must purge	Must not purge	Machine Check
`itr.d`	May purge	Must purge	Must not purge	Machine Check
`ptc.l`	Must purge	Must purge	Machine Check	Machine Check
`ptc.g`, `ptc.ga` (local)[e]	Must purge	Must purge	Machine Check	Machine Check
`ptc.g`, `ptc.ga` (remote)[e]	Must purge	Must purge	Must not purge Must not Machine Check[f]	Must not purge Must not Machine Check
`ptc.e`	Must purge	Must purge	Must not purge	Must not purge
`ptr.i`	Must purge	May purge	Must purge	Must not purge
`ptr.d`	May purge	Must purge	Must not purge	Must purge

[a] Must purge: requires that all partially or fully overlapped translations are removed prior to the insert or purge operation.

b. May purge: indicates that a processor may remove partially or fully overlapped translations prior to the insert or purge operation. However, software must not rely on the purge.

c. Machine Check: indicates that a processor will cause a Machine Check abort if an attempt is made to insert or purge a partially or fully overlapped translation. The machine check abort may not be delivered synchronously with the TLB insert or purge operation itself, but is guaranteed to be delivered, at the latest, on a subsequent instruction serialization operation.

d. Must not purge: the processor does not remove (or check for) partially or fully overlapped translations prior to the insert or purge operation. Software can rely on this behavior.

e. `ptc.g`, `ptc.ga`: two forms of global TLB purges are distinguished: local and remote. The local form indicates that the `ptc.g` or `ptc.ga` was initiated on the local processor. The remote form indicates that this is an incoming TLB shoot-down from a remote processor.

f. Must not Machine Check: Remote `ptc.g` or `ptc.ga` operations must not cause local translation registers to be purged. Remote `ptc.g` or `ptc.ga` operations must not cause the local processor to machine check.

4.1.1.5 Translation Insertion Format

Figure 4-5 shows the register interface to insert entries into the TLB. TLB insertions are performed by issuing the Insert Translation Cache (`itc.d`, `itc.i`) and Insert Translation Registers (`itr.d`, `itr.i`) instructions. The first 64-bit field containing the physical address, attributes and permissions is supplied by a general purpose register operand. Additional protection key and page size information is supplied by the Interruption TLB Insertion Register (ITIR). The Interruption Faulting Address register (IFA) specifies the virtual address for instruction and data TLB inserts. ITIR and IFA are defined in "Control Registers" on page 2:24. The upper 3 bits of IFA (VRN bits{63:61}) select a virtual region register that supplies the RID field for the TLB entry. The RID of the selected region is tagged to the translation as it is inserted into the TLB. If reserved fields or reserved encodings are used, a Reserved Register Field fault is raised on the insert instruction.

Software must issue an instruction serialization operation to ensure installs into the ITLB are observed by dependent instruction fetches and a data serialization operation to ensure installs into the DTLB are observed by dependent memory data references.

Figure 4-5. Translation Insertion Format

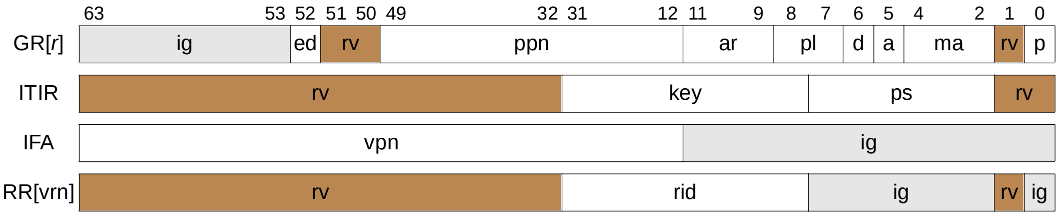


Table 4-2 describes all the translation interface fields.

Table 4-2. Translation Interface Fields

TLB Field	Source Field	Description
rv	GR[*r*]{5,51:50}, ITIR{1:0,63:32}, RR[vrn]{1,63:32}	reserved
p	GR[*r*]{0}	Present bit – When 0, references using this translation cause an Instruction or Data Page Not Present fault. Most other fields are ignored by the processor, see Figure 4-6 for details. This bit is typically used to indicate that the mapped physical page is not resident in physical memory. The present bit is not a valid bit. For each TLB entry, the processor maintains an additional hidden valid bit indicating if the entry is enabled for matching.

Table 4-2. Translation Interface Fields (Continued)

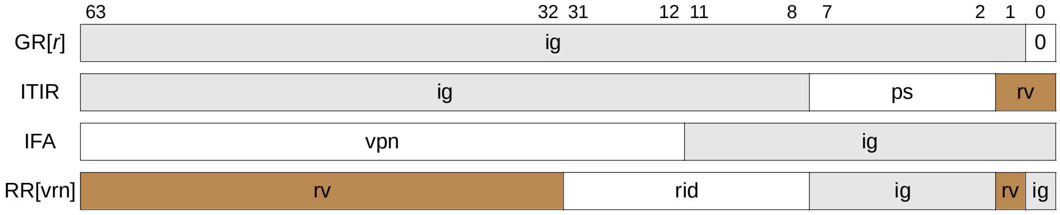
TLB Field	Source Field	Description
ma	GR[*r*]{4:2}	Memory Attribute – describes the cacheability, coherency, write-policy and speculative attributes of the mapped physical page. See “Memory Attributes” on page 2:63 for details.
a	GR[*r*]{5}	Accessed Bit – When 0 and PSR.da is 0, data references to the page cause a Data Access Bit fault. When 0 and PSR.ia is 0, instruction references to the page cause an Instruction Access Bit fault. When 0, IA-32 references to the page cause an Instruction or Data Access Bit fault. This bit can trigger a fault on reference for tracing or debugging purposes. The processor does not update the Accessed bit on a reference.
d	GR[*r*]{6}	Dirty Bit – When 0 and PSR.da is 0, Intel® Itanium™ store or semaphore references to the page cause a Data Dirty Bit fault. When 0, IA-32 store or semaphore references to the page cause a Data Dirty Bit fault. The processor does not update the Dirty bit on a write reference.
pl	GR[*r*]{8:7}	Privilege Level – Specifies the privilege level or promotion level of the page. See “Page Access Rights” on page 2:46 for complete details.
ar	GR[*r*]{11:9}	Access Rights – page granular read, write and execute permissions and privilege controls. See “Page Access Rights” on page 2:46 for details.
ppn	GR[*r*]{49:12}	Physical Page Number – Most significant bits of the mapped physical address. Depending on the page size used in the mapping, some of the least significant PPN bits are ignored.

ig	GR[*r*]{63:53}
	IFA{11:0},
	RR[vrn]{0,7:2}
		available – Software can use these fields for operating system defined parameters. These bits are ignored when inserted into the TLB by the processor.

ed	GR[*r*]{52}	Exception Deferral – For a speculative load that results in an exception, the speculative load’s instruction page TLB.ed bit is one of the conditions which determines whether the exception must be deferred. See “Deferral of Speculative Load Faults” on page 2:88 for complete details. This bit is ignored in the data TLB for data memory references and for IA-32 memory references.
ps	ITIR{7:2}	Page Size – Page size of the mapping. For page sizes larger than 4K bytes the low-order bits of PPN and VPN are ignored. Page sizes are defined as 2^{ps} bytes. See “Page Sizes” on page 2:47 for a list of supported page sizes.
key	ITIR{31:8}	Protection Key – uniquely tags the translation to a protection domain. If a translation’s Key is not found in the Protection Key Registers (PKRs), access is denied and a Data or Instruction Key Miss fault is raised. See “Protection Keys” on page 2:48 for complete details.
vpn	IFA{63:12}	Virtual Page Number – Depending on a translation’s page size, some of the least-significant VPN bits specified are ignored in the translation process. VPN{63:61} (VRN) selects the region register.
rid	RR[VRN].rid	Virtual Region Identifier – On TLB inserts the Region Identifier selected by VPN{63:61} (VRN) is used as additional match bits for subsequent accesses and purges (much like vpn bits).

The format in Figure 4-6 is defined for not-present translations (P-bit is zero).

Figure 4-6. Translation Insertion Format – Not Present



4.1.1.6 Page Access Rights

Page granular access controls use 4 levels of privilege. Privilege level 0 is the most privileged and has access to all privileged instructions; privilege level 3 is least privileged. Access (including IA-32) to a page is determined by the TLB.ar and TLB.pl fields, and by the privilege level of the access, as defined in Table 4-3. RSE fills and spills obtain their privilege level from RSC.pl; all other accesses (including IA-32) obtain their privilege level from PSR.cpl. Within each cell, “–” means no access, “R” means read access, “W” means write access, “X” means execute access, and “Pn” means promote PSR.cpl to privilege level “n” when an Enter Privileged Code (epc) instruction is executed.

Table 4-3. Page Access Rights

TLB.ar	TLB.pl	Privilege Level[a]				Description
		3	2	1	0	
0	3	R	R	R	R	read only
	2	–	R	R	R	
	1	–	–	R	R	
	0	–	–	–	R	
1	3	RX	RX	RX	RX	read, execute
	2	–	RX	RX	RX	
	1	–	–	RX	RX	
	0	–	–	–	RX	
2	3	RW	RW	RW	RW	read, write
	2	–	RW	RW	RW	
	1	–	–	RW	RW	
	0	–	–	–	RW	
3	3	RWX	RWX	RWX	RWX	read, write, execute
	2	–	RWX	RWX	RWX	
	1	–	–	RWX	RWX	
	0	–	–	–	RWX	
4	3	R	RW	RW	RW	read only / read, write
	2	–	R	RW	RW	
	1	–	–	R	RW	
	0	–	–	–	RW	
5	3	RX	RX	RX	RWX	read, execute / read, write, exec
	2	–	RX	RX	RWX	
	1	–	–	RX	RWX	
	0	–	–	–	RWX	
6	3	RWX	RW	RW	RW	read, write, execute / read, write
	2	–	RWX	RW	RW	
	1	–	–	RWX	RW	
	0	–	–	–	RW	
7	3	X	X	X	RX	exec, promote[b] / read, execute
	2	XP2	X	X	RX	
	1	XP1	XP1	X	RX	
	0	XP0	XP0	XP0	RX	

a. RSC.pl, for RSE fills and spills; PSR.cpl for all other accesses.
b. User execute only pages can be enforced by setting PL to 3.

Software can verify page level permissions by the `probe` instruction, which checks accessibility to a given virtual page by verifying privilege levels, page level read and write permission, and protection key read and write permission.

Execute-only pages (TLB.ar 7) can be used to promote the privilege level on entry into the operating system. User level code would typically branch into a promotion page (controlled by the operating system) and execute the Enter Privileged Code (`epc`) instruction. When `epc` successfully promotes, the next instruction group is executed at the target privilege level specified by the promotion page. A procedure return branch type (`br.ret`) can demote the current privilege level.

4.1.1.7 Page Sizes

A range of page sizes are supported to assist software in mapping system resources and improve TLB/VHPT utilization. Typically, operating systems will select a small range of fixed page sizes to implement virtual memory algorithms. Larger pages may be statically allocated. For example, large areas of the virtual address space may be reserved for operating system kernels, frame buffers, or memory-mapped I/O regions. Software may also elect to pin these translations, by placing them in the translation registers.

Table 4-4 lists insertable and purgeable page sizes that are supported by all processor models. Insertable page sizes can be specified in the translation cache, the translation registers, the region registers and the VHPT. Insertable page sizes can also be used as parameters to TLB purge instructions (`ptc.l`, `ptc.g`, `ptc.ga` or `ptr`). Page sizes that are purgeable only may only be used as parameters to TLB purge instructions.

Processors may also support additional insertable and purgeable page sizes. Please see the processor specific documentation for further information on the page sizes supported by the Itanium processor.

Table 4-4. Architected Page Sizes

	Page Sizes										
	4k	8k	16k	64k	256k	1M	4M	16M	64M	256M	4G
Insertable	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	-
Purgeable	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes

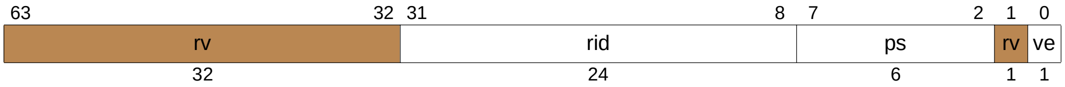
Page sizes are encoded in translation entries and region registers as a 6-bit encoded page size field. Each field specifies a mapping size of 2^N bytes, thus a value of 12 represents a 4K-byte page. If unimplemented page sizes are specified to an `itc`, `itr` or `mov` to region register instruction, a Reserved Register/Field fault is raised. If unimplemented page sizes are specified for a TLB purge instruction an implementation may raise a Machine Check abort, may under-purge translations up to ignoring the request, or may over-purge translations up to removal of all entries from the translation cache. If unimplemented page sizes are specified by a `ptc.g` or `ptc.ga` broadcast from another processor, an implementation may under-purge translations up to ignoring the request, or may over-purge translations up to removal of all entries from the translation cache. However, it must not raise a Machine Check abort.

Virtual and physical pages are aligned on the natural boundary of the page. For example, 4K-byte pages are aligned on 4K-byte boundaries, and 4 M-byte pages on 4 M-byte boundaries.

4.1.2 Region Registers (RR)

Associated with each of the 8 virtual regions is a privileged Region Register (RR). Each register contains a Region Identifier (RID) along with several other region attributes, see Figure 4-7. The values placed in the region register by the operating system can be viewed as a collection of process address space identifiers.

Figure 4-7. Region Register Format



Regions support multiple address space operating systems by avoiding the need to flush the TLB on a context switch. Sharing between processes is promoted by mapping common global or shared region identifiers into the region register working set of multiple processes. All IA-32 memory references are through region register 0.

Table 4-5 describes the region register fields. Region Identifier (rid) bits 0 through 17 must be implemented on all processor models. Some processor models may implement additional bits. Additional implemented bits must be contiguous and start at bit 18. Unimplemented bits are reserved. Please see the processor specific documentation for further information on the size of the Region Identifier implemented on the Itanium processor.

Table 4-5. Region Register Fields

Field	Bits	Description
rv	1,63:32	reserved
ve	0	VHPT Walker Enable – When 1, the VHPT walker is enabled for the region. When 0, disabled.
ps	7:2	Preferred page Size – Selects the virtual address bits used in hash functions for set-associative TLBs or the VHPT. Encoded as 2^{ps} bytes. The processor may make significant performance optimizations for the specified preferred page size for the region.[a]
rid	31:8	Region Identifier – During TLB inserts, the region identifier from the select region register is used to tag translations to a specific address space. During TLB/VHPT lookups, the region identifier is used to match translations and to distribute hash indexes among VHPT and TLB sets.

[a] For more details on the usage of this field, See "VHPT Hashing" on page 2:54.

Software must issue an instruction serialization operation to ensure writes into the region registers are observed by dependent instruction fetches and issue a data serialization operation for dependent memory data references.

4.1.3 Protection Keys

Protection Keys provide a method to restrict permission by tagging each virtual page with a unique protection domain identifier. The Protection Key Registers (PKR) represent a register cache of all protection keys required by a process. The operating system is responsible for management and replacement polices of the protection key cache. Before a memory access (including IA-32) is permitted, the processor compares a translation's key value against all keys contained in the PKRs. If a matching key is not found, the processor raises a Key Miss fault. If a matching Key is found, access to the page is qualified by additional read, write and execute protection checks specified by the matching protection key register. If these checks fail, a Key Permission fault is raised. Upon receipt of a Key Miss or Key Permission fault, software can implement the desired security policy for the protection domain. Figure 4-8 and Table 4-6 describe the protection key register format and protection key register fields.

Figure 4-8. Protection Key Register Format

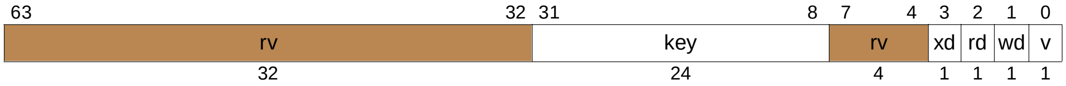


Table 4-6. Protection Register Fields

Field	Bits	Description
v	0	Valid – When 1, the Protection Register entry is valid and is checked by the processor when performing protection checks. When 0, the entry is ignored.
wd	1	Write Disable – When 1, write permission is denied to translations in the protection domain.
rd	2	Read Disable – When 1, read permission is denied to translations in the protection domain.
xd	3	Execute Disable – When 1, execute permission is denied to translations in the protection domain.
key	31:8	Protection Key – uniquely tags translation to a given protection domain.
rv	7:4,63:32	reserved

Processor models have at least 16 protection key registers, and at least 18-bits of protection key. Some processor models may implement additional protection key registers and protection key bits. Unimplemented bits and registers are reserved. Key registers have at least as many implemented key bits as region registers have rid bits. Additional implemented bits must be contiguous and start at bit 18. Please see the processor specific documentation for further information on the number of protection key registers and protection key bits implemented on the Itanium processor.

Software must issue an instruction serialization operation to ensure writes into the protection key registers are observed by dependent instruction fetches and a data serialization operation for dependent memory data references.

The processor ensures uniqueness of protection keys by checking new valid protection keys against all protection key registers during the move to PKR instruction. If a valid matching key is found in any PKR register, the processor invalidates the matching PKR register by setting PKR.v to zero, before performing the write of the new PKR register. The other fields in any matching PKR remain unchanged when it is invalidated.

Key Miss and Permission faults are only raised when memory translations are enabled (PSR.dt is 1 for data references, PSR.it is 1 for instruction references, PSR.rt is 1 for register stack references), and protection key checking is enabled (PSR.pk is one).

Data TLB protection keys can be acquired with the Translation Access Key (`tak`) instruction. Instruction TLB key values are not directly readable. To acquire instruction key values software should make provisions to read memory structures.

4.1.4 Translation Instructions

Table 4-7 lists translation instructions used to manage translations. Region registers, protection key registers and the TLBs are accessed indirectly; the register number is determined by the contents of a general register.

The processor does not ensure that modification of the translation resources is observed by subsequent instruction fetches or data memory references. Software must issue an instruction serialization operation before any dependent instruction fetch and a data serialization operation before any dependent data memory reference.

Table 4-7. Translation Instructions

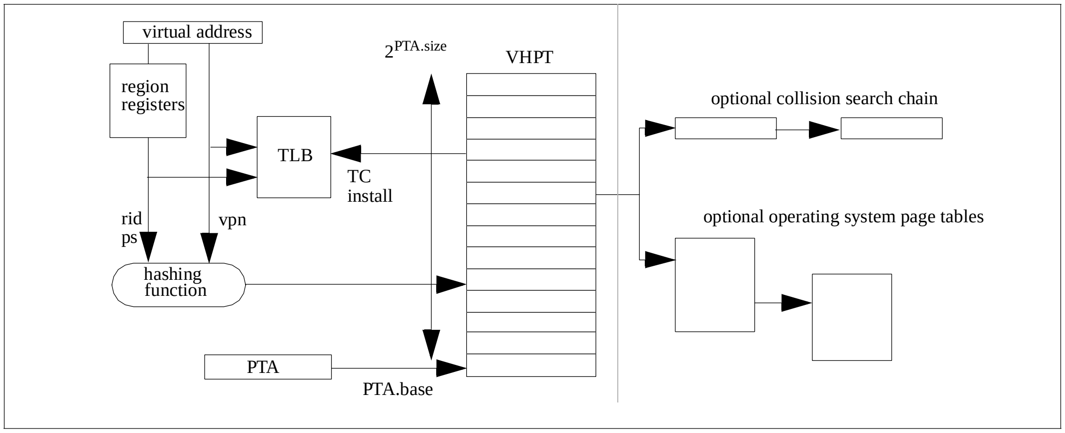
Mnemonic	Description	Operation	Instr. Type	Serialization Requirement
`mov rr[`r_3`] = `r_2	Move to region register	RR[GR[r_3]] = GR[r_2]	M	data/inst
`mov `r_1` = rr[`r_3`]`	Move from region register	GR[r_1] = RR[GR[r_3]]	M	none
`mov pkr[`r_3`] = `r_2	Move to protection key register	PKR[GR[r_3]] = GR[r_2]	M	data/inst
`mov `r_1` = pkr[`r_3`]`	Move from protection key register	GR[r_1] = PKR[GR[r_3]]	M	none
`itc.i `r_3	Insert instruction translation cache	ITC = GR[r_3], IFA, ITIR	M	inst
`itc.d `r_3	Insert data translation cache	DTC = GR[r_3], IFA, ITIR	M	data
`itr.i itr[`r_2`] = `r_3	Insert instruction translation register	ITR[GR[r_2]] = GR[r_3], IFA, ITIR	M	inst
`itr.d dtr[`r_2`] = `r_3	Insert data translation register	DTR[GR[r_2]] = GR[r_3], IFA, ITIR	M	data
`probe `r_1` = `r_3`, `r_2	Probe data TLB for translation		M	none
`ptc.l `r_3`, `r_2	Purge a translation from local processor instruction and data translation cache		M	data/inst
`ptc.g `r_3`, `r_2	Globally purge a translation from multiple processor's instruction and data translation caches		M	data/inst
`ptc.ga `r_3`, `r_2	Globally purge a translation from multiple processor's instruction and data translation caches and remove matching entries from multiple processor's ALATs		M	data/inst
`ptc.e `r_3	Purge local instruction and data translation cache of all entries		M	data/inst
`ptr.i `r_3`, `r_2	Purge instruction translation registers		M	inst
`ptr.d `r_3`, `r_2	Purge data translation registers		M	data
`tak `r_1` = `r_3	Obtain data TLB entry protection key		M	none
`thash `r_1` = `r_3	Generate translation's VHPT hash address		M	none
`ttag `r_1` = `r_3	Generate translation tag for VHPT		M	none
`tpa `r_1` = `r_3	Translate a virtual address to a physical address		M	none

4.1.5 Virtual Hash Page Table (VHPT)

The VHPT is an extension of the TLB hierarchy designed to enhance virtual address translation performance. The processor's VHPT walker can optionally be configured to search the VHPT for a translation after a failed instruction or data TLB search. The VHPT walker provides significant performance enhancements by reducing the rate of flushing the processor's pipelines due to a TLB Miss fault, and by providing speculative translation fills concurrent to other processor operations.

The VHPT, resides in the virtual memory space and is configurable as either the primary page table of the operating system or as a single large translation cache in memory (see Figure 4-9). Since the VHPT resides in the virtual address space, an additional TLB miss can be raised when the VHPT is referenced. This property allows the VHPT to also be used as a linear page table.

Figure 4-9. Virtual Hash Page Table (VHPT)



The processor does not manage the VHPT or perform any writes into the table. Software is responsible for insertion of entries into the VHPT (including replacement algorithms), dirty/access bit updates, invalidation due to purges and coherency in a multiprocessor system. The processor does not ensure the TLBs are coherent with the VHPT memory image.

If software needs to control the entries inserted into the TLB more explicitly, or programs the VHPT with differing mappings for the same virtual address range, it may need to take additional action to ensure forward progress. See "VHPT Searching" on page 2:52.

4.1.5.1 VHPT Configuration

The Page Table Address (PTA) register determines whether the processor is enabled to walk the VHPT, anchors the VHPT in the virtual address space, and controls VHPT size and configuration information. The VHPT can be configured as either a per-region virtual linear page table structure (8-byte short format) or as a single large hash page table (32-byte long format). No mixing of formats is allowed within the VHPT.

To implement a per-region linear page table structure an operating system would typically map the leaf page table nodes with small backing virtual translations. The size of the table is expanded to include all possible virtual mappings, effectively creating a large per-region flat page table within the virtual address space.

To implement a single large hash page table, the entire VHPT is typically mapped with a single large pinned virtual translation placed in the translation registers and the size of the table is reduced such that only a subset of all virtual mappings can be resident within the table. Operating systems can tune the size of the hash page table based on the size of physical memory and operating system performance requirements.

4.1.5.2 VHPT Searching

When enabled, the processor’s VHPT walker searches the VHPT for a translation after a failed instruction or data TLB search. The VHPT walker checks only the specific VHPT entry addressed by the short- or the long-format hash function, as selected by PTA.vf. If additional TLB misses are encountered during the VHPT access, a VHPT Translation fault is raised. If the region-based short-format VHPT entry contains no reserved bits or encodings, it is installed into the TLB, and the processor again attempts to translate the failed instruction or data reference. If the long-format VHPT entry’s tag specifies the correct region identifier and virtual address, and the entry contains no reserved bits or encodings, it is installed into the TLB, and the processor again attempts to translate the failed instruction or data reference. Otherwise the processor raises a TLB Miss fault. The translation is installed into the TLB even if its VHPT entry is marked as not present (p=0). Software may optionally search additional VHPT collision chains (associativities) or search for translations within the operating system’s primary page tables. Performance is optimized by placing frequently referenced translations within the VHPT structure directly searched by the processor.

The VHPT walker is optional on a given processor model. Software can neither assume the presence of a VHPT walker, nor that the VHPT walker will find a translation in the VHPT. The VHPT walker can abort a search at any time for implementation-specific reasons, even if the required translation entry is in the VHPT. Operating systems must regard the VHPT walker strictly as a performance optimization and must be prepared to handle TLB misses if the walker fails.

VHPT walks may be done speculatively by the processor's VHPT walker. Additionally, VHPT walks triggered by non-speculatively-executed instructions are not required to be done in program order. Therefore, if the walker is enabled and if the VHPT contains multiple entries that map the same virtual address range, software must set up these entries such that any of them can be used in the translation of any part of this virtual address range. Additionally, if software inserts a translation into the TLB which is needed for forward progress, and this translation has a smaller page size than the translation which would have been inserted on a VHPT walk for the same address, then software may need to disable the VHPT walker in order to ensure forward progress, since this inserted translation may be displaced by a VHPT walk before it can be used.

4.1.5.3 Region-based VHPT Short Format

The region-based VHPT short format shown in Figure 4-10 uses 8-byte VHPT entries to support a per-region linear page table configuration. To use the short-format VHPT, PTA.vf must be set to 0.

Figure 4-10. VHPT Short Format

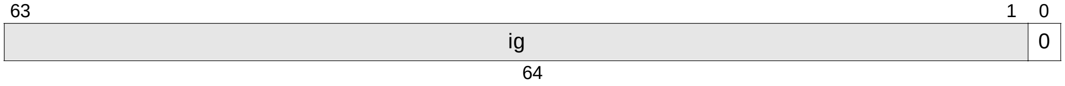
63 53	52	51 50	49 12	11 9	8 7	6	5	4 2	1	0
ig	ed	rv	ppn	ar	pl	d	a	ma	rv	p
11	1	2	38	3	2	1	1	3	1	1

See “Translation Insertion Format” on page 2:44 for a description of all fields. The VHPT walker provides the following default values when entries are installed into the TLB.

- Virtual Page Number – implied by the position of the entry in the VHPT. The hashed short-format entry is considered to be the matching translation.
- Region Identifiers are not specified in the short format. To ensure uniqueness, software must provide unique VHPT mappings per region. Region identifiers obtained from the referenced region register are tagged with the translation when inserted into the TLB.
- Page Size – specified by the accessed region's preferred page size (RR[VA{63:61}].ps)
- Protection Key – specified by the accessed region identifier value (RR[VA{63:61}].rid). As a result, all implementations must ensure that the number of implemented key bits is greater than or equal to the number of implemented region identifier bits.

If a translation is marked as not present, ignored fields are usable by software as noted in Figure 4-11.

Figure 4-11. VHPT Not-present Short Format



4.1.5.4 VHPT Long Format

The long-format VHPT uses 32-byte VHPT entries to support a single large virtual hash page table. To use the long-format VHPT, PTA.vf must be set to 1. The long format is a superset of the TLB insertion format, as noted in Figure 4-12, and specifies full translation information (including protection keys and page sizes). Additional fields are defined in Table 4-8. The long format is typically used to build the hash page table configuration.

Figure 4-12. VHPT Long Format

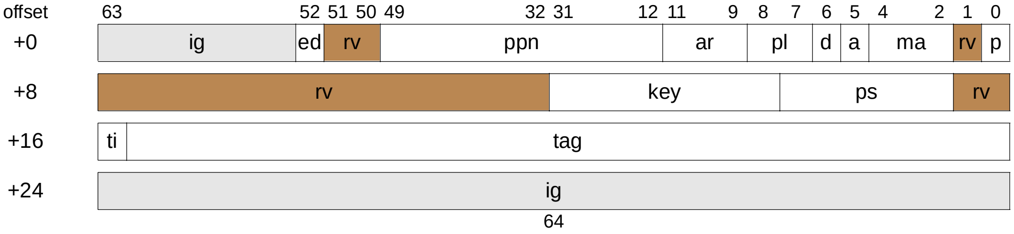
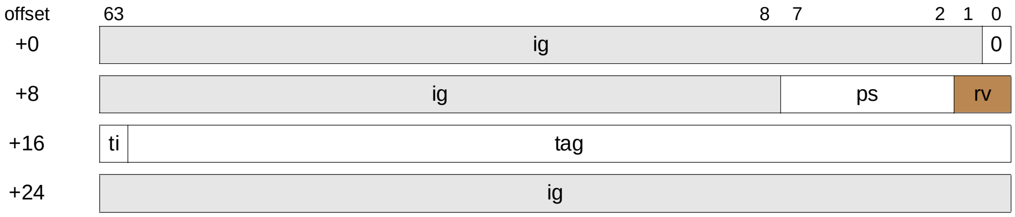


Table 4-8. VHPT Long-format Fields

Field	Offset	Description
tag	+16	Translation Tag – The tag, in conjunction with the VHPT hash index, is used to uniquely identify the translation. Tags are computed by hashing the virtual page number and the region identifier. See "VHPT Hashing" on page 2:54 for details on tag and hash index generation.
ti	+16	Tag Invalid Bit – If one, this bit of the tag indicates an invalid tag. On all processor implementations, the VHPT walker and the `ttag` instruction generate tags with the ti bit equal to 0. A VHPT entry with the ti bit equal to one will never be inserted into the processor's TLBs. Software can use the ti bit to invalidate long-format VHPT entries in memory.
ig	+24	available – field for software use, ignored by the processor. Operating systems may store any value, such as a link address to extend collision chains on a hash collision.

If a translation is marked as not present, ignored fields are usable by software as noted in Figure 4-13.

Figure 4-13. VHPT Not-present Long Format



For multiprocessor systems, atomic updates of long-format VHPT entries may be ensured by software as follows:

- Before making multiple non-atomic updates to a VHPT entry in memory, software is required to set its ti bit to one.
- After making multiple non-atomic updates to a VHPT entry in memory, software may clear its ti bit to zero to re-enable tag matches.

The updates to the VHPT entry in memory must be constrained to be observable only after the store that sets the ti bit to one is observable. This can be accomplished with a `mf` instruction, or by performing the updates to the VHPT entry with release stores. Similarly, the clearing of the ti bit must be constrained to be observable only after all of the updates to the VHPT entry are observable. This can be accomplished with a `mf` instruction, or by performing the clear of the ti bit with a release store.

4.1.6 VHPT Hashing

The processor provides two methods for software to determine a VHPT entry's address: the Translation Hash (`thash`) instruction, and the Interruption Hash Address (IHA) register defined on page 2:34. The virtual address of the VHPT entry is placed in the IHA register when a VHPT Translation or TLB fault is delivered. In the long format, IHA can be used as a starting address to scan additional collision chains (associativities) defined by the operating system or to perform a search in software. The `thash` instruction is used to generate a VHPT entry's address outside of interruption handlers and provides the same hash function that is used to calculate IHA.

`thash` produces a VHPT entry's address for a given virtual address and region identifier, depending on the setting of the PTA.vf bit. When PTA.vf=0, `thash` returns the region-based short-format index as defined in "Region-based VHPT Short-format Index" on page 2:54. When PTA.vf=1, `thash` returns the long-format hash as defined in "Long-format VHPT Hash" on page 2:55. The `ttag` instruction is only useful for long-format hashing, and generates a unique 64-bit ti/tag identifier that the processor's VHPT walker will check when it looks up a given virtual address and region identifier. Software should use the `ttag` instruction, and either the `thash` instruction or the IHA register when forming translation tags and hash addresses for the long-format VHPT. These resources encapsulate the implementation-specific long-format hashing functionality and improve performance.

4.1.6.1 Region-based VHPT Short-format Index

In the region-based short format, the linear page table for each region resides in the referenced region itself. As a result, the short-format VHPT consists of separate per-region page tables, which are anchored in each region by PTA.base{60:15}. For regions in which the VHPT is enabled, the operating system is required to maintain a per-region linear page table. As defined in Figure 4-14, the VHPT walker uses the virtual address, the region's preferred page size, and the PTA.size field to compute a linear index into the short-format VHPT.

Figure 4-14. Region-based VHPT Short-format Index Function

```
Mask = (1 << PTA.size) - 1;
VHPT_Offset = (VA{IMPL_VA_MSB:0} u>> RR[VA{63:61}].ps) << 3;
VHPT_Addr = (VA{63:61} << 61) |
    (((PTA.base{60:15} & ~Mask{60:15}) | (VHPT_Offset{60:15} &
        Mask{60:15})) << 15) |
    VHPT_Offset{14:0};
```

The size of the short-format VHPT (PTA.size) defines the size of the mapped virtual address space. The maximum architectural table size in the short format is 2^{52} bytes per region. To map an entire region (2^{61} bytes) using 4Kbyte pages, $2^{(61-12)} = 2^{49}$ pages must be mappable. A short-format VHPT entry is 8 bytes = 2^3 bytes large. As a result, the maximum table size is $2^{(61-12+3)} = 2^{52}$ bytes per region. If the short format is used to map an address space smaller than 2^{61}, a smaller short-format table (PTA.size<52) can be used. Mapping of an address space of 2^n with 4KByte pages requires a minimum PTA.size of (n-9).

In the short format, the `thash` instruction returns the region-based short-format index defined in Figure 4-14. The `ttag` instruction is not used with the short format. VHPT translation and TLB miss faults write the IHA register with the region-based short-format index defined in Figure 4-14.

4.1.6.2 Long-format VHPT Hash

The long-format VHPT is a single large contiguous hash table that resides in the region defined by PTA.base. As defined in Figure 4-15, the VHPT walker uses the virtual address, the region identifier, the region's preferred page size, and the PTA.size field to compute a hash index into the long-format VHPT. PTA.base{63:15} defines the base address and the region of the long-format VHPT. PTA.size reflects the size of the hash table, and is typically set to a number significantly smaller than 2^{64}; the exact number is based on operating system performance requirements.

Figure 4-15. VHPT Long-format Hash Function

```
Mask = (1 << PTA.size) - 1;
HPN = VA{IMPL_VA_MSB:0} u>> RR[VA{63:61}].ps;
Hash_Index = tlb_vhpt_hash_long(HPN,RR[VA{63:61}].rid);
// model-specific hash function
VHPT_Offset = Hash_Index << 5;
VHPT_Addr = (PTA.base{63:61} << 61) |
    (((PTA.base{60:15} & ~Mask{60:15}) | (VHPT_Offset{60:15}
    & Mask{60:15})) << 15) | VHPT_Offset{14:0};
```

The long-format hash function (*`tlb_vhpt_hash_long`*) and long-format tag generation function are implementation specific. However, on all processor models the hash and tag functions must exclude the virtual region number (virtual address bits VA{63:61}) from the hash and tag computations. This ensures that a unique 85-bit global virtual address hashes to the same VHPT hash address, regardless of which region the address is mapped to. All processor implementations guarantee that the most significant bit of the tag (ti bit) is zero for all valid tags. The hash index and tag together must uniquely identify a translation. The processor must ensure that the indices into the hashed table, the region's preferred page size, and the tag specified in an indexed entry can be used in a reverse hash function to uniquely regenerate the region identifier and virtual address used to generate the index and tag. This must be possible for all supported page sizes, implemented virtual addresses and legal values of region identifiers. A hash function is reversible if using the hash result and all but one input produces the missing input as the result of the reverse hash function. The easiest hash function and reverse hash function is a simple XOR of bits. To ensure uniqueness, software must follow these rules:

1. Software must use only one preferred page size for each unique region identifier at any given time; otherwise, processor operation is undefined.
2. All tags for translations within a given region must be created with the preferred page size assigned to the region; otherwise, processor operation is undefined.
3. Software is not allowed to have pages in the VHPT that are smaller than the preferred page size for the region; otherwise, processor operation is undefined. Software can specify a page with a page size larger than the preferred page size in the VHPT, but tag values for the entries representing that page size must be generated using the preferred page size assigned to that region.
4. To reuse a region identifier with a different preferred page size, software must first ensure that the VHPT contains no insertable translations for that rid, purge all translations for that rid from all processors that may have used it, and then update the region register with the new preferred page size.

4.1.7 VHPT Environment

The processor’s VHPT walker can optionally be configured to search the VHPT for a translation after a failed instruction or data TLB search. The VHPT walker is enabled for different types of references under the following conditions:

- Data and non-access references (including IA-32): PTA.ve=1, and RR[VA{63:61}].ve=1, and PSR.dt=1.
- Instruction fetches (including IA-32): PTA.ve=1, and RR[VA{63:61}].ve=1, and PSR.dt=1, and PSR.it=1, and PSR.ic=1.
- RSE references: PTA.ve=1, and RR[VA{63:61}].ve=1, and PSR.dt=1, and PSR.rt=1.

If the walker is not enabled, and an attempt is made to reference the VHPT, an Alternate Instruction/Data TLB Miss fault is raised. The remainder of this section assumes that the VHPT is enabled.

Region registers must support all implemented page sizes so software can use IHA, `thash` and `ttag` to manage the VHPT. `thash` and `ttag` are defined to operate on all page sizes supported by the translation cache, regardless of the VHPT walker’s supported page sizes. The PTA register must be implemented on processor models that do not implement a VHPT walker. Software must ensure PTA is initialized and serialized before issuing `ttag`, `thash`, before enabling the VHPT walker or issuing a reference that may cause a VHPT walk. The minimum VHPT size is 32KBytes (PTA.size=15), and operating systems must ensure that the VHPT is aligned on the natural boundary of the structure; otherwise, processor operation is undefined. For example, a 64K-byte table must be aligned on a 64K-byte boundary.

VHPT walker references to the VHPT are performed at privilege level 0, regardless of the state of PSR.cpl. VHPT byte ordering is determined by the state of DCR.be. When DCR.be=1, VHPT walker references are performed using big-endian memory formats; otherwise, VHPT walker references are little-endian. A long-format VHPT reference is matched against the data break-point registers as a 32-byte reference.

The VHPT is accessed by the processor only if the VHPT is virtually mapped into cacheable memory areas. The walker may access the VHPT speculatively, i.e., references may be performed that are not required by an in-order execution of the program. Any VHPT or TLB faults encountered during a VHPT walker's search are not reported until the faulting translation is required by an in-order execution of the program. If the VHPT is mapped into non-cacheable memory areas the VHPT is not referenced, and all TLB misses result in an Instruction/Data TLB Miss fault.

The VHPT walker will abort the search and deliver an Instruction/Data TLB Miss fault if an attempt is made to install translations that have reserved bits or encodings, or if the translation mapping the VHPT would have taken one of the following faults: Data Page Not Present, Data NaT Page Consumption, Data Key Miss, Data Key Permission, Data Access Bit, or Data Debug. The VHPT walker may abort a search and deliver an Instruction/Data TLB Miss fault at any time for implementation-specific reasons.

The processor's VHPT walker is required to read and insert VHPT entries from memory atomically (an 8-byte atomic read-and-insert for short format, and a 32-byte atomic read-and-insert for long format). Some implementation strategies for achieving this atomicity are as follows:

- If the walker performs its VHPT read with multiple cache accesses which are not done as an atomic unit, and if an update to part of the entry that is being installed is made in-between these multiple reads, the walker must abort the insert and deliver an Instruction/Data TLB Miss.
- If the walker performs its VHPT read and the insertion of the entry into the TLB as separate actions, and not as an atomic unit, and if an update to part of the entry that is being installed is made in-between the read and the insert, the walker must either abort the insert and deliver an Instruction/Data TLB Miss, or ignore the update and install the complete old entry.
- If the purge address range of a TLB purge operation (`ptc.l`, `ptc.e`, local or remote `ptc.g` or `ptc.ga`, `ptr.i`, or `ptr.d`) overlaps the virtual address the walker is attempting to insert, then the walker must either abort the insert and deliver an Instruction/Data TLB Miss, or delay the purge operation until after the walker either completes the insertion or aborts the walk.

The RSE can only raise a VHPT fault on a mandatory RSE spill/fill operation as defined for successful execution of an `alloc`, `loadrs`, `flushrs`, `br.ret` or `rfi` instruction. Eager RSE operations may generate speculative VHPT walks provided encountered faults are not reported.

Data TLB Miss faults encountered during a VHPT walk are permitted and, when PSR.ic=1, are converted into a VHPT Translation fault as defined in the next section.

4.1.8 Translation Searching

The general sequence of searching the TLB and VHPT is shown in Figure 4-16. On a failed TLB search, if the VHPT walker is disabled for the referenced region an Alternate Instruction/Data TLB Miss fault is raised. If the VHPT walker is enabled for the referenced region, the VHPT is accessed to locate the missing translation. See "VHPT Environment" on page 2:56. If additional TLB misses are encountered during the VHPT walker's references, a VHPT Translation fault is raised. If the VHPT walker does not find the required translation in the VHPT or the search is aborted, an

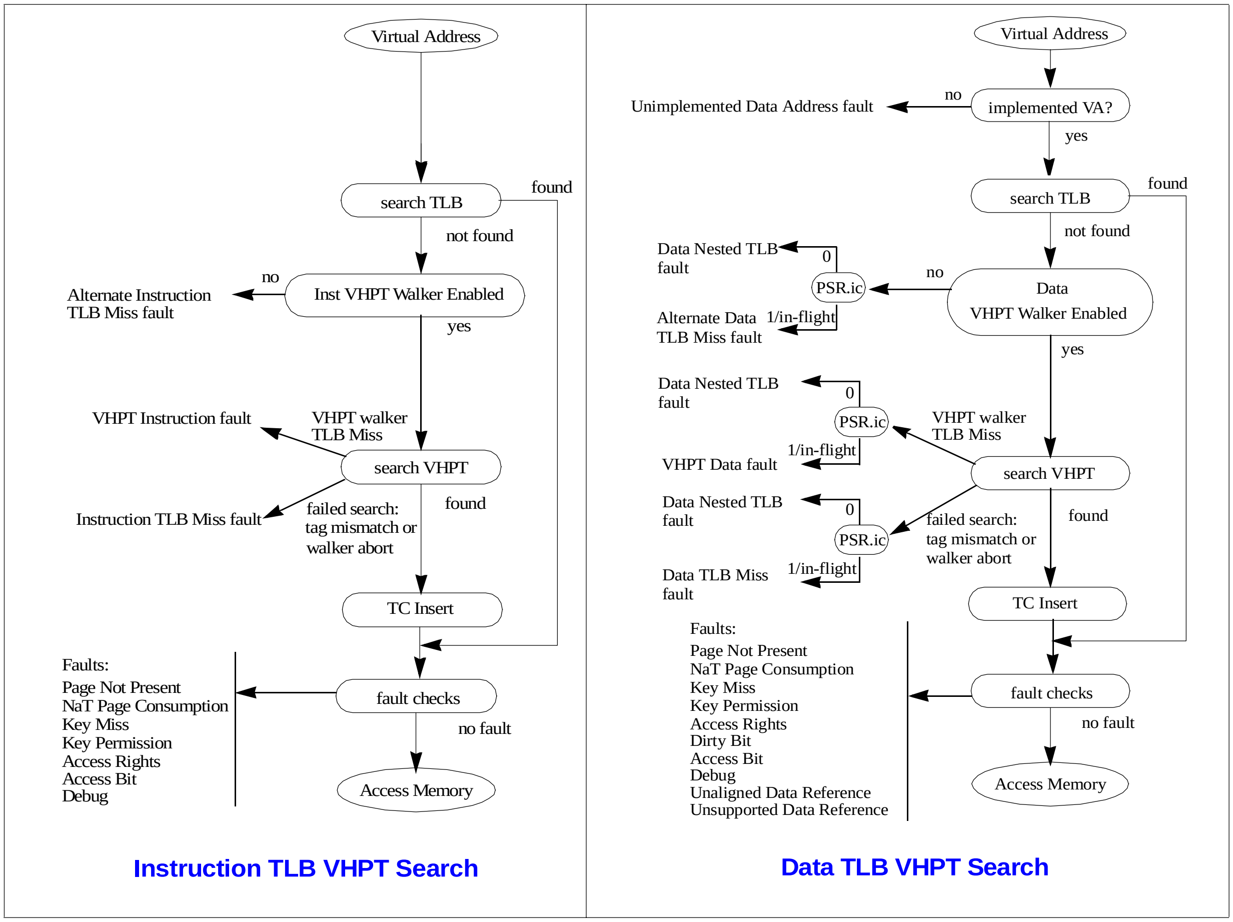
Instruction/Data TLB Miss fault is raised. Otherwise the entry is loaded into the ITC or DTC. Provided the above fault conditions are not detected, the processor may load the entry into the ITC or DTC even if an in-order execution of the program did not require the translation.

The VHPT walker's inserts into the TC follow the same purge-before-insert rules that software inserts are subject to (see Table 4-1, "Purge Behavior of TLB Instructions," on page 2:43). VHPT walker inserts into the DTC behave like `itc.d`; VHPT walker inserts into the ITC behave like `itc.i`. If an instruction reference results in a VHPT walk that misses in the data TLB, the DTC insert for the translation for the VHPT acts like an `itc.d`. VHPT walker insertions of entries that exist in TRs are not allowed. Specifically, the VHPT walker may search for any virtual address, but if the address is mapped by a TR, it must not be inserted into the TC. Software must not create overlapping translations in the VHPT that are larger than a currently existing TR translation. A VHPT walker insert may result in a Machine Check abort if an overlap exists between a TR and the inserted VHPT entry.

After the translation entry is loaded, additional TLB faults are checked; these include in priority order: Page Not Present, NaT page Consumption, Key Miss, Key Permission, Access Rights, Access Bit, and Dirty Bit faults. Table 4-9 describes the TLB and VHPT walker related faults.

On a failed TLB/VHPT search, the processor loads interruption registers and translation defaults as defined in "Interruption Vector Descriptions" on page 2:147 defining the parameters of the translation fault. Provided the operating system accepts the defaults provided, only the physical address portion of a TLB entry need be provided on a TLB insert.

Figure 4-16. TLB/VHPT Search



Instruction TLB VHPT Search

Data TLB VHPT Search

Table 4-9. TLB and VHPT Search Faults

Fault	Description
VHPT Instruction/Data	Raised if there is an additional TLB miss when the VHPT walker attempts to access the VHPT. Typically used to construct leaf table mappings for linear page table configurations.
Alternate Instruction/Data TLB Miss	Raised when the VHPT walker is not enabled and an instruction or data reference causes a TLB miss. For example, the VHPT walker can be disabled within a given virtual region so region-specific translation algorithms can be utilized.
Instruction/Data TLB Miss	Raised when the VHPT walker is enabled, but the processor: • Cannot locate the required VHPT entry, or • The processor aborts the VHPT search for implementation-specific reasons, or • The VHPT walker is not implemented, or • The referenced region specifies a non-supported VHPT preferred page size, or • Reserved fields or unimplemented PPN bits are used in the translation, or • The hash address falls into unimplemented virtual address space, or • The hash address matches a data debug register. Instruction/Data TLB Miss handlers are essentially software walkers of the VHPT.
Data Nested TLB	Raised when a Data TLB Miss, Alternate Data TLB Miss, or VHPT Data Translation fault occurs and PSR.ic is 0 and not in-flight (e.g., fault within a TLB miss handler). Data Nested TLB faults enable software to avoid overheads for potential data TLB Miss faults.
Instruction/Data Page Not Present	The referenced translation's P-bit is 0.
Instruction/Data NaT Page Consumption	A non-speculative load, store, mandatory RSE load/store, execution on, or semaphore operation accesses a page marked with the physical memory attribute NaTPage. See "Not a Thing Attribute (NaTPage)" on page 2:72 for details.
Instruction/Data Key Miss	The referenced translation's permission key is not present in the set of valid protection key registers.
Instruction/Data Key Permission	The referenced translation is denied read, write, execute permissions by the matching protection key registers.
Instruction/Data Access Rights	Page granular read, write, execute and privilege level accesses are denied.
Data Dirty Bit	The referenced translation's Dirty bit is 0 on a store or semaphore operation.
Instruction/Data Access Bit	The referenced translation's Access bit is 0.

4.1.9 32-bit Virtual Addressing

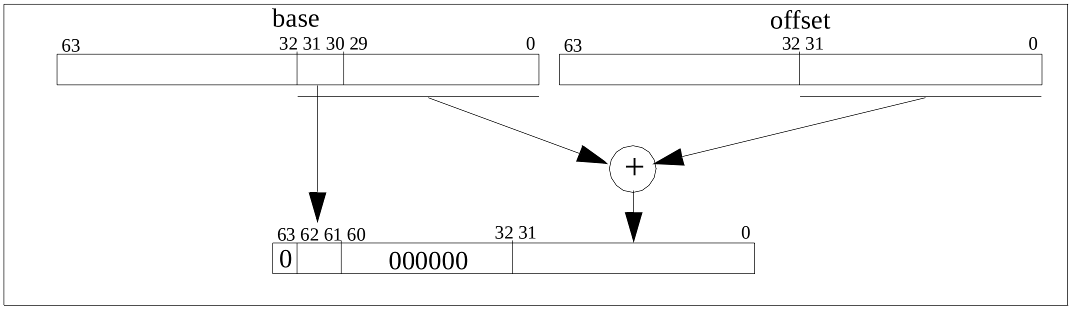
32-bit virtual data addressing is supported in the Itanium instruction set architecture by three models: zero-extension, sign-extension, and pointer "swizzling". IA-32 memory references use the zero-extension model, all IA-32 32-bit virtual linear addresses are zero extended into the 64-bit virtual address space.

The zero-extension model performs address computations with the `add` and `shladd` instructions while software ensures that the upper 32-bits are always zeros. This model constrains 32-bit virtual addressing to virtual region zero. In this model, regions 1 to 7 are accessible only by 64-bit addressing.

In the sign-extension model, software ensures that the upper 32-bits of a virtual address are always equal to bit 31. Address computations use the `add`, `shladd`, and `sxt` instructions. This model splits the 32 bit address space into 2 halves that are spread into 2^{31} bytes of virtual regions 0 and 7 within the 64-bit virtual address space. In this model, regions 2 to 6 are accessible only by 64-bit addressing.

The pointer “swizzling” model performs address computations with the `addp4`, and `shladdp4` instructions. These instructions generate a 32-bit address within the 64-bit virtual address space as shown in Figure 4-17. The 32-bit virtual address space is divided into 4 sections that are spread into 2^{30} bytes of virtual regions 0 to 3 within the 64-bit virtual address space. In this model, regions 4 to 7 are accessible only by 64-bit addressing.

Figure 4-17. 32-bit Address Generation using addp4



In the pointer “swizzling” model, mappings within each region do not necessarily start at offset zero, since the upper 2-bits of a 32-bit address serve both as the virtual region number and an offset within each region. Virtual address bits{62:61} do not participate in the address addition, therefore some regions may be effectively larger than 2^{30} bytes due to the addition of a 32-bit offset and lack of a carry into bits{62:61}. Note that the conversion is non-destructive: a converted 64-bit pointer can be used as a 32-bit pointer. Flat 31 or 32 bit address spaces can be constructed by assigning the same region identifier to contiguous region registers. Branches into another 2^{30}-byte region are performed by first calculating the target address in the 32-bit virtual space and then converting to a 64-bit pointer by `addp4`. Otherwise, branch targets will extend above the 2^{30} byte boundary within the originating region.

4.1.10 Virtual Aliasing

Virtual aliasing (two or more virtual pages mapped to the same physical page) is functionally supported for memory references (including IA-32), however performance may be degraded on some processor models where the distance between virtual aliases is less than 1 MB. To avoid any possible performance degradation, software is advised to use aliases whose virtual addresses differ by an integer multiple of 1 MB. The processor ensures cache coherency and data dependencies in the presence of an alias. Stores using a virtual alias followed by a load with another alias to the same physical location see the effects of prior stores to the same physical memory location.

To support advanced loads in the presence of a virtual alias, the processor ensures that the Advanced Load Address Table (ALAT) is resolved using physical addresses and is coherent with physical memory. For details, please refer to "Detailed Functionality of the ALAT and Related Instructions" on page 2:58.

4.2 Physical Addressing

Objects in memory and I/O occupy a common 63-bit physical address space that is accessed using byte addresses. Accesses to physical memory and I/O may be performed via virtual addresses mapped to the 63-bit physical address space or by direct physical addressing. Current page table formats allow for mapping virtual addresses into 50 bits of physical address space (on processor implementations that support this many physical address bits). Future extensions to the page table formats will allow larger mappings, up to the full 63 bits of physical address space.

Physical addressing for instruction references (including IA-32) is enabled when PSR.it is 0, data references (including IA-32) when PSR.dt is 0, and register stack references when PSR.rt is 0.

While software views the physical addressing as being 63-bits, implementations may implement between 32 and 63 physical address bits. All processor models must implement a contiguous set of physical address bits starting at bit 32 and continuing upwards. Please see the processor specific documentation for further information on the number of physical address bits implemented on the Itanium processor. Implementations must validate that memory references are performed to implemented physical address bits. Instruction references to unimplemented physical addresses result in an Unimplemented Instruction Address Trap on the last valid instruction. Data references to unimplemented physical addresses result in an Unimplemented Data Address fault. Memory references to unpopulated address ranges result in an asynchronous Machine Check abort, when the platform signals a transaction time-out. Exact machine check behavior is model specific.

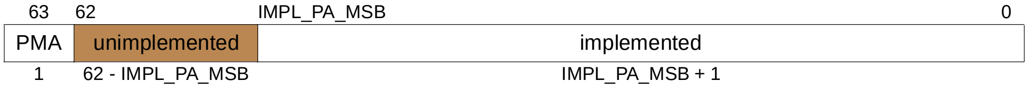
4.3 Unimplemented Address Bits

Based on the processor model, some physical and/or virtual address bits may not be implemented. Regardless of the number of implemented address bits, all general purpose, branch, control and application registers implement all 64 register bits on all processors. Similarly, regardless of the number of implemented address bits, data and instruction breakpoint registers must implement all 64 address bits and all 56 mask bits on all processors.

4.3.1 Unimplemented Physical Address Bits

As shown in Figure 4-18, a 64-bit physical address consists of three fields: physical memory attribute (PMA), unimplemented and implemented bits.

Figure 4-18. Physical Address Bit Fields



All processor models implement at least 32 physical address bits, bits 0 to 31, plus the physical memory attribute bit. Additional implemented physical bits must be contiguous starting at bit 32. IMPL_PA_MSB is the implementation-specific position of the most significant implemented physical address bit. In a processor that implements all physical address bits, IMPL_PA_MSB is 62. Please see the processor specific documentation for further information on the number of physical address bits implemented on the Itanium processor.

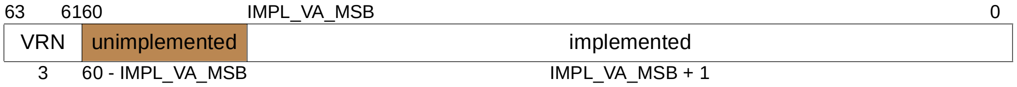
If unimplemented physical address bits are set by software, an Unimplemented Data Address fault is raised during the TLB insert instructions (`itc`, `itr`). Inserts performed by the VHPT walker, as noted in "VHPT Hashing" on page 2:54, abort the VHPT search if unimplemented or reserved fields are used. For translations marked as Not-Present (TLB.p is 0), the processor does not check the validity of PPN and some reserved bits as noted in Figure 4-6.

When a processor model does not implement all physical address bits, the missing bits are defined to be zero. Physical addresses in which bits PA{62:min(IMPL_PA_MSB+1,62)} are not zero are considered "unimplemented" physical addresses on that processor model. Physical addresses are checked for correctness on use by ensuring that PA{62:min(IMPL_PA_MSB+1,62)} bits are zero.

4.3.2 Unimplemented Virtual Address Bits

As shown in Figure 4-19, a 64-bit virtual address consists of three fields: virtual region number (VRN), unimplemented and implemented bits.

Figure 4-19. Virtual Address Bit Fields



All processor models provide three VRN bits in VA{63:61}. IMPL_VA_MSB is the implementation-specific bit position of the most significant implemented virtual address bit. In addition to the three VRN bits, all processor models implement at least 51 virtual address bits; i.e., the smallest IMPL_VA_MSB is 50. In a processor that implements all 64 virtual address bits IMPL_VA_MSB is 60. Please see the processor specific documentation for further information on the number of virtual address bits implemented on the Itanium processor.

When a processor model does not implement all virtual address bits, the missing bits are defined to be a sign-extension of VA{IMPL_VA_MSB}. Virtual addresses in which bits VA{60:min(IMPL_VA_MSB+1,60)} do not match VA{IMPL_VA_MSB} are considered "unimplemented" virtual addresses on that processor model. Virtual addresses are checked for correctness on use by ensuring that VA{60:min(IMPL_VA_MSB+1,60)} bits are identical to VA{IMPL_VA_MSB}.

4.3.3 Instruction Behavior with Unimplemented Addresses

The use of an unimplemented address affects instruction execution as follows:

- Non-speculative memory references (non-speculative loads, stores, and semaphores), the following non-access references: `fc`, `tpa`, `lfetch.fault`, and `probe.fault`, and mandatory RSE operations to unimplemented addresses result in an Unimplemented Data Address fault.

- Virtual addresses used by instruction and data TLB purge/insert operations are checked, and if the base address (register r3 of the purge, IFA for inserts) targets an unimplemented virtual address, a Unimplemented Data Address fault is raised. The page size of the insert or purge is ignored.
- Speculative loads from unimplemented addresses always return a NaT bit in the target register.
- A non-faulting `probe` instruction to an unimplemented address returns zero in the target register.
- A `tak` instruction to an unimplemented address returns one in the target register.
- A non-faulting `lfetch` to an unimplemented address is silently ignored.
- Eager RSE operations to unimplemented addresses do not fault.
- A branch (taken or fall-through), an `rfi`, or an instruction fetch to an unimplemented virtual address results in an Unimplemented Instruction Address Trap on the branch, the `rfi`, or the last executed instruction.
- When `ptc.g` or `ptc.ga` operations place a virtual address on the bus, the virtual address is sign-extended to a full 64-bit format. If an incoming `ptc.g` or `ptc.ga` presents a virtual address base that targets an unimplemented virtual address, the upper (unimplemented) virtual address bits are dropped, and the purge is performed with the truncated address.

4.4 Memory Attributes

When virtual addressing is enabled, memory attributes defining the speculative, cacheability and write-policies of the virtually mapped physical page are defined by the TLB. When physical addressing is enabled, memory attributes are supplied as described in "Physical Addressing Memory Attributes" on page 2:64.

4.4.1 Virtual Addressing Memory Attributes

For virtual memory references, the memory attribute field of each virtual translation describes physical memory properties as shown in Table 4-10.

Table 4-10. Virtual Addressing Memory Attribute Encodings

Attribute	Mnemonic	ma	Cacheability	Write Policy	Speculation	Coherent[a] with Respect to
Write Back	WB	000	Cacheable	Write back	Non-sequential & speculative	WB, WBL
Write Coalescing	WC	110	Uncacheable	Coalescing		Not MP coherent[b]
Uncacheable	UC	100		Non-coalescing	Sequential & non-speculative	UC, UCE
Uncacheable Exported	UCE	101				
Reserved[c]		001				
Reserved		010 011				
NaTPage	NaTPage	111	Cacheable	N/A	Speculative	N/A

a. The Coherency column in this table refers to multiprocessor coherence on normal, side-effect free memory. The data dependency rules defined in "Memory Access Ordering" on page 2:65 ensure uni-processor coherence for the memory attributes listed in each row.

b. WC is not MP coherent w.r.t. any memory attribute, but is uni-processor coherent w.r.t. itself.
c. This memory attribute is reserved for Software use.

The attribute UCE is identical to UC except when executing an `fetchadd` instruction. UCE enables the exporting of the `fetchadd` instruction outside the processor. Support for UCE is model-specific; see "Effects of Memory Attributes on Memory Reference Instructions" on page 2:72 for details.

Insert TLB instructions (`itc`, `itr`) that attempt to insert reserved memory attributes (Table 4-10) into the TLB raise Reserved Register/Field faults. External system operation is undefined if software inserts a memory attribute supported by the processor but not supported by the external system.

If software modifies the memory attributes for a page, it must follow the attribute transition requirements in Section 4.4.11, "Memory Attribute Transition" on page 2:74.

It is recommended that processor models report a Machine Check abort if the following memory attribute aliasing is detected:

- cache hit on an uncacheable page, other than as the target of a local or remote flush cache (`fc`) instruction (see "Effects of Memory Attributes on Memory Reference Instructions" on page 2:72).

4.4.2 Physical Addressing Memory Attributes

The selection of memory attributes for physical addressing is selected by bit 63 of the address contained in the address base register as shown in Figure 4-20 and Table 4-11.

Figure 4-20. Physical Addressing Memory

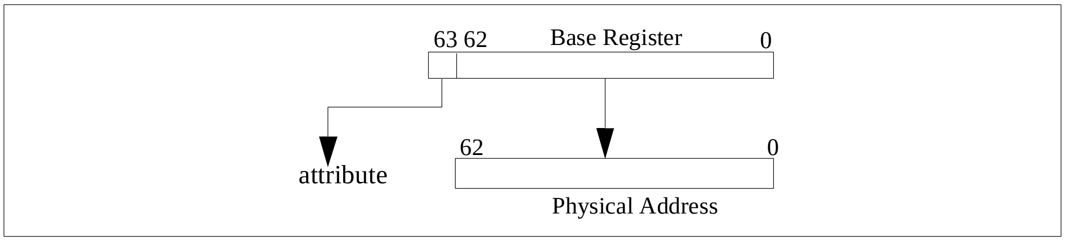


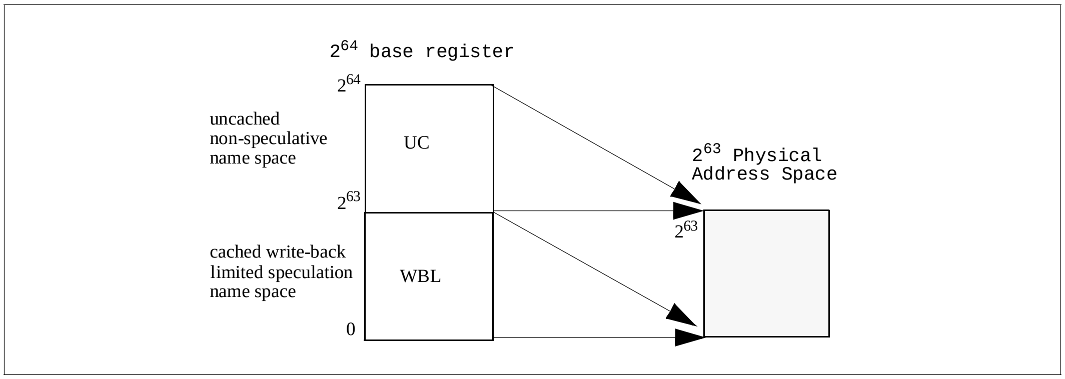
Table 4-11. Physical Addressing Memory Attribute Encodings

Bit{63}	Mnemonic	Cacheability	Write Policy	Speculation	Coherent[a] with respect to
0	WBL	Cacheable	Write Back	Non-sequential & limited speculation	WBL, WB
1	UC	Uncached	Non-coalescing	Sequential & non-speculative	UC, UCE

a. Coherency here refers to multiprocessor coherence on normal, side-effect free memory.

See "Speculation Attributes" on page 2:67 for a description of physical addressing limited speculation. Bit{63} is discarded when forming the physical address, effectively creating a write-back name space and an uncached name space as shown in Figure 4-21.

Figure 4-21. Addressing Memory Attributes



Software must use the correct name space when using physical addressing; otherwise, I/O devices with side-effects may be accessed speculatively. Physical addressing accesses are ordered only if ordered loads or ordered stores are used. Otherwise, physical addressing memory references are unordered.

4.4.3 Cacheability and Coherency Attribute

A page can be either **cacheable** or **uncacheable**. If a page is marked cacheable, the processor is permitted to allocate a local copy of the corresponding physical memory in all levels of the processor memory/cache hierarchy. Allocation may be modified by the cache control hints of memory reference instructions.

A page which is cached is coherent with memory; i.e., the processor and memory system ensure that there is a consistent view of memory from each processor. Processors support multiprocessor cache coherence based on physical addresses between all processors in the coherence domain (tightly coupled multiprocessors). Coherency is supported in the presence of virtual aliases, although software is recommended to use aliases which are an integer multiple of 1 MB apart to avoid any possible performance degradation.

Processors are not required to maintain coherency between processor local instruction and data caches for Itanium-based code; i.e., locally initiated Itanium stores may not be observed by the local instruction cache. Processors are required to maintain coherency between processor local instruction and data caches for IA-32 code. Instruction caches are also not required to be coherent with multiprocessor Itanium instruction set originated memory references. Instruction caches are required to be coherent with multiprocessor IA-32 instruction set originated memory references. The processor must ensure that transactions from other I/O agents (such as DMA) are physically coherent with the instruction and data cache.

For non-cacheable references the processor provides no coherency mechanisms; the memory system must ensure that a consistent view of memory is seen by each processor. See “Coalescing Attribute” on page 2:66 for a description of coherency for the coalescing memory attribute.

4.4.4 Cache Write Policy Attribute

Write-back cacheable pages need only modify the processor’s copy of the physical memory location; written data need only be passed to the memory system when the processor’s copy is displaced, or a Flush Cache (`fc`) instruction is issued to flush a virtual address. A cache line can only be written back to memory if a store, semaphore (successful or not), the `ld.bias`, a mandatory RSE store, or a `.excl` hinted lfetch instruction targeting that line has executed without a fault. These events enable write-backs. A synchronized `fc` instruction disables subsequent write-backs (after the line has been flushed).

As described in “Invalidating ALAT Entries” on page 2:60, platform visible removal of cache lines from a processor’s caches (e.g., cache line write-backs or platform visible replacements) cause the corresponding ALAT entries to be invalidated.

4.4.5 Coalescing Attribute

For uncacheable pages, the **coalescing** attribute informs the processor that multiple stores to this page may be collected in a coalescing buffer and issued later as a single larger merged transaction. The processor may accumulate stores for an indefinite period of time. Multiple pending loads may also be coalesced into a single larger transaction which is placed in a coalescing buffer. Coalescing is a performance hint for the processor; a processor may or may not implement coalescing.

A processor with multiple coalescing buffers must provide a flush policy that flushes buffers at roughly equal rate even if some buffers are only partially full. The processor may make coalesced buffer flushes visible in any order. Furthermore, individual bytes within a single coalesced buffer may be flushed and made visible in any order.

Stores (including IA-32), which are coalesced, are performed out of order; coalescing may occur in both the space and time domains. For example, a write to bytes 4 and 5 and a write to bytes 6 and 7 may be coalesced into a single write of bytes 4, 5, 6, and 7. In addition, a write of bytes 5 and 6 may be combined with a write of bytes 6 and 7 into a single write of bytes 5, 6, and 7.

Any release operation (regardless of whether it references a page with a coalescing memory attribute), or any fence type instruction, forces write-coalesced data to be flushed and made visible prior to the instruction itself becoming visible. (See Table 4-14 on page 2:69 for a list of release and fence instructions.) Any IA-32 serializing instruction, or access to an uncached memory type, forces write-coalesced data to become flushed and made visible prior to itself becoming visible. Even though IA-32 stores and loads are ordered, the write-coalesced data is not flushed unless the IA-32 stores or loads are to uncached memory types.

The Flush Cache (`fc`) instruction flushes all write-coalesced data whose address is within at least 32 bytes of the 32-byte aligned address specified by the `fc` instruction, forcing the data to become visible. The `fc` instruction may also flush additional write-coalesced data. The Flush Write buffers (`fwb`) instruction is a “hint” to the processor to expedite flushing (visibility) of any pending stores held in the coalescing buffer(s), without regard to address.

No indication is given when the flushing of the stores is completed. An `fwb` instruction does not ensure ordering of coalesced stores, since later stores may be flushed before prior stores. To ensure prior coalesced stores are made visible before later stores, software must issue a release operation between stores.

The processor may at any time flush coalesced stores in any order before explicitly requested to do so by software.

Coalesced pages are not ensured to be coherent with other processors' coalescing buffers or caches, or with the local processor's caches. Loads to coalesced memory pages by a processor see the results of all prior stores by the same processor to the same coalesced memory page. Memory references made by the coalescing buffer (e.g., buffer flushes) have an unordered non-sequential memory ordering attribute. See "Sequentiality Attribute and Ordering" on page 2:69.

Data that has been read or prefetched into a coalescing buffer prior to execution of an Itanium acquire or fence type instruction is invalidated by the acquire or fence instruction. (See Table 4-14 for a list of acquire and fence instructions.)

4.4.6 Speculation Attributes

For present pages (TLB.p=1) which are marked with a **speculative** or a NaTPage memory attribute, the processor may prefetch instructions (including IA-32), perform address generation and perform load accesses (including IA-32) without resolving prior control dependencies, including predicates, branches and interruptions. A page should only be marked speculative if accesses to that page have no side-effects. For example, many memory-mapped I/O devices have side-effects associated with reads and should be marked non-speculative. If a page is marked speculative, a processor can read any location in the page at any time independent of a programmer's intentions or control flow changes. As a result, software is required, at all times, to maintain valid page table attributes for the ppn, ps and ma fields of all present translations whose memory attribute is speculative or NaTPage. High-performance operation is only attainable on speculative pages. The speculative attribute is a hint; a processor may behave non-speculatively.

Prefetches are enabled if a speculative translation exists. Prefetches are asynchronous data and instruction memory accesses that appear logically to initiate and finish between some pair of instructions. This access may not be visible to subsequent flush cache (`fc`) and/or TLB purge instructions. This behavior is implementation-dependent.

The processor will not initiate memory references (16-byte instruction bundle fetches, IA-32 instruction fetches, RSE fills and spills, VHPT references, and data memory accesses) to non-speculative pages until all previous control dependencies (predicates, branches, and exceptions) are resolved; i.e., the memory reference is required by an in-order execution of the program. Additionally, for references to non-speculative pages, the processor:

- May not generate any memory access for a control or data speculative data reference.
- Will generate exactly one memory access for each aligned, non-speculative data reference. (Misaligned data references may cause multiple memory accesses, although these accesses are guaranteed to be non-overlapping – each byte will be accessed exactly once.).
- May generate multiple 16-byte memory accesses (to the same address) for each 16-byte instruction bundle fetch reference.

Limited speculation is used to improve performance when using physical addressing to cachable memory. Because the memory is physically addressed, the processor can have no expectation as to whether or not a given 4k-byte physical page exists until the page has been successfully accessed through a **non-speculative reference**. A non-speculative reference is an instruction or data reference made to the page by an in-order execution of the program. An instruction fetch (or data fetch) which meets this requirement, but which takes an Instruction Debug (or Data Debug) fault or an External interrupt is still a non-speculative reference. Data-speculative references are considered non-speculative for this purpose. Control-speculative references are not allowed for limited-speculation pages and thus do not affect limited-speculation behavior.

Unless a limited-speculation page is **speculatively accessible**, only non-speculative references may be made to it. While a limited-speculation page is speculatively accessible, the processor may access it normally including the use of caching and **hardware-generated speculative references** to improve performance. Hardware-generated speculative references include non-demand instruction prefetches (including IA-32), data references by instructions which have not yet been determined to be required by an in-order execution of the program (due to potential exceptions on prior instructions or mispredictions on prior branches), hardware-generated data prefetch references, and eager RSE memory references. A limited-speculation page can be made speculatively accessible only after the successful completion of a non-speculative reference to the page. Once a limited-speculation page is speculatively accessible, the page can be made **speculatively inaccessible** either explicitly by software (described in Section 4.4.11, “Memory Attribute Transition” on page 2:74) or implicitly for implementation-specific reasons.

To ensure virtual and physical accesses to non-speculative pages are performed in program order and only once per program order occurrence, the rules in Table 4-12 and Table 4-13 are defined. Software should also ensure that RSE spill/fill transactions are not performed to non-speculative memory that may contain I/O devices; otherwise, system behavior is undefined.

Table 4-12. Permitted Speculation

Memory Attribute	Load (ld)[a]	Speculative Load (ld.s)[b]	Advanced Load (ld.a)	Speculative Advanced Load (ld.sa)	Hardware-generated Speculative References[c]
Speculative	Yes	Yes	Yes	Yes	Yes
Non-speculative	Yes	Always Fail	Always Fail	Always Fail	Prohibited
Limited Speculation	Yes	Always Fail	Yes	Always Fail	Limited[d]

a. Includes the faulting form of line prefetch (`lfetch.fault`).

b. Includes the non-faulting form of line prefetch (`lfetch`), which does not cause a cache fill if the memory attribute is non-speculative or limited speculation.

c. Hardware-generated speculative references include non-demand instruction prefetches (including IA-32), hardware-generated data prefetch references, and eager RSE memory references.

d. The processor may only issue hardware-generated speculative references to a 4K-byte physical page while the page is speculatively accessible.

Table 4-13. Register Return Values on Non-faulting Advanced/Speculative Loads

Memory Attribute	Speculative Load (ld.s)		Advanced Load (ld.a)		Speculative Advanced Load (ld.sa)	
	success	failure	success	failure	success	failure
Speculative	Value	Nat[a]	Value	N/a	Value	NaT[a]
Non-speculative	N/A	Nat[b]	N/A	Zero[c]	N/A	NaT[b]
Limited Speculation	N/A	Nat[b]	Value	N/a	N/a	NaT[b]

a. Speculative or speculative advanced loads that cause deferred exceptions result in failed speculation. The processor aborts the reference. If the target of the load is a GR, the processor sets the register’s NaT bit to one. If the target of the load is an FR, the processor sets the target FR to NaTVal. The processor performs all other side-effects (such as post-increment).

b. Speculative or speculative advanced loads to limited or non-speculative memory pages result in failed speculation. The processor aborts the reference. If the target of the load is a GR, the processor sets the register’s NaT bit to 1. If the target of the load is an FR, the processor sets the target FR to NaTVal. The processor performs all other side-effects (such as post-increment).

c. Advanced loads to non-speculative memory pages always fail. The processor aborts the reference, sets the target register to zero, and performs all other side-effects (such as post-increment).

4.4.7 Sequentiality Attribute and Ordering

Memory ordering is defined in Section 4.4.7, "Memory Access Ordering" on page 2:65 in Volume 1. This section defines additional ordering rules for non-cacheable memory, cache synchronization (`sync.i`) and global TLB purge operations (`ptc.g, ptc.ga`).

As described in Section 4.4.7 in Volume 1, read-after-write, write-after-write, and write-after-read dependencies to the same memory location (memory dependency) are performed in program order by the processor[1]. Otherwise, all other memory references may be performed in any order unless the reference is specifically marked as ordered. IA-32 memory references follow a stronger processor consistency memory model. See "IA-32 Memory Ordering" on page 2:238. for IA-32 memory ordering details. Explicit ordering takes the form of a set of Itanium instructions: ordered load and check load (`ld.acq`, `ld.c.clr.acq`), ordered store (`st.rel`), semaphores (`cmpxchg`, `xchg`, `fetchadd`), memory fence (`mf`), synchronization (`sync.i`) and global TLB purge (`ptc.g`, `ptc.ga`). The `sync.i` instruction is used to maintain an ordering relationship between instruction and data caches on local and remote processors. The global TLB purge instructions maintain multiprocessor TLB coherence.

Table 4-14 defines a set of "Orderable Instructions" that follow one of four ordering semantics: **unordered**, **release**, **acquire** or **fence**. The table defines the ordering semantics and the instructions of each category. Only these Itanium instructions can be used to establish multiprocessor ordering relations.

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 an instruction. For memory accesses and semaphores this involves at least reading or writing memory. For `mf.a`, visibility is defined by platform acceptance of previous memory accesses. Visibility of `sync.i` is defined by visibility of previous flush cache (`fc`) operations. For ALAT lookups (`ld.c`, `chk.a`), visibility is determination of ALAT hit or miss. For global TLB purge operations, visibility is defined by removal of an address translation from the TLBs on all processors in the TLB coherence domain. Global TLB purge instructions (`ptc.g` and `ptc.ga`) follow release semantics both on the local and the remote processor.

Table 4-14. Ordering Semantics and Instructions

Ordering Semantics	Description	Orderable Intel® Itanium™ Instructions
Unordered	Unordered instructions may become visible in any order.	`ld`, `ld.s`, `ld.a`, `ld.sa`, `ld.fill`, `ldf`, `ldf.s`, `ldf.sa`, `ldf.fill`, `ldfp`, `ldfp.s`, `ldfp.sa`, `st`, `st.spill`, `stf`, `stf.spill`, `mf.a`, `sync.i`, `ld.c`, `chk.a`
Release	Release instructions guarantee that all previous orderable instructions are made visible prior to being made visible themselves.	`cmpxchg.rel`, `fetchadd.rel`, `st.rel`, `ptc.g`, `ptc.ga`

[1] Although VHPT walks are performed somewhat asynchronously with respect to program execution, each walker VHPT read appears as though it were performed atomically, at some single point in the program order.

Table 4-14. Ordering Semantics and Instructions (Continued)

Ordering Semantics	Description	Orderable Intel® Itanium™ Instructions
Acquire	Acquire instructions guarantee that they are made visible prior to all subsequent orderable instructions.	`cmpxchg.acq`, `fetchadd.acq`, `xchg`, `ld.acq`, `ld.c.clr.acq`
Fence	Fence instructions combine the release and acquire semantics into a bi-directional fence; i.e., they guarantee that all previous orderable instructions are made visible prior to any subsequent orderable instruction being made visible.	`mf`

Itanium memory accesses to **sequential** pages occur in program order with respect to all other sequential pages in the same peripheral domain, but are not necessarily ordered with respect to non-sequential page accesses. A peripheral domain is a platform-specific collection of uncacheable addresses. An I/O device is normally contained in a peripheral domain and all sequential accesses from one processor to that device will be ordered with respect to each other. Sequentiality ensures that uncacheable, non-coalescing memory references from one processor to a peripheral domain reach that domain in program order. Sequentiality does not imply visibility.

Inter-Processor Interrupt Messages (8-byte stores to a Processor Interrupt Block address, through a UC memory attribute) are exceptions to the sequential semantics. IPI’s are not ordered with respect to other IPI’s directed at the same processor. Further, fence operations do not enforce ordering between two IPI’s. See Section 5.8.4.2, “Interrupt and IPI Ordering” on page 2:113.

Table 4-15 defines the ordering between unordered, release, acquire and fence type operations to sequential and non-sequential pages. Table 4-15 defines the minimal ordering requirements; an implementation may enforce more restrictive ordering than required by the architecture. The actual mechanism for enforcing memory access ordering is implementation dependent.

Table 4-15. Ordering Semantics

First Operation		Second Operation						
		Fence	Non-sequential			Sequential[a]		
			Acquire	Release	Unordered	Acquire	Release	Unordered
	Fence	O	O	O	O	O	O	O
Non-sequential	Acquire	O	O	O	O	O	O	O
	Release	O	–	O	–	–	O	–
	Unordered	O	–	O	–	–	O	–
Sequential[a]	Acquire	O	O	O	O	OS	OS	OS
	Release	O	–	O	–	S	OS	S
	Unordered	O	–	O[b]	–[c]	S[d]	OS[e]	S

a. Except for IPI.
b. “O” indicates that the first and second operation become visible in program order.
c. A dash indicates no ordering is implied.
d. “S” indicates that the first and the second operation reach a peripheral domain in program order.
e. “OS” implies that both “O” and “S” ordering relations apply.

Table 4-15 establishes an order between operations on a particular processor. For operations to cacheable write-back memory the order established by these rules is observed by all observers in the coherence domain.

For example, when this sequence is executed on a processor:

```
st [a]
st.rel [b]
```

and a second processor executes this sequence:

```
ld.acq [b]
ld [a]
```

if the second processor observes the store to [b], it will also observe the store to [a].

Unless an ordering constraint from Table 4-15 prevents a memory read[1] from becoming visible, the read may be satisfied with values found in a store buffer (or any logically equivalent structure). These values need not be globally visible even when the operation that created the value was a `st.rel`. This local bypassing behavior may make accesses of different sizes but with overlapping memory references appear to complete non-atomically. To ensure that a memory write is globally observed prior to a memory read, software must place an explicit fence operation between the two operations.

Aligned `st.rel` and semaphore operations[2] from multiple processors to cacheable write-back memory become visible to all observers in a single total order (i.e., in a particular interleaving; if it becomes visible to any observer, then it is visible to all observers), except that for `st.rel` each processor may observe (via `ld` or `ld.acq`) its own update prior to it being observed globally.

The Itanium architecture ensures this single total order only for aligned `st.rel` and semaphore operations to cacheable write-back memory. Other memory operations[3] from multiple processors are not required to become visible in any particular order, unless they are constrained w.r.t. each other by the ordering rules defined in Table 4-15.

Ordering of loads is further constrained by data dependency. That is, if one load reads a value written by an earlier load by the same processor (either directly or transitively, through either registers or memory), then the two loads become visible in program order.

For example, when this sequence is executed on a processor:

```
st [a] = data
st.rel [b] = a
```

and a second processor executes this sequence:

```
ld x = [b]
ld y = [x]
```

if the second processor observes the store to [b], it will also observe the store to [a].

Also for example, when this sequence is executed on a processor:

```
st [a]
st.rel [b] = ‘new’
```

[1] This includes all types of loads (`ld` and `ld.acq`), and RSE and VHPT memory reads. Note, however, that the read operation of semaphores cannot be satisfied with values found in a store buffer.

[2] Both acquire and release semaphore forms

[3] e.g. unordered stores, loads, `ld.acq`, or memory operations to pages with attributes other than write-back cacheable.

and a second processor executes this sequence:

```
        ld x = [b]
        cmp.eq p1 = x, ‘new’
(p1)    ld y = [a]
```

if the second processor observes the store to [b], it will also observe the store to [a].

And for example, when this sequence is executed on a processor:

```
        st [a]
        st.rel [b] = ‘new’
```

and a second processor executes this sequence:

```
        ld x = [b]
        cmp.eq p1 = x, ‘new’
(p1)    br target
        ...
target:
        ld y = [a]
```

if the second processor observes the store to [b], it will also observe the store to [a].

The flush cache (`fc`) instruction follows data dependency ordering. `fc` is ordered with respect to previous and subsequent load, store, or semaphore instructions to the same line, regardless of the specified memory attribute. `fc` is not ordered with respect to memory operations to different lines. `mf` does not ensure visibility of `fc` operations. Instead, the `sync.i` instruction synchronizes `fc` instructions, and the `sync.i` is made visible using an `mf` instruction.

4.4.8 Not a Thing Attribute (NaTPage)

A NaTPage attribute prevents non-speculative references to a page, and ensures that speculative references to the page always defer the Data NaT Page Consumption fault. However, as described in “Speculation Attributes” on page 2:67, the processor may issue memory references to a NaTPage. As a result, all NaTPages must be backed by a valid physical page.

Speculative or speculative advanced loads to pages marked as a NaTPage cause the deferred exception indicator (NaT or NaTVal) to be written to the load target register, and the memory reference is aborted. However, all other effects of the load instruction such as post-increment are performed. Instruction fetches, loads, stores and semaphores (including IA-32), but except for Itanium speculative loads, pages marked as NaTPage raise a NaT Page Consumption fault.

A speculative reference to a page marked as NaTPage may still take lower priority faults, if not explicitly deferred in the DCR. See “Deferral of Speculative Load Faults” on page 2:88.

4.4.9 Effects of Memory Attributes on Memory Reference Instructions

Memory attributes affect the following Itanium instructions.

- `ldfe`, `stfe`: Hardware support for 10-byte memory accesses to a page that is neither a cacheable page with write-back write policy nor a NaTPage is optional. On processor implementations that do not support such accesses, an Unsupported Data Reference Fault is raised when an unsupported reference is attempted.

For extended floating-point loads the fault is delivered only on the normal, advanced, and check load flavors (`ldfe`, `ldfe.a`, `ldfe.c.nc`, `ldfe.c.clr`). Control speculative flavors of the `ldfe` instruction that target pages that are not cacheable with write-back policy always defer the fault. Refer to "Deferral of Speculative Load Faults" on page 2:88 for details.

- `cmpxchg` and `xchg`: These instructions are only supported to cacheable pages with write-back write policy. `cmpxchg` and `xchg` accesses to NaTPages causes a Data NaT Page Consumption fault. `cmpxchg` and `xchg` accesses to pages with other memory attributes cause an Unsupported Data Reference fault.
- `fetchadd`: The `fetchadd` instruction can be executed successfully only if the access is to a cacheable page with write-back write policy or to a UCE page. `fetchadd` accesses to NaTPages cause a Data NaT Page Consumption fault. Accesses to pages with other memory attributes cause an Unsupported Data Reference fault. When accessing a cacheable page with write-back write policy, atomic fetch and add operation is ensured by the processor cache-coherence protocol. For highly contended semaphores, the cache line transactions required to guarantee atomicity can limit performance. In such cases, a centralized "fetch and add" semaphore mechanism may improve performance. If supported by the processor and the platform, the UCE attribute allows the processor to "export" the `fetchadd` operation to the platform as an atomic "fetch and add". Effects of the exported `fetchadd` are platform dependent. If exporting of `fetchadd` instructions is not supported by the processor, a `fetchadd` instruction to a UCE page takes an Unsupported Data Reference fault.
- Flush Cache Instructions – `fc` instructions must always be "broadcast" to other processors, independent of the memory attribute in the local processor. It is legal to use an uncacheable memory attribute for any valid address when used as a flush cache (`fc`) instruction target. This behavior is required to enable transitions from one memory attribute to another and in case different memory attributes are associated with the address in another processor.
- Prefetch instructions – `lfetch` and any implicit prefetches to pages that are not cacheable are suppressed. No transaction is initiated. This allows programs to issue prefetch instructions even if the program is not sure the memory is cacheable.

4.4.10 Effects of Memory Attributes on Advanced/Check Loads

The ALAT behavior of advanced and check loads is dependent on the memory attribute of the page referenced by the load. These behaviors are required; advanced and check load completers are not hints.

All speculative pages have identical behavior with respect to the ALAT. Advanced loads to speculative pages always allocate an ALAT entry for the register, size, and address tuple specified by the advanced load. Speculative advanced loads allocate an ALAT entry if the speculative load is successful (i.e., no deferred exception); if the speculative advanced load results in a deferred exception, any matching ALAT entry is removed and no new ALAT entry is allocated. Check loads with clear completers (`ld.c.clr`, `ld.c.clr.acq`, `ldf.c.clr`) remove a matching ALAT entry on ALAT hit and do not change the state of the ALAT on ALAT miss. Check loads with no-clear completers (`ld.c.nc`, `ldf.c.nc`) allocate an ALAT entry on ALAT miss. On ALAT hit, the ALAT is unchanged if an exact ALAT match is found (register, address, and size); a new ALAT entry with the register, address, and size specified by the no-clear check load may be allocated if a partial ALAT match is found (match on register).

Advanced loads (speculative or non-speculative variants) to non-speculative pages always remove any matching ALAT entry. Check loads to non-speculative pages that miss the ALAT never allocate an ALAT entry, even in the case of a no-clear check load. ALAT hits on check loads to non-speculative pages (which can occur if a previous advanced load referenced that page via a speculative memory attribute) result in undefined behavior; when changing an existing page from speculative to non-speculative (or vice-versa), software should ensure that any ALAT entries corresponding to that page are invalidated.

Limited speculation pages behave like non-speculative pages with respect to speculative advanced loads, and behave like speculative pages with respect to all other advanced and/or check loads.

Table 4-16 describes the ALAT behavior of advanced and check loads for the different speculation memory attributes.

Table 4-16. ALAT Behavior on Non-faulting Advanced/Check Loads

Memory Attribute	ld.sa Response		ld.a Response	ld.c.clr, ld.c.clr.acq, ldf.c.clr Response		ld.c.nc, ldf.c.nc Response	
	no NaT	NaT		ALAT hit	ALAT miss	ALAT hit	ALAT miss
speculative	alloc	remove	alloc	remove	nop	unchanged[a]	alloc
non-speculative	n/a	remove	remove	undefined	nop	undefined	must not alloc
limited speculation	n/a	remove	alloc	remove	nop	unchanged[a]	alloc

[a] May allocate a new ALAT entry if size and/or address are different than the corresponding ld.a or ld.sa whose ALAT entry was matched.

4.4.11 Memory Attribute Transition

If software modifies the memory attributes for a page, it must perform explicit actions to ensure that subsequent reads and writes using the new attribute will be coherent with prior reads and writes that were performed with the old attribute. Processors may have separate buffers for coalescing, uncacheable and cacheable references, and these buffers need not be coherent with each other.

4.4.11.1 Virtual Addressing Memory Attribute Transition

To change a virtually-addressed page from one attribute to another, software must perform the following sequence. (The address of the page whose attribute is being modified is referred to as “X”).

Note: This sequence is ONLY required if the new mapping and the old mapping do not have the same memory attribute.

On the processor initiating the transition, perform the following steps 1-3:

1. `PTE[X].p = 0 // Mark page as not present`

   This prevents any processors from reading the old mapping (with the old attribute) from the VHPT after this point.

2.
   ```
   ptc.ga [X] ;; // Global shootdown and ALAT invalidate
                 // for the entire page
   ```

   This removes the mapping from all processor TC's in the coherence domain, and it forces all processors to flush any pending WC or UC stores from write buffers.

3.
   ```
   mf ;;     // Ensure visibility of ptc.ga to local data stream
   srlz.i ;; // Ensure visibility of ptc.ga to local instruction stream
   ```

   After step 3, no processor in the coherence domain will initiate new memory references or prefetches to the old translation. Note, however, that memory references or prefetches initiated to the old translation prior to step 2 may still be in progress after step 3. These outstanding memory references and prefetches may return instructions or data which may be placed in the processor cache hierarchy; this behavior is implementation-specific.

   If the new memory attribute is an uncacheable attribute, and if the old attribute was cacheable (or if it is not known at this point in the code sequence what the old attribute was), then software must drain any current prefetches and ensure that any cached data from the page is removed from caches. To do this, software must perform steps 4-10. If the new memory attribute is cacheable, then software may skip steps 4-10, and go straight to step 11.

4. Call PAL_PREFETCH_VISIBILITY

   Call PAL_PREFETCH_VISIBILITY with the input argument *trans_type* equal to zero to indicate that the transition is for virtual memory attributes. The return argument from this procedure informs the caller if this procedure call is needed on remote processors or not. If this procedure call is not needed on remote processors, then software may skip the IPI in step 5 and go straight to step 6 below.

5. Using the IPI mechanism defined in "Inter-processor Interrupt Messages" on page 2:111 to reach all processors in the coherence domain, perform step 4 above on all processors in the coherence domain, and wait for all PAL_PREFETCH_VISIBILITY calls to complete on all processors in the coherence domain before continuing.

   After steps 4 and 5, no more new instruction or data prefetches will be made to page "X" by any processor in the coherence domain. However, processor caches in the coherence domain may still contain "stale" data or instructions from prior prefetch or memory references to page "X".

6. Insert a temporary UC translation for page "X".

7.
   ```
   fc [X] // flush all processor caches in the coherence domain
   fc [X+32]
   fc [X+64]
   ... // ... for all of page "X" (page size = ps)
   fc [X+ps-32] ;;

   // Ensure cache flushes are also seen by processors' instruction fetch
   sync.i ;;
   ```

   After step 7, all flush cache instructions initiated in step 7 are visible to all processors in the coherence domain, i.e., no processor in the coherence domain will respond with a cache hit on a memory reference to an address belonging to page "X".

8. Purge the temporary UC translation from the TLB

9. Call PAL_MC_DRAIN

10. Using the IPI mechanism defined in "Inter-processor Interrupt Messages" on page 2:111 to reach all processors in the coherence domain, perform step 9 above on all processors in the coherence domain, and wait for all PAL_MC_DRAIN calls to complete on all processors in the coherence domain before continuing.

    This further guarantees that any cache lines containing addresses belonging to page [X] have been evicted from all caches in the coherence domain and forced onto the bus. Note that this operation does not ensure that the cache lines have been written back to memory.

11. Insert the new mapping with the new memory attribute

4.4.11.2 Physical Addressing Attribute Transition - Disabling Prefetch/ Speculation and Removing Cacheability

When a non-speculative reference is made to a physical address with the WBL attribute, the 4K page containing that address becomes speculatively accessible. This allows the processor that made the non-speculative reference to subsequently make speculative references to this page. (See the description of limited speculation in Section 4.4.6, "Speculation Attributes" on page 2:67.)

If the same physical memory is then to be accessed with the UC attribute, software must first make all such addresses speculatively inaccessible and flush any cached copies from the cache. Otherwise, an uncacheable reference may hit in cache, causing a Machine Check abort.

Also, if physical memory is to be removed from the system, or if physical memory is to be re-configured in such a way that some physical address X, which used to correspond to some portion of memory will now corresponds to nothing in the system, software take these same actions. Otherwise, the processor may initiate a speculative prefetch after the memory has been removed or re-configured, causing a Machine Check abort.

On the processor initiating the transition, perform the following steps:

1. Call PAL_PREFETCH_VISIBILITY

   Call PAL_PREFETCH_VISIBILITY with the input argument *trans_type* equal to one to indicate that the transition is for physical memory attributes. This PAL call terminates the processor's rights to make speculative references to any limited speculation pages (i.e., it makes all WBL pages speculatively inaccessible - see the discussion on limited speculation in Section 4.4.6.)

   The return argument from this procedure informs the caller if this procedure call is needed on remote processors or not. If this procedure call is not needed on remote processors, then software may skip the IPI in step 2 and go straight to step 3 below.

2. Using the IPI mechanism defined in "Inter-processor Interrupt Messages" on page 2:111 to reach all processors in the coherence domain, perform step 1 above on all processors in the coherence domain, and wait for all PAL_PREFETCH_VISIBILITY calls to complete on all processors in the coherence domain before continuing.

On the processor initiating the disabling process, continue the sequence:

3.
   ```
   fc [X] // flush all processor caches in the coherence domain
   fc [X+32]
   fc [X+64]
   ... // ... for all of page “X” (page size = ps)
   fc [X+ps-32] ;;

   // Ensure cache flushes are also seen by processors' instruction fetch
   sync.i ;;
   ```

   After step 3, all flush cache instructions initiated in step 3 are visible to all processors in the coherence domain, i.e., no processor in the coherence domain will respond with a cache line hit on a memory reference to an address belonging to page “X”.

4. Call PAL_MC_DRAIN.

5. Using the IPI mechanism defined in “Inter-processor Interrupt Messages” on page 2:111 to reach all processors in the coherence domain, perform step 4 above on all processors in the coherence domain, and wait for all PAL_MC_DRAIN calls to complete on all processors in the coherence domain before continuing.

   This further guarantees that any cache lines containing addresses belonging to page [X] have been evicted from all caches in the coherence domain and forced onto the bus. Note that this operation does not ensure that the cache lines have been written back to memory.

   This sequence ensures that speculation and prefetch are disabled for all WBL pages, that all outstanding prefetches have completed, and that the caches have been flushed. It may also be necessary to take additional platform-dependent steps to ensure that all cache write-back transactions have completed to memory before removing or re-configuring physical memory.

4.5 Memory Datum Alignment and Atomicity

All Itanium instruction fetches, aligned load, store and semaphore operations (including IA-32) are atomic, except for floating-point extended memory references (`ldfe`, `stfe`, and IA-32 10-byte memory references) to non-write-back cacheable memory. In some processor models, aligned 10-byte Itanium floating-point extended memory references to non-write-back cacheable memory may raise an Unsupported Data Reference fault. See “Effects of Memory Attributes on Memory Reference Instructions” on page 2:72 for details. Loads are allowed to be satisfied with values obtained from a store buffer (or any logically equivalent structure) where architectural ordering permits, and values loaded may appear to be non-atomic. For details, refer to “Sequentiality Attribute and Ordering” on page 2:69.

Load pair instructions are performed atomically under the following conditions: a 16-byte aligned load integer/double pair is performed as an atomic 16-byte memory reference. An 8-byte aligned load single pair is performed as an atomic 8-byte memory reference.

Aligned Itanium data memory references never raise an Unaligned Data Reference fault. Minimally, each Itanium instruction and its corresponding template are fetched together atomically. Itanium unordered loads can use the store buffer for data values. See “Sequentiality Attribute and Ordering” on page 2:69 for details.

When PSR.ac is 1, any Itanium data memory reference that is not aligned on a boundary the size of the operand results in an Unaligned Data Reference fault; e.g., 1, 2, 4, 8, 10, and 16-byte datums should be aligned on 1, 2, 4, 8, 16, and 16-byte boundaries respectively to avoid generation of an Unaligned Data Reference fault. When PSR.ac is 1, any IA-32 data memory reference that is not aligned on a boundary the size of the operand results in an IA-32_Exception(AlignmentCheck) fault.

Note: 10-byte and floating-point load double pair datum alignment is 16-bytes. The alignment of long format 32-byte VHPT references is always 32-bytes.

Unaligned Itanium semaphore references (`cmpxchg,` `xchg`, `fetchadd`) result in an Unaligned Data Reference fault regardless of the state of PSR.ac.

When PSR.ac is 0, Itanium data memory references that are not aligned may or may not result in an Unaligned Data Reference fault based on the implementation. The level of unaligned memory support is implementation specific. However, all implementations will raise an Unaligned Data Reference fault if the datum referenced by an Itanium instruction spans a 4K aligned boundary, and many implementations will raise an Unaligned Data Reference fault if the datum spans a cache line. Implementations may also raise an Unaligned Data Reference fault for any other unaligned Itanium memory reference. Software is strongly encouraged to align data values to avoid possible performance degradation for both IA-32 and Itanium-based code. When PSR.ac is 0 and IA-32 alignment checks are also disabled, no fault is raised regardless of alignment for IA-32 data memory references.

Unaligned advanced loads are supported, though a particular implementation may choose not to allocate an ALAT entry for an unaligned advanced load. Additionally, the ALAT may “pessimistically” allocate an entry for an unaligned load by allocating a larger entry than the natural size of the datum being loaded, as long as the larger entry completely covers the unaligned address range (e.g. a `ld4.a` to address 0x3 may allocate an 8-byte entry starting at address 0x0). Stores (unaligned or otherwise) may also pessimistically invalidate unaligned ALAT entries.

Interruptions 5

Interruptions are events that occur during instruction processing, causing the flow control to be passed to an interruption handling routine. In the process, certain processor state is saved automatically by the processor. Upon completion of interruption processing, a return from interruption (`rfi`) is executed which restores the saved processor state. Execution then proceeds with the interrupted instruction.

From the viewpoint of response to interruptions, the processor behaves as if it were not pipelined. That is, it behaves as if a single Itanium instruction (along with its template) is fetched and then executed; or as if a single IA-32 instruction is fetched and then executed. Any interruption conditions raised by the execution of an instruction are handled at execution time, in sequential instruction order. If there are no interruptions, the next Itanium instruction and its template, or the next IA-32 instruction, are fetched.

This chapter describes both the IA-32 and Itanium interruption mechanisms as well as the interactions between them. The descriptions of the Itanium interruption vectors and IA-32 exceptions, interruptions, and intercepts are in Chapter 8.

5.1 Interruption Definitions

Depending on how an interruption is serviced, interruptions are divided into: IVA-based interruptions and PAL-based interruptions.

- **IVA-based interruptions** are serviced by the operating system. IVA-based interruptions are vectored to the Interruption Vector Table (IVT) pointed to by CR2, the IVA control register (See "IVA-based Interruption Vectors" on page 2:96).
- **PAL-based interruptions** are serviced by PAL firmware, system firmware, and possibly the operating system. PAL-based interruptions are vectored through a set of hardware entry points directly into PAL firmware (See Chapter 11, "Processor Abstraction Layer").

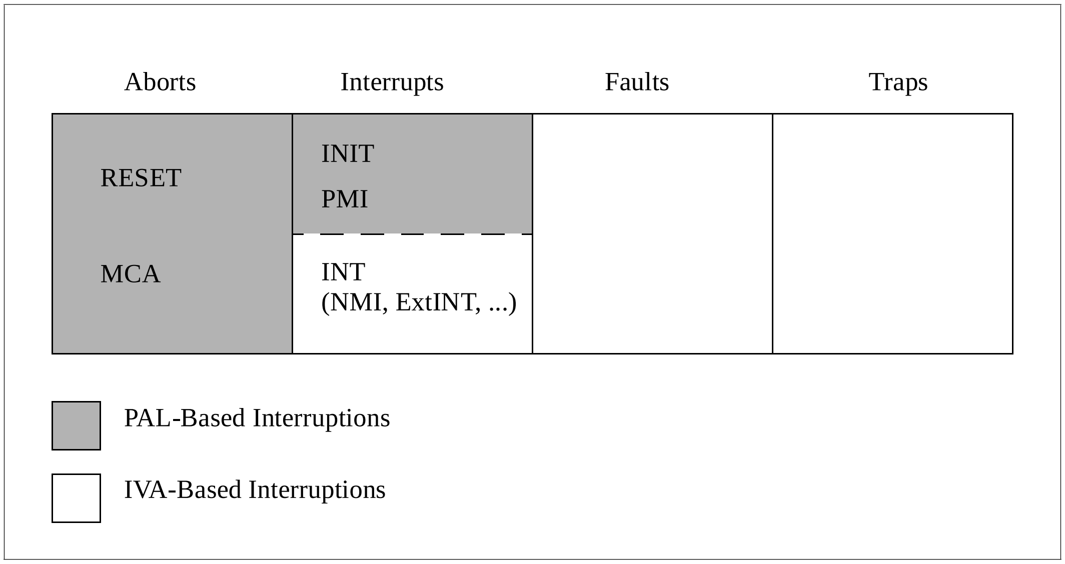
Interruptions are divided into four types: Aborts, Interrupts, Faults, and Traps.

- **Aborts**
  A processor has detected a Machine Check (internal malfunction), or a processor reset. Aborts can be either synchronous or asynchronous with respect to the instruction stream. The abort may cause the processor to suspend the instruction stream at an unpredictable location with partially updated register or memory state. Aborts are PAL-based interruptions.
  - **Machine Checks (MCA)**
    A processor has detected a hardware error which requires immediate action. Based on the type and severity of the error the processor may be able to recover from the error and continue execution. The PALE_CHECK entry point is entered to attempt to correct the error.
  - **Processor Reset (RESET)**
    A processor has been powered-on or a reset request has been sent to it. The PALE_RESET entry point is entered to perform processor and system self-test and initialization.

- **Interrupts**
  An external or independent entity (e.g., an I/O device, a timer event, or another processor) requires attention. Interrupts are asynchronous with respect to the instruction stream. All previous instructions (including IA-32) appear to have completed. The current and subsequent instructions have no effect on machine state. Interrupts are divided into Initialization interrupts, Platform Management interrupts, and External interrupts. Initialization and Platform Management interrupts are PAL-based interruptions; external interrupts are IVA-based interruptions.
  - **Initialization Interrupts (INIT)**
    A processor has received an initialization request. The PALE_INIT entry point is entered and the processor is placed in a known state.
  - **Platform Management Interrupts (PMI)**
    A platform management request to perform functions such as platform error handling, memory scrubbing, or power management has been received by a processor. The PALE_PMI entry point is entered to service the request. Program execution may be resumed at the point of interruption. PMIs are distinguished by unique vector numbers. Vectors 0 through 3 are available for platform firmware use and are present on every processor model. Vectors 4 and above are reserved for processor firmware use. The size of the vector space is model specific.
  - **External Interrupts (INT)**
    A processor has received a request to perform a service on behalf of the operating system. Typically these requests come from I/O devices, although the requests could come from any processor in the system including itself. The External Interrupt vector is entered to handle the request. External Interrupts are distinguished by unique vector numbers in the range 0, 2, and 16 through 255. These vector numbers are used to prioritize external interrupts. Two special cases of External Interrupts are Non-Maskable Interrupts and External Controller Interrupts.
    - **Non-Maskable Interrupts (NMI)**
      Non-Maskable Interrupts are used to request critical operating system services. NMIs are assigned external interrupt vector number 2.
    - **External Controller Interrupts (ExtINT)**
      External Controller Interrupts are used to service Intel 8259A-compatible external interrupt controllers. ExtINTs are assigned locally within the processor to external interrupt vector number 0.
- **Faults**
  The current Itanium or IA-32 instruction which requests an action which cannot or should not be carried out, or system intervention is required before the instruction is executed. Faults are synchronous with respect to the instruction stream. The processor completes state changes that have occurred in instructions prior to the faulting instruction. The faulting and subsequent instructions have no effect on machine state. Faults are IVA-based interruptions.
- **Traps**
  The IA-32 or Itanium instruction just executed requires system intervention. Traps are synchronous with respect to the instruction stream. The trapping instruction and all previous instructions are completed. Subsequent instructions have no effect on machine state. Traps are IVA-based interruptions.

Figure 5-1 summarizes the above classification.

Figure 5-1. Interruption Classification



Unless otherwise indicated, the term “interruptions” in the rest of this chapter refers to IVA-based interruptions. PAL-based interruptions are described in detail in Chapter 11.

5.2 Interruption Programming Model

When an interruption event occurs, hardware saves the minimum processor state required to enable software to resolve the event and continue. The state saved by hardware is held in a set of interruption resources, and together with the interruption vector gives software enough information to either resolve the cause of the interruption, or surface the event to a higher level of the operating system. Software has complete control over the structure of the information communicated, and the conventions between the low-level handlers and the high-level code. Such a scheme allows software rather than hardware to dictate how to best optimize performance for each of the interruptions in its environment. The same basic mechanisms are used in all interruptions to support efficient low-level fault handlers for events such as a TLB fault, speculation fault, or a key miss fault.

On an interruption, the state of the processor is saved to allow a software handler to resolve the interruption with minimal bookkeeping or overhead. The banked general registers (see “Efficient Interruption Handling” on page 2:86) provide an immediate set of scratch registers to begin work. For low-level handlers (e.g., TLB miss) software need not open up register space by spilling registers to either memory or control registers.

Upon an interruption, asynchronous events such as external interrupt delivery are disabled automatically by hardware to allow software to either handle the interruption immediately or to safely unload the interruption resources and save them to memory. Software will either deal with the cause of the interruption and `rfi` back to the point of the interruption, or it will establish a new environment and spill processor state to memory to prepare for a call to higher-level code. Once enough state has been saved (such as the IIP, IPSR, and the interruption resources needed to resolve the fault) the low-level code can re-enable interruptions by restoring the PSR.ic bit and then the PSR.i bit. (See “Re-enabling External Interrupt Delivery” on page 2:103.) Since there is only one set of interruption resources, software must save any interruption resource state the operating system may require prior to unmasking interrupts or performing an operation that may raise a synchronous interruption (such as a memory reference that may cause a TLB miss).

The PSR.ic (interruption state collection) bit supports an efficient nested interruption model. Under normal circumstances the PSR.ic bit is enabled. When an interruption event occurs, the various interruption resources are overwritten with information pertaining to the current event. Prior to saving the current set of interruption resources, it is often advantageous in a miss handler to perform a virtual reference to an area which may not have a translation. To prevent the current set of resources from being overwritten on a nested fault, the PSR.ic bit is cleared on any interruption. This will suppress the writing of critical interruption resources if another interruption occurs while the PSR.ic bit is cleared. If a data TLB miss occurs while the PSR.ic bit is zero, then hardware will vector to the Data Nested TLB fault handler.

For a complete description of interruption resources (IFA, IIP, IPSR, ISR, IIM, IIPA, ITIR, IHA, IFS) see “Control Registers” on page 2:24.

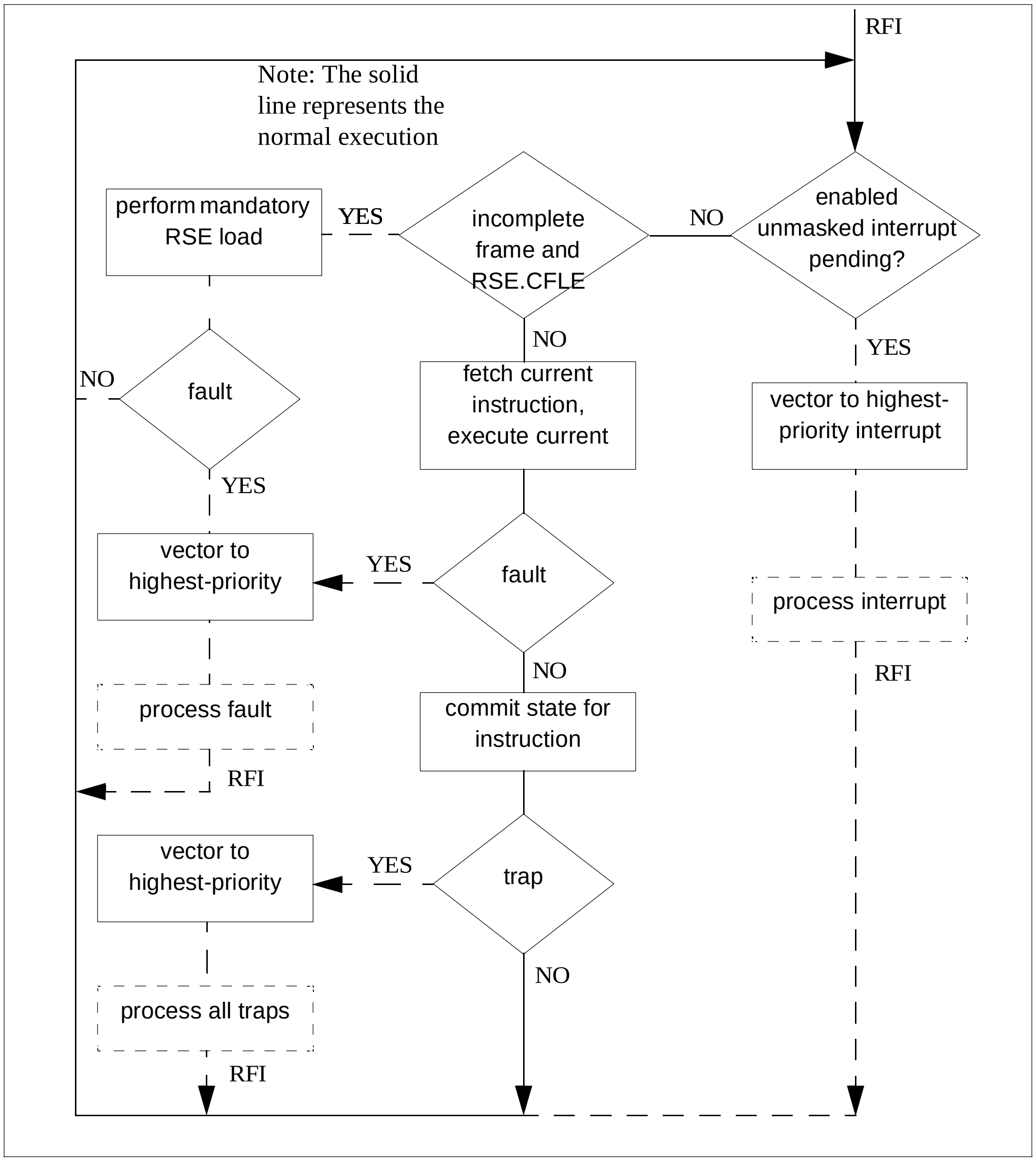
5.3 Interruption Handling during Instruction Execution

Execution of Itanium instructions involves calculating the address of the current bundle from the region registers and the IP and then fetching, decoding, and executing instructions in that bundle. Execution of IA-32 instructions involves calculating the 64-bit linear address of the current instruction from the EIP, code segment descriptors, and region registers and then fetching, decoding, and executing the IA-32 instruction. (See Section 3.4).

The execution process involves performing the events listed below. The values of the PSR bits are the values that exist before the instruction is executed (except for the case of instructions that are immediately preceded by a mandatory RSE load which clears the PSR.da and PSR.dd bits). Changes to the PSR bits only affect subsequent instructions, and are only guaranteed to be visible by the insertion of the appropriate serializing operation. See “Serialization” on page 2:13 Execution flow is shown in Figure 5-2.

1. Resets are always enabled, and may occur anytime during instruction execution.
2. If the PSR.mc bit is 0 then machine check aborts may occur.
3. The processor checks for enabled pending INITs and PMIs, and for enabled unmasked pending external interrupts.
4. For Itanium-based code, the processor checks for a valid register stack frame.
   - If incomplete and RSE Current Frame Load Enable (RSE.CFLE) is set, then perform a mandatory RSE load and start again at step one. The mandatory load operation may fault. A non-faulting mandatory RSE load will clear PSR.da and PSR.dd.
   - If valid, then clear RSE.CFLE.

Figure 5-2. Interruption Processing


RFI
Note: The solid line represents the normal execution
perform mandatory RSE load
YES
incomplete frame and RSE.CFLE
NO
enabled unmasked interrupt pending?
NO
YES
NO
fault
fetch current instruction, execute current
vector to highest-priority interrupt
YES
vector to highest-priority
YES
fault
process interrupt
NO
RFI
process fault
commit state for instruction
RFI
vector to highest-priority
YES
trap
NO
process all traps
RFI

5. For IA-32 code, IA-32 instruction addresses are checked for possible instruction breakpoint faults. The IA-32 effective instruction address (EIP) is converted into a 64-bit virtual linear address IP and IA-32 defined code segmentation and code fetch faults are checked and may result in a fault.

6. When PSR.is is 0, the bundle is fetched using the IP. When PSR.is is 1, an IA-32 instruction is fetched using IP.
   - If the PSR.it bit is 1, virtual address translation of the instruction address is performed. Address translation may result in a fault.
   - If the PSR.pk bit is 1, access key checking is enabled and may result in a fault.
   - For Itanium instructions the IBR registers are checked for possible instruction breakpoint faults.

- The fetched instruction is decoded and executed.
- For IA-32 code, the fetched IA-32 instruction is checked to see if the opcode is an illegal opcode, results in an instruction intercept or the opcode bytes are longer than 15 bytes resulting in an fault.
- If a fault occurs during execution, the processor completes all effects of the instructions prior to the faulting instruction, and does not commit the effect of the faulting instruction and all subsequent instructions. It then takes the interruption for the fault. IIP is loaded with the IP of the bundle or IA-32 instruction which contains the instruction that caused the fault.
- The PSR.dd, PSR.id, PSR.ia, PSR.da, and PSR.ed bits are set to 0 after an Itanium instruction is successfully executed without raising a fault. The PSR.da and PSR.dd bits are also set to 0 after the execution of each mandatory RSE memory reference that does not raise a fault. PSR.da, PSR.ia, PSR.dd, and PSR.ed bits are cleared before the first IA-32 instruction starts execution after a `br.ia` or `rfi` instruction. EFLAG.rf and PSR.id bits are set to 0 after an IA-32 instruction is successfully executed.
- If an `rfi` instruction is in the current bundle, then on the execution of `rfi,` the value from the IIP is copied into the IP, the value from IPSR is copied into the PSR, and the RSE.CFLE is set. On an `rfi` if IFS.v is set, then IFS.pfm is copied into CFM and the register stack BOF is decremented by CFM.sof. The following Itanium or IA-32 instruction is executed based on the new IP and PSR values.

7. Traps are handled after execution is complete.
   - If the instruction just completed set the instruction pointer (IP) to an unimplemented address, an Unimplemented Instruction Address trap is taken.
   - If the instruction just completed was an Itanium floating-point instruction which raised a trap, a Floating-point trap is taken.
   - For IA-32 instructions, if Data Breakpoint traps are enabled and one or more data breakpoint registers matched during execution of the instruction, a Data Breakpoint trap is taken.
   - If the PSR.lp bit is 1, and an Itanium branch lowers the privilege level, then a Lower-Privilege Transfer trap is taken.
   - If the PSR.tb bit is 1 and a branch (including IA-32) occurred during execution, then a Taken Branch trap occurs.
   - If no other trap was taken and the PSR.ss bit is 1, then a Single Step trap occurs.
   - If more than one trap is triggered (such as Unimplemented Instruction Address trap, Lower-Privilege Transfer trap, and Single Step trap) the highest priority trap is taken. The ISR.code contains a bit vector with one bit set for each trap triggered.

A sequential execution model is presented in the preceding description. Implementations are free to use a variety of performance techniques such as pipelined, speculative, or out-of-order execution provided that, to the programmer, the illusion that instructions are executed sequentially is preserved.

5.4 PAL-based Interruption Handling

The actions a processor takes and the state that it modifies immediately after a PAL-based interruption is received are implementation dependent, unless otherwise indicated. For example, an implementation may choose to support a set of shadow resources on a machine check abort which enables recovery even when PSR.ic is 0. It may also choose to use the same resources as an IVA-based interruption event, and hence only support recovery if PSR.ic is 1 at the time of the abort. On the other hand, a processor must set PSR.it to 0 and PSR.mc to 1 after a machine check abort. See Chapter 11, "Processor Abstraction Layer" for details on PAL-based interruptions. See model specification documentation for the processor state and actions for all PAL-based firmware interruptions.

5.5 IVA-based Interruption Handling

IVA-based interruption handling is implemented as a fast context switch. On IVA-based interruptions, instruction and data translation is left unchanged, the endian mode is set to the system default, and delivery of most PSR-controlled interruptions is disabled (including delivery of asynchronous events such as external interrupts). The processor is responsible for saving only a minimal amount of state in the interruption resource registers prior to vectoring to the Itanium-based software handler.

When an interruption occurs, the processor takes the following actions:

1. If PSR.ic is 0:
   - IPSR, IIP, IIPA, and IFS.v are unchanged.
   - Interruption-specific resources IFA, IIM, and IHA are unchanged.

   If PSR.ic is 1:

   - PSR is saved in IPSR. If PSR is in-flight, IPSR will get the most recent in-flight value of PSR (i.e., PSR is serialized by the processor before it is written into IPSR). For Itanium traps, the value written to IPSR.ri is the next instruction slot that would have been executed if there had been no trap. For all other interruptions, the value written to IPSR.ri is the instruction slot on which the interruption occurred (1 for interruptions on the L+X instruction of an MLX). For interruptions in the IA-32 instruction set, IPSR.ri is set to 0.
   - IP is written into IIP. For faults and external interrupts, the saved IP is the IP at which the interruption occurred. For traps, the saved IP is the value after the execution of the IA-32 or Itanium instruction which caused the trap. For branch-related traps, IIP is written with the target of the branch; for all other traps, IIP is written with the address of the bundle or IA-32 instruction containing the next sequential instruction.
   - IIPA receives the IP of the last successfully executed Itanium instruction. For IA-32 instructions, IIPA receives the IP of the faulting or trapping IA-32 instruction.
   - The interruption resources IFA, IIM, IHA, and ITIR are written with information specific to the particular fault, trap, or interruption taken. These registers serve as parameters to each of the interruption vectors. The IFS valid bit (IFS.v) is cleared. All other bits in the IFS are undefined.

If PSR.ic is in-flight:

- Interruption state may or may not be collected in IIP, IPSR, IIPA, ITIR, IFA, IIM, and IHA.
- The value of IFS (including IFS.v) is undefined.

2. ISR bits are overwritten on all interruptions except for a Data Nested TLB fault. The instruction slot which caused the interruption is saved in ISR.ei (2 for traps, 1 for other interruptions, on the L+X instruction of an MLX). For IA-32 code, ISR.ei is set to 0. If PSR.ic is 0 or in-flight when the interruption occurs, ISR.ni is set to 1. Otherwise, ISR.ni is set to 0. ISR.ni is always 0 for interruptions taken in IA-32 code.
3. The defined bits in the PSR are set to zero except as follows:
    - PSR.up, PSR.mfl, PSR.mfh, PSR.pk, PSR.dt, PSR.rt, PSR.mc, and PSR.it are unchanged for all interruptions.
    - PSR.be is set to the value of the default endian bit (DCR.be). If DCR.be is in-flight at the time of interruption, PSR.be may receive either the old value of DCR.be or the in-flight value.
    - PSR.pp is set to the value of the default privileged performance monitor bit (DCR.pp). If DCR.pp is in-flight at the time of interruption, PSR.pp may receive either the old value of DCR.pp or the in-flight value.

    Since PSR.cpl is set to zero, the processor will execute at the most privileged level.
4. RSE.CFLE is set to zero.
5. IP gets the appropriate IVA vector for the interruption. If IVA is in-flight at the time of interruption, IP receives either the vector specified by the old IVA value or the vector specified by the in-flight value.
6. The processor performs an instruction serialization and execution of Itanium instructions begins at the IP obtained in step 5 above. The instruction serialization event ensures that all previous control register changes and side effects due to such changes are visible to the first instruction of the interruption handler.

5.5.1 Efficient Interruption Handling

A set of 16 banked registers are provided by the processor to assist in the efficient processing of low-level Itanium interruptions and instruction emulation. These registers allow a low-level routine to have immediate access to a small set of static registers without having to save and restore their contents to memory at the start and end of each handler. The extra bank of registers exists in the same name space as the normal registers, overlapping GR16 to GR31. Which set of physical registers are accessed through GR16 to GR31 is determined by the PSR.bn bit. On an interruption this bit is forced to zero allowing access to the alternate set of 16 registers which can be used as scratch space or to hold predetermined values. Software can return to the original set of 16 GRs by setting the PSR.bn bit to one with `bsw` instruction. The rfi instruction may also restore the PSR.bn bit to the value at the time of the interruption which is held in the IPSR. Eight additional registers (KR0-KR7) can be used to hold latency critical information for a handler. These application registers (KR0-KR7) can be read but not written by non-privileged code.

When the processor handles an interruption event the current stack frame remains unchanged and the IFS valid bit is cleared. The remaining contents of IFS are undefined. While the interruption handler is running, the register stack engine (RSE) may spill/fill registers to/from the backing store if eager RSE stores/loads are enabled. The RSE will not load or store registers in the current frame (except as required on a `br.ret` or `rfi` in order to load the contents of the frame before continuing execution). For most low-level interruptions the current frame will not be modified. High-performance interruption handlers will not need to perform any register stack manipulation. For example, a TLB miss handler does not need access to any registers in the interrupted frame. An `rfi` instruction after an interruption and before a `cover` operation will also leave the frame marker unchanged (desired behavior for a low-level interruption handler). When an interruption handler falls off the fast path it is required to issue a `cover` instruction so that the interrupted frame can become part of backing store. See “Switch from Interrupted Context” on page 2:131.

It may be desirable to emulate a faulting instruction in the interruption handler and `rfi` back to the next sequential instruction rather than resuming at the faulting instruction. Some Itanium instructions can be emulated without having to read the bundle from memory, through knowledge of the vector, software convention, and information from the ISR (e.g., emulation of `tpa`). However, most Itanium instructions will require reading the bundle from memory and decoding the operation (e.g., an unaligned load). To correctly emulate an unaligned load, the bundle is read from memory using the value in the IIP which contains the bundle address. The instruction within the bundle that caused the interruption is determined by the ISR.ei field. Once the operation is decoded and emulation completes, the effect of the faulting instruction must be nullified when control is returned to the point of the fault.

An Itanium instruction is skipped by adjusting PSR.ri and possibly IIP prior to performing the `rfi` to the interrupted bundle. This is done by incrementing IPSR.ri by the number of slots this instruction occupies (usually 1). If the resulting IPSR.ri is 3, then reset IPSR.ri to 0 and advance IIP by 1 bundle (16 bytes). Emulating X-unit instructions requires setting IPSR.ri to 0 and setting IIP to the next bundle (X-unit instructions take up two instruction slots). IPSR, IIP, and IFS.pfm (if valid) will be restored on an `rfi` to the PSR, IP, and CFM registers.

5.5.2 Non-access Instructions and Interruptions

The non-access Itanium instructions are: `fc`, `lfetch`, `probe`, `tpa`, and `tak`. These instructions reference the TLB but do not directly read or write memory. They are distinguished from normal load/store instructions since an operating system may wish to handle an interruption raised by a non-access instruction differently.

All non-access Itanium instructions can cause interruptions (`tpa`, `fc`, `probe`, `tak` only for non-TLB related reasons). ISR.code will be set to indicate which non-access instruction caused the interruption. See Table 5-1 for ISR field settings for non-access instructions.

Table 5-1. ISR Settings for Non-access Instructions

Instruction	ISR Fields			
	code{3:0}	na	r	w
`tpa`	0	1	0	0
`fc`	1	1	1	0
`probe`	2	1	0 or 1[a]	0 or 1[a]
`tak`	3	1	0	0
`lfetch`, `lfetch.fault`	4	1	1	0
`probe.fault`	5	1	0 or 1[a]	0 or 1[a]

[a] Sets r or w or both to 1 depending on the `probe` form.

5.5.3 Single Stepping

The processor can single step through a series of instructions by enabling the single step PSR.ss bit. This is accomplished by setting the IPSR.ss bit and performing an `rfi` back to the instruction to be single stepped over. When single stepping, the processor will execute one IA-32 instruction or one Itanium instruction pointed to by the IPSR.ri field.

After single stepping Itanium instruction slot 2 (IPSR.ri = 2) or when the template is MLX and single stepping instruction slot 1 (IPSR.ri = 1), the IIP will point to the next bundle, and IPSR.ri will point to slot 0.

5.5.4 Single Instruction Fault Suppression

Four bits, PSR.id, PSR.da, PSR.ia, and PSR.dd are defined to suppress faults for one Itanium instruction or one mandatory RSE memory operation. The PSR.id bit is used to suppress the instruction debug fault for one IA-32 or Itanium instruction. This bit will be cleared in the PSR after the first successfully executed instruction. The PSR.ia bit is used to suppress the Instruction Access Bit fault for one Itanium instruction. This bit will be cleared in the PSR after the first successfully executed instruction. The PSR.da and PSR.dd bits are used to suppress Dirty-Bit, Data Access-Bit and Data Debug faults for one Itanium instruction, or for one mandatory RSE memory reference. The PSR.da and PSR.dd bits will be cleared in the PSR after the first instruction is executed without raising a fault, or after the first mandatory RSE memory reference that does not raise a fault completes. PSR.da, PSR.ia and PSR.dd are cleared before the first IA-32 instruction starts execution after a `br.ia` or `rfi` instruction. Software may set the PSR.id, PSR.da, PSR.ia and PSR.dd bits in the IPSR prior to an `rfi`. The `rfi` will restore the PSR from the IPSR. By using these disable bits, software may step over a debug or dirty/access event and continue execution.

5.5.5 Deferral of Speculative Load Faults

Speculative and speculative advanced loads can defer fault handling by suppressing the speculative memory reference, and by setting the deferred exception indicator (NaT bit or NaTVal) of the load target register. Other effects of the instruction (such as post increment) are performed. Additionally, software can suppress the memory reference of speculative and speculative advanced loads independent of any exception.

Deferral is the process of generating a deferred exception indicator and not performing the exception processing at the time of its detection (and potentially never at all). Once a deferred exception indicator is generated, it will propagate through all uses until the speculation is checked by using either a `chk.s` instruction, a `chk.a` instruction (for speculative advanced loads), or a non-speculative use. This causes the appropriate action to be invoked to deal with the exception.

Three different programming models are supported: **no-recovery**, **recovery** and **always-defer**. In the no-recovery model, only fatal exceptional conditions are deferred - these are conditions which cannot be resolved without either involving the program’s exception-handling code or terminating the program. In the recovery model, performance may be increased by deferring additional exceptional conditions. The recovery model is used only if the program provides additional “recovery” code to re-execute failed speculative computations. When a speculative load is executed with PSR.ic equal to 1, and ITLB.ed equal to 0, the no-recovery model is in effect. When PSR.ic is 1 and ITLB.ed is 1, the recovery model is in effect. The **always-defer** model is supported for use in system code which has PSR.ic equal to 0. In this model, all exceptional conditions which can be deferred are deferred. This permits speculation in environments where faulting would be unrecoverable.

Table 5-2. Programming Models

PSR.ic	PSR.it	ITLB.ed	Model	DCR-based Deferral
0	x	x	Always defer	No
1	0	x	No recovery	No
1	1	0	No recovery	No
1	1	1	Recovery	Yes

Speculative load exceptions are categorized into three groups:

- Ones which always raise a fault
- Ones which always defer
- Ones which always raise a fault in the no-recovery model, but can defer based on the speculative deferral control bits in the DCR control register, in the recovery model.

Aborts, external interrupts, RSE or instruction-fetch-related faults that happen to occur on a speculative load are always raised (since they are not related to the speculative load instruction). Illegal Operation faults and Disabled Floating-point Register faults that occur on a speculative load are always raised.

Processing of exception conditions for speculative and speculative advanced loads is done in three stages: qualification, deferral and prioritization.

During the execution of a load instruction, multiple exception conditions may be detected simultaneously. For non-speculative loads these exception conditions are prioritized and only the highest priority one raises a fault. For speculative loads, however, some exception conditions may be deferred. As a result, it is possible for lower priority exceptions, which are not also deferred, to raise a fault. For some exception conditions, though, other lower priority conditions are meaningless, and are said to be qualified, or precluded. Exception qualification is described in Table 5-3.

Table 5-3. Exception Qualification

Exception Condition	Precluded by Concurrent Exception Condition	
Register NaT Consumption (NaT'ed address)	none	
Unimplemented Data Address	Register NaT Consumption	
Alternate Data TLB	Register NaT Consumption	Unimplemented Data Address
VHPT data	Register NaT Consumption	Unimplemented Data Address
Data TLB	Register NaT Consumption	Unimplemented Data Address
Data Page Not Present	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB
Data NaT Page Consumption	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB Data Page Not Present
Data Key Miss	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB Data Page Not Present

Table 5-3. Exception Qualification (Continued)

Exception Condition	Precluded by Concurrent Exception Condition	
Data Key Permission	Register NaT Consumption Unimplemented Data Address VHPT data Data TLB	Alternate Data TLB Data Page Not Present Data Key Miss
Data Access Rights	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB Data Page Not Present
Data Access Bit	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB Data Page Not Present
Data Debug	Register NaT Consumption	Unimplemented Data Address
Unaligned Data Reference	Register NaT Consumption	Unimplemented Data Address
Unsupported Data Reference	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB Data Page Not Present

After exception conditions are detected and qualified, the remaining exception conditions are checked for deferral. Deferral occurs after fault qualification and determines which memory access exceptions raised by speculative loads are automatically deferred by hardware.

Deferral is controlled by PSR.ed, PSR.it, PSR.ic, the speculative deferral control bits in the DCR, the exception deferral bit of the code page's instruction TLB entry (ITLB.ed), and the memory attribute of the referenced data page. The speculative load and speculative advanced load exception deferral conditions are as follows:

- When PSR.ic is 0 and regardless of the state of DCR, and ITLB.ed bits (see Table 5-2), all exception conditions related to the data reference are deferred.
- Regardless of the state of DCR, PSR.it, PSR.ic, and ITLB.ed bits, Unimplemented Data Address exception conditions and Data NaT Page Consumption exception conditions (caused by references to NaTPages) are always deferred.
- When PSR.it and ITLB.ed are both 1, and the appropriate DCR bit is 1 for the exception, the speculative load exception is deferred.
- When PSR.it and ITLB.ed are both 1, Unaligned Data Reference exception conditions are deferred.

The conditions for deferral are given in Table 5-4. See also "Default Control Register (DCR – CR0)" on page 2:25.

Table 5-4. Qualified Exception Deferral

Qualified Exception	Deferred if
Register NaT Consumption (NaT'ed address)	always
Unimplemented Data Address	always
Alternate Data TLB	!PSR.ic || (PSR.it && ITLB.ed && DCR.dm)
VHPT data	!PSR.ic || (PSR.it && ITLB.ed && DCR.dm)
Data TLB	!PSR.ic || (PSR.it && ITLB.ed && DCR.dm)
Data Page Not Present	!PSR.ic || (PSR.it && ITLB.ed && DCR.dp)
Data NaT Page Consumption	always
Data Key Miss	!PSR.ic || (PSR.it && ITLB.ed && DCR.dk)
Data Key Permission	!PSR.ic || (PSR.it && ITLB.ed && DCR.dx)
Data Access Rights	!PSR.ic || (PSR.it && ITLB.ed && DCR.dr)
Data Access Bit	!PSR.ic || (PSR.it && ITLB.ed && DCR.da)
Data Debug	!PSR.ic || (PSR.it && ITLB.ed && DCR.dd)
Unaligned Data Reference	!PSR.ic || (PSR.it && ITLB.ed)
Unsupported Data Reference	always

After checking for deferral, execution of a speculative load instruction proceeds as follows:

- When PSR.ed is 1, then a deferred exception indicator (NaT bit or NaTVal) is written to the load target register, regardless of whether it has an exception or not and regardless of the state of DCR, PSR.it, PSR.ic and the ITLB.ed bits.
- If PSR.ed is 0 and there is at least one exception condition which is neither precluded nor deferred, then a fault is taken corresponding to the highest-priority exception condition which is neither precluded nor deferred. Prioritization of non-deferred speculative load faults follows the same interruption priorities as non-speculative instruction faults (Table 5-5 on page 2:92). However, deferred speculative load faults do not take part in the prioritization. As a result, depending on DCR settings, a lower priority fault may be taken, even if a higher priority exception condition exists, but is deferred.
- If PSR.ed is 0 and there are exception conditions, but all are either precluded or deferred, then a deferred exception indicator (NaT bit or NaTVal) is written to the load target register.
- If PSR.ed is 0, and there are no exception conditions, and if the memory attribute of the referenced page is uncacheable or limited speculation, then a deferred exception indicator (NaT bit or NaTVal) is written to the load target register. See "Speculation Attributes" on page 2:67.
- Otherwise, the load executes normally.

If automatic hardware deferral is not enabled, software may still choose to defer exception processing (for speculative loads) at the time of the fault. If the code page has its ITLB.ed bit equal to 1, then the operating system may choose to defer a non-fatal exception. It is expected that the operating system will always defer fatal exceptions. To assist software in the deferral of non-fatal or fatal exceptions, the system architecture provides three additional resources: ISR.sp, ISR.ed, and PSR.ed.

ISR.sp indicates whether the exception was the result of a speculative or speculative advanced load. The ISR.ed bit captures the code page ITLB.ed bit, and allows deferral of a non-fatal exception due to a speculative load. If both the ISR.sp and ISR.ed bit are 1 on an interruption, then the operating system may defer a non-fatal exception by using the PSR.ed bit to perform the action of hardware deferral for one executed instruction. Software may use the same PSR.ed mechanism to defer fatal speculative load exceptions.

5.6 Interruption Priorities

Table 5-5 contains a complete list of the architecture defined interruptions (including IA-32), grouped according to type (aborts, interrupts, faults and traps), instruction set, and listed in priority order. Interruptions are delivered in priority order. If more than one instruction detects an interruption within a bundle, the interruption occurring in the lowest numbered instruction slot is raised. Lower priority faults and traps are discarded. Lower priority interrupts are held pending.

The shaded interruptions are disabled if the instruction generating the interruption is predicated off. All other interruptions are either "bundle related" (so the predicate bits do not affect them) or are caused by instructions that cannot be predicated off. Incomplete Register frame (IR) faults 6 through 17 are identical in behavior to faults 43, 48 through 59 (exclusive of 57) except they are of a higher priority. IR faults 6 through 17 can only be caused by mandatory RSE load operations that result from `br.ret`, or `rfi` instructions, but not from `loadrs` instructions (for details see Section "RSE Interruptions" on page 2:127).

The number in parenthesis after each vector name is the page number where the vector is described in detail.

Table 5-5. Interruption Priorities

Type	Instr. Set	Interruption Name		Vector Name	IA-32 Class[a]
Aborts	IA-32, Intel® Itanium™	1	Machine Reset (RESET)	PALE_RESET vector	N/A
		2	Machine Check (MCA)	PALE_CHECK vector	
Inter-rupts		3	Initialization Interrupt (INIT)	PALE_INIT vector	N/A
		4	Platform Management Interrupt (PMI)	PALE_PMI vector	
		5	External Interrupt (INT)	External Interrupt vector	
Faults	Intel® Itanium™	6	IR Unimplemented Data Address fault	General Exception vector	N/A
		7	IR Data Nested TLB fault	Data Nested TLB vector	
		8	IR Alternate Data TLB fault	Alternate Data TLB vector	
		9	IR VHPT Data fault	VHPT Translation vector	
		10	IR Data TLB fault	Data TLB vector	
		11	IR Data Page Not Present fault	Page Not Present vector	
		12	IR Data NaT Page Consumption fault	NaT Consumption vector	
		13	IR Data Key Miss fault	Data Key Miss vector	
		14	IR Data Key Permission fault	Key Permission vector	
		15	IR Data Access Rights fault	Data Access Rights vector	
		16	IR Data Access Bit fault	Data Access-Bit vector	
		17	IR Data Debug fault	Debug vector	

Table 5-5. Interruption Priorities (Continued)

Type	Instr. Set	Interruption Name	Vector Name	IA-32 Class[a]
Faults	IA-32	18 IA-32 Instruction Breakpoint fault	IA-32 Exception vector (Debug)	A
		19 IA-32 Code Fetch fault[b]	IA-32 Exception vector (GPFault)	
	IA-32, Intel® Itanium™	20 Alternate Instruction TLB fault	Alternate Instruction TLB vector	
		21 VHPT Instruction fault	VHPT Translation vector	
		22 Instruction TLB fault	Instruction TLB vector	
		23 Instruction Page Not Present fault	Page Not Present vector	
		24 Instruction NaT Page Consumption fault	NaT Consumption vector	
		25 Instruction Key Miss fault	Instruction Key Miss vector	
		26 Instruction Key Permission fault	Key Permission vector	
		27 Instruction Access Rights fault	Instruction Access Rights vector	
		28 Instruction Access Bit fault	Instruction Access-Bit vector	
	Intel® Itanium™	29 Instruction Debug fault	Debug vector	
	IA-32	30 IA-32 Instruction Length > 15 bytes	IA-32 Exception vector (GPFault)	B
		31 IA-32 Invalid Opcode fault	IA-32 Intercept vector (Instruction)	
		32 IA-32 Instruction Intercept fault	IA-32 Intercept vector (Instruction)	
	Intel® Itanium™	33 Illegal Operation fault	General Exception vector	
		34 Illegal Dependency fault	General Exception vector	
		35 Break Instruction fault	Break Instruction vector	
		36 Privileged Operation fault	General Exception vector	
	IA-32, Intel® Itanium™	37 Disabled Floating-point Register fault	Disabled FP-Register vector	B
		38 Disabled Instruction Set Transition fault	General Exception vector	
	IA-32	39 IA-32 Device Not Available fault	IA-32 Exception vector (DNA)	
		40 IA-32 FP Error fault[c]	IA-32 Exception vector (FPError)	C
	IA-32, Intel® Itanium™	41 Register NaT Consumption fault	NaT Consumption vector	
	Intel® Itanium™	42 Reserved Register/Field fault	General Exception vector	
		43 Unimplemented Data Address fault	General Exception vector	
		44 Privileged Register fault	General Exception vector	
		45 Speculative Operation fault	Speculation vector	

Table 5-5. Interruption Priorities (Continued)

Type	Instr. Set	Interruption Name	Vector Name	IA-32 Class[a]
	IA-32	46 IA-32 Stack Exception	IA-32 Exception vector (StackFault)	C
		47 IA-32 General Protection Fault	IA-32 Exception vector (GPFault)	
Faults	IA-32, Intel® Itanium™	48 Data Nested TLB fault	Data Nested TLB vector	
		49 Alternate Data TLB fault[d]	Alternate Data TLB vector	
		50 VHPT Data fault[d]	VHPT Translation vector	
		51 Data TLB fault[d]	Data TLB vector	
		52 Data Page Not Present fault[d]	Page Not Present vector	
		53 Data NaT Page Consumption fault[d]	NaT Consumption vector	
		54 Data Key Miss fault[d]	Data Key Miss vector	
		55 Data Key Permission fault[d]	Key Permission vector	
		56 Data Access Rights fault[d]	Data Access Rights vector	
		57 Data Dirty Bit fault	Dirty-Bit vector	
		58 Data Access Bit fault[d]	Data Access-Bit vector	
	Intel® Itanium™	59 Data Debug fault[d]	Debug vector	
		60 Unaligned Data Reference fault[d]	Unaligned Reference vector	
	IA-32	61 IA-32 Alignment Check fault	IA-32 Exception vector (AlignmentCheck)	C
		62 IA-32 Locked Data Reference fault	IA-32 Intercept vector (Lock)	
		63 IA-32 Segment Not Present fault	IA-32 Exception vector (NotPresent)	
		64 IA-32 Divide by Zero fault	IA-32 Exception vector (Divide)	
		65 IA-32 Bound fault	IA-32 Exception vector (Bound)	
		66 IA-32 Streaming SIMD Extension Numeric Error fault	IA-32 Exception vector (StreamSIMD)	
	Intel® Itanium™	67 Unsupported Data Reference fault	Unsupported Data Reference vector	
		68 Floating-point fault	Floating-point Fault vector	
Traps	Intel® Itanium™	69 Unimplemented Instruction Address trap	Lower-Privilege Transfer Trap vector	
		70 Floating-point trap	Floating-point Trap vector	
		71 Lower-Privilege Transfer trap	Lower-Privilege Transfer Trap vector	
		72 Taken Branch trap	Taken Branch Trap vector	
		73 Single Step trap	Single Step Trap vector	
	IA-32	74 IA-32 System Flag Intercept trap	IA-32 Intercept vector (SystemFlag)	D
		75 IA-32 Gate Intercept trap	IA-32 Intercept vector (Gate)	
		76 IA-32 INTO trap	IA-32 Exception vector (Overflow)	
		77 IA-32 Breakpoint (INT 3) trap	IA-32 Exception vector (Debug)	
		78 IA-32 Software Interrupt (INT) trap	IA-32 Interrupt vector (Vector#)	
		79 IA-32 Data Breakpoint trap	IA-32 Exception vector (Debug)	
		80 IA-32 Taken Branch trap	IA-32 Exception vector (Debug)	
		81 IA-32 Single Step trap	IA-32 Exception vector (Debug)	

[a] IA-32 Interruption Class, see Section 5.6.1 for details

[b] IA-32 Code Fetch faults include Code Segment Limit Violation and other Code Fetch checks defined in Section 6.2.3.3.

[c] IA-32 FP Error fault conditions detected on an IA-32 FP instruction are reported as a fault on the next IA-32 FP instruction that performs an FWAIT operation.

[d] If not deferred.

5.6.1 IA-32 Interruption Priorities and Classes

Table 5-5 establishes a well defined priority between faults, traps and interrupts (including IA-32). However, IA-32 instruction set generated interruptions are divided into interruption classes. While priority among these IA-32 interruption classes is well defined by the table (except as noted below), interruption priority within each IA-32 interruption class is implementation dependent and may vary from processor to processor as defined below:

Class A - Faults from fetching an instruction. Priority of IA-32 Instruction Breakpoint, IA-32 Code Fetch (GPFault(0)), and Instruction TLB faults (Alternate Instruction TLB fault to Instruction Access Bit fault) may vary based on instruction alignment and page boundaries in a model specific way. Faults are prioritized as defined in the table if the instruction does not span a virtual page. If an IA-32 instruction spans a virtual page, IA-32 Code Fetch faults (IA-32_Exception(GPFault)) due to code segment (CS) Limit violations can be raised above or below Instruction TLB faults as defined below:

- If the starting effective address of the IA-32 instruction exceeds the code segment limit, then the IA-32 Code Fetch fault has higher priority than any Instruction TLB faults. If the starting effective address of the IA-32 instruction is within the code segment limit, then Instruction TLB faults have higher priority for the starting effective address.
- If the IA-32 instruction spans a virtual page and the code segment limit is equal to the page boundary, the IA-32 Code Fetch fault has higher priority than any Instruction TLB faults on the second page. Otherwise if the code segment limit is greater than the page boundary, any Instruction TLB faults on the second page have higher priority than the IA-32 Code Fetch fault.

Class B - Faults from decoding an instruction. Priority of IA-32 Instruction Length, IA-32 Invalid Opcode, and IA-32 Instruction Intercept, Disabled Floating Point Register, Disabled Instruction Set Transition, and Device Not Available faults are model specific. If the IA-32 instruction spans a virtual page, IA-32 Instruction Length >15 byte Faults (IA-32_Exception(GPFault)) can have higher priority than Instruction TLB faults as defined below:

- If the IA-32 prefix bytes on the first page are >= 15 bytes, an IA-32 Instruction >15 byte fault (GPFault) is taken first regardless of any Instruction TLB faults on the second page.
- If the IA-32 prefix bytes on the first page are < 15 bytes, Instruction TLB faults on the second page may or may not have priority over any possible IA-32 Instruction Length fault.

Class C - Faults resulting from executing an instruction. Priority of faults is model specific and can vary across processor implementations. Most faults are related to data memory references, other fault priorities can vary due to model-specific differences across processor implementations. The memory fault priorities (IA-32 Stack Exception through Data Access Bit fault) defined in the table only apply to a single IA-32 data memory reference that does not cross a virtual page. If an IA-32 instruction requires multiple data memory references or a single data memory reference crosses a virtual page:

- If any given IA-32 instruction requires multiple data memory references, all possible faults are raised on the first data memory reference before any faults are checked on subsequent data memory references. This implies lower priority faults on an earlier memory reference will be raised before higher priority faults on a later data memory reference within a single IA-32 instruction. The order of data memory references initiated by an IA-32 instruction is implementation dependent and may vary from processor to processor. Software can not assume all higher priority data memory faults are raised before all lower priority data memory faults within a single IA-32 instruction.

- If a single IA-32 data memory reference crosses a virtual page, the processor checks for faults in a model specific order: Any faults present on one page are checked and reported before any faults are checked and reported on the other page. This implies that a single data reference that crosses a virtual page can raise lower priority data memory faults on one page before higher priority data memory faults are raised on the other page. For example, Data Key Miss faults (lower priority) on the first page could be raised before a Data TLB Miss Fault (higher priority) on the second page. Software can not assume all higher priority data memory faults are raised before all lower priority data memory faults within a single IA-32 instruction.

Class D - Traps on the current IA-32 instruction. Trap conditions are reported concurrently on the same exception vector or via a trap code specifying all concurrent traps.

5.7 IVA-based Interruption Vectors

Table 5-6 contains the processor's interruption vector table (IVT). The base of the IVT is held in the IVA control register. The size of the IVT is 32KB. The first 20 vectors are designed to provide more code space by allowing 64 bundles per vector (16 bytes per bundle) for performance-critical interruption handlers. The second 48 vectors provide 16 bundles per vector. Several vectors have more than one interruption associated with them. Information provided in the ISR allows the handler to distinguish which fault or trap caused the event.

Some vectors require additional software decoding to determine the cause of the interruption. Additional information for this decoding is provided in the ISR.code field. See Chapter 8, "Interruption Vector Descriptions" for a complete specification of the information supplied in the ISR for each of the vectors.

Note: PAL-based interruptions (RESET, MCA, INIT, and PMI) do not reference the IVT.

Table 5-6. Interruption Vector Table (IVT)

Offset	Vector Name	Interruption(s)	Page
0x0000	VHPT Translation vector	9, 21, 50	2:153
0x0400	Instruction TLB vector	22	2:155
0x0800	Data TLB vector	10, 51	2:156
0x0c00	Alternate Instruction TLB vector	20	2:157
0x1000	Alternate Data TLB vector	8, 49	2:158
0x1400	Data Nested TLB vector	7, 48	2:159
0x1800	Instruction Key Miss vector	25	2:160
0x1c00	Data Key Miss vector	13, 54	2:161
0x2000	Dirty-Bit vector	57	2:162
0x2400	Instruction Access-Bit vector	28	2:163
0x2800	Data Access-Bit vector	16, 58	2:164
0x2c00	Break Instruction vector	35	2:165
0x3000	External Interrupt vector	5	2:166
0x3400	Reserved		
0x3800	Reserved		
0x3c00	Reserved		
0x4000	Reserved		
0x4400	Reserved		

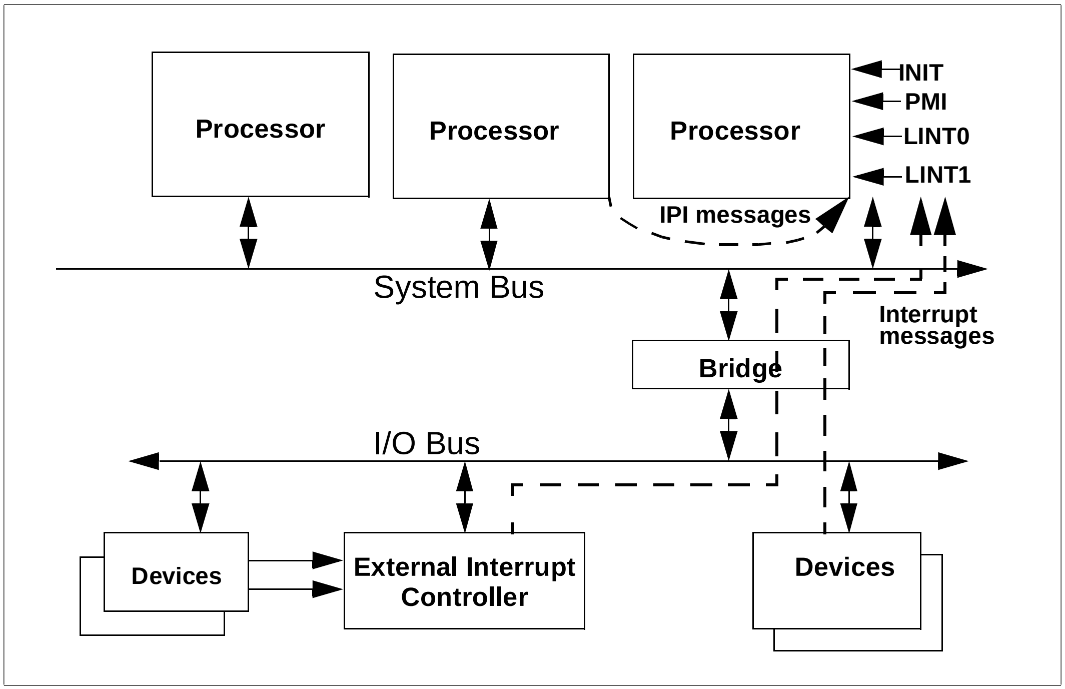
Table 5-6. Interruption Vector Table (IVT) (Continued)

Offset	Vector Name	Interruption(s)	Page
0x4800	Reserved		
0x4c00	Reserved		
0x5000	Page Not Present vector	11, 23, 52	2:167
0x5100	Key Permission vector	14, 26, 55	2:168
0x5200	Instruction Access Rights vector	27	2:169
0x5300	Data Access Rights vector	15, 56	2:170
0x5400	General Exception vector	6, 33, 34, 36, 38, 42, 43, 44	2:171
0x5500	Disabled FP-Register vector	37	2:173
0x5600	NaT Consumption vector	12, 24, 41, 53	2:174
0x5700	Speculation vector	45	2:176
0x5800	Reserved		
0x5900	Debug vector	17, 29, 59	2:177
0x5a00	Unaligned Reference vector	60	2:178
0x5b00	Unsupported Data Reference vector	67	2:179
0x5c00	Floating-point Fault vector	68	2:180
0x5d00	Floating-point Trap vector	70	2:181
0x5e00	Lower-Privilege Transfer Trap vector	69, 71	2:182
0x5f00	Taken Branch Trap vector	72	2:183
0x6000	Single Step Trap vector	73	2:184
0x6100	Reserved		
0x6200	Reserved		
0x6300	Reserved		
0x6400	Reserved		
0x6500	Reserved		
0x6600	Reserved		
0x6700	Reserved		
0x6800	Reserved		
0x6900	IA-32 Exception vector	18, 19, 30, 39, 40, 46, 47, 61, 63, 64, 65, 76, 77, 79, 80, 81	2:185
0x6a00	IA-32 Intercept vector	31, 32, 62, 74, 75	2:186
0x6b00	IA-32 Interrupt vector	78	2:187
0x6c00	Reserved		
	Reserved		
0x7f00	Reserved		

5.8 Interrupts

This section describes the programming model of the high performance interrupt architecture. As shown in Figure 5-3, interrupts are managed by the processor and by one or more intelligent external interrupt controllers or devices in the I/O subsystem. The processor is responsible for queuing and masking interrupts, sending and receiving inter-processor interrupt (IPI) messages, receiving interrupt messages from external interrupt controller(s), and managing local interrupt sources. This document describes the processor's interrupt control mechanism only; for details on external interrupt controllers or I/O devices refer to platform documentation.

Figure 5-3. Interrupt Architecture Overview



As defined in "Interruption Definitions" on page 2:79 there are three kinds of interrupts: initialization interrupts (INITs), platform management interrupts (PMIs), and external interrupts (INTs).

The processors and external interrupt controllers communicate over the processor's system bus with an implementation specific interrupt messaging protocol. Interrupts are generated by a number of different interrupt sources in the system:

- **External (I/O) devices** - Interrupt messages from any external source can be directed to any one processor by an external interrupt controller or by I/O devices capable of directly sending interrupt messages. An interrupt message informs the processor that an interrupt request is being made, and, in the case of PMIs and external interrupts, specifies a unique vector number for the interrupt. Interrupt messages are only issued on the "assertion edge" of an interrupt; "deassertion" of an interrupt does not result in an interrupt message.
- **Locally connected devices** - These interrupts originate on the processor's interrupt pins (LINT, INIT, PMI), and are always directed to the local processor. The LINT pins can be connected directly to an Intel 8259A-compatible external interrupt controller. The LINT pins are programmable to be either **edge-sensitive** or **level-sensitive**, and for the kind of interrupt that gets generated. If programmed to generate external interrupts, the vector number is a programmed constant per LINT pin. Only the LINT pins connected to the processor can directly generate level-sensitive interrupts (See "Edge- and Level-sensitive Interrupts" on page 2:114). LINT pins cannot be programmed to generate level-sensitive PMIs or INITs. The INIT and PMI pins generate their corresponding interrupts. For PMI pins a PMI vector 0 interrupt is generated.

- **Internal processor interrupts** - such as interval timer, performance monitoring, and corrected machine checks. These are always directed to the local processor. A unique vector number can be programmed for each source.
- **Other processors** - A processor can interrupt any individual processor, including itself, by sending an Inter-Processor Interrupt (IPI) message to a specific target processor. See "Inter-processor Interrupt Messages" on page 2:111.

The destination of an interrupt message is any one processor in the system, and is specified by a unique processor identifier. A different destination can be specified for each interrupt. There is no mechanism to "broadcast" a single interrupt to all processors in the system.

The following terms are used in the interrupt definition:

- The processor is said to **receive** an interrupt, if one of the processor's interrupt pins is asserted, the processor detected an interrupt message bus transaction containing the processor's unique identifier, or the processor detected an internal interrupt event.
- After receiving an interrupt, the processor internally holds the interrupt **pending**. The interrupt is said to be **pended** when it is received and held by the processor.
- For edge-sensitive interrupts, an external interrupt is held pending until the interrupt is acquired by software at which point it is said to be in-service. INITs and PMIs are held pending until the corresponding PAL vector is entered and PAL firmware clears the pending indication at which point they are said to be completed. For level-sensitive interrupts programmed through the LINT pins, the interrupt is held pending as long as the pin is asserted. Deassertion of a level-sensitive interrupt removes the pending indication (see "Edge- and Level-sensitive Interrupts" on page 2:114).
- The processor maintains an individual interrupt pending indication for INITs. Since external interrupts and PMIs are also signified by a unique interrupt **vector** number, the processor maintains individual pending indications per vector. An occurrence of an interrupt on a vector that is already marked as pending cannot be distinguished from previous interrupts on the same vector because the interrupts are pended in the same internal pending bit, and are therefore treated as "the same" interrupt occurrence.
- When interrupt delivery is enabled and the highest priority pending interrupt is unmasked (as defined below), the processor **accepts** the pending interrupt, interrupts the control flow of the processor and transfers control to the software interrupt handler.
- An external interrupt is said to be **in-service** when software **acquires** the interrupt vector from the processor by reading the IVR register (see "External Interrupt Vector Register (IVR – CR65)" on page 2:106). The processor then removes the pending indication for the interrupt vector. The processor maintains one in-service indicator for each unique vector number. Note that there are no in-service indicators for INITs and PMIs.
- Once an external interrupt is in-service it remains so until software indicates service for that external interrupt is **complete.** By writing to the EOI register (see "End of External Interrupt Register (EOI – CR67)" on page 2:107) software indicates that service for the highest-priority in-service external interrupt is complete. The processor then removes the in-service indication for the highest-priority external interrupt vector. INITs and PMIs are completed when PAL firmware clears the corresponding pending indication.
- The **priority** of interrupts is defined in Table 5-7. Entry *A* is higher priority than interrupt *B*, if entry *A* appears at a higher location in the table than entry *B*. Interrupt priority is used to select interrupts that require urgent service over less urgent interrupt requests.

- Interrupt **delivery** is **enabled** when software programs the processor to accept any unmasked interrupt. INITs delivery is enabled when PSR.mc is 0. PMIs delivery is enabled when PSR.ic is 1. For Itanium-based code execution, external interrupts delivery is enabled when PSR.i is 1.
- **Masking** applies only to external interrupts. Unmasked interrupts are those external interrupts of higher priority than the highest priority external interrupt vector currently in-service (if any) and whose priority level is higher than the current priority masking level specified by the TPR register (see "Task Priority Register (TPR – CR66)" on page 2:106). Masking conditions are defined in Table 5-7. PSR.i does not affect masking of external interrupts.

Figure 5-4 shows how this terminology is applied to the handling of a PAL-based interrupt. Similarly, Figure 5-5 shows the handing of a vectored external interrupt *n*. Both figures show the different states and transitions interrupts go through.

Figure 5-4. PAL-based Interrupt States

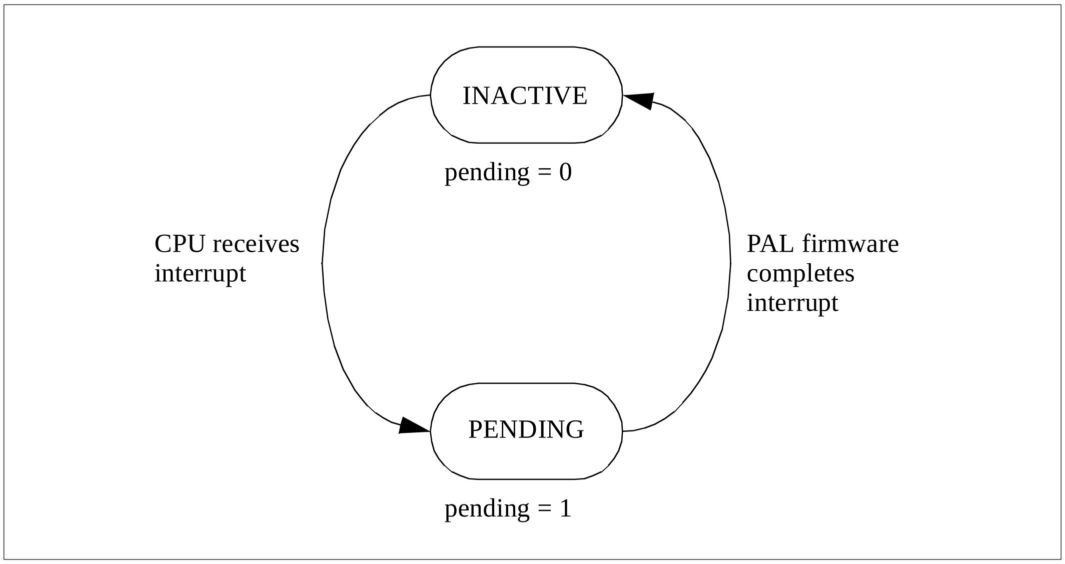
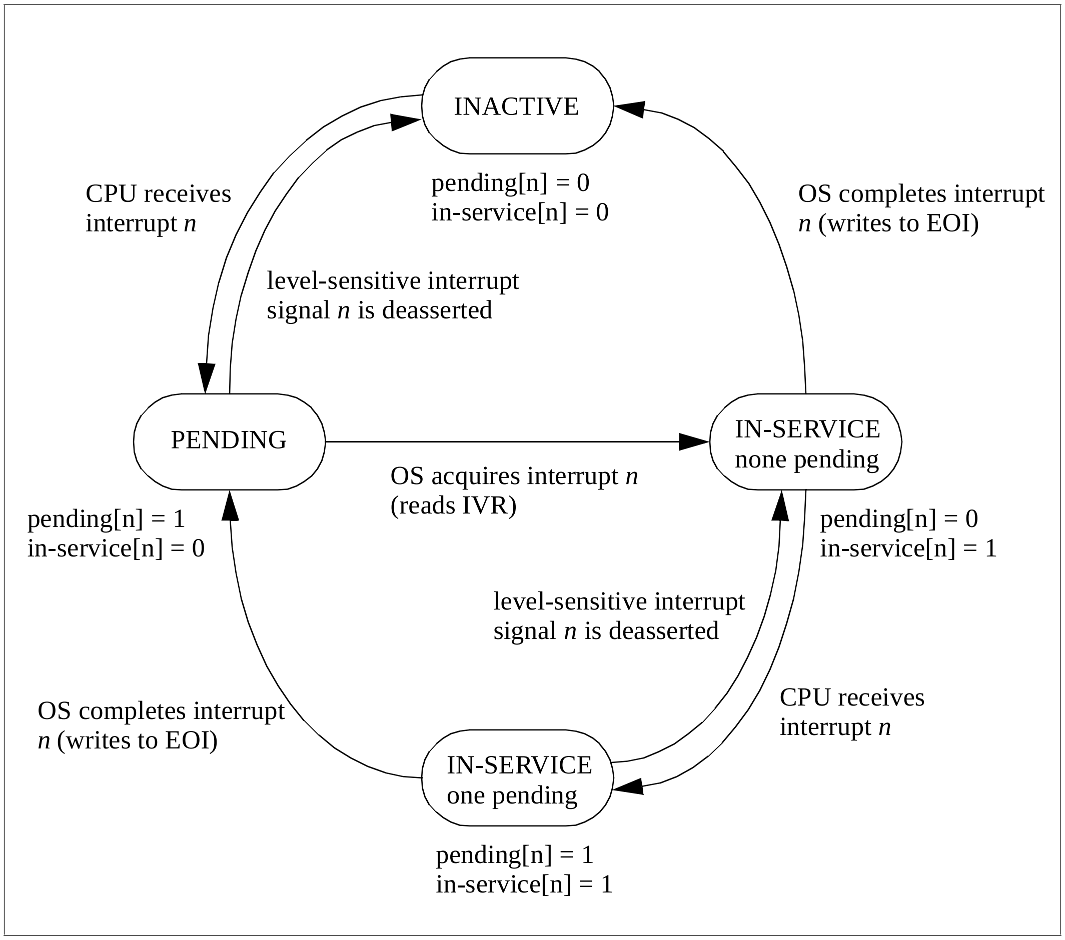


Figure 5-5. External Interrupt States



5.8.1 Interrupt Vectors and Priorities

As indicated in Table 5-5 on page 2:92, INITs have higher priority than PMIs, which in turn have higher priority than external interrupts. PMIs and external interrupts are further prioritized by vector number.

PMIs have a separate vector space from external interrupts. PMI vectors 0-3 can be used by platform firmware. PMI vectors 4 and above are reserved for use by processor firmware. Assertion of the processor’s PMI pin results in PMI vector number 0. PMI vector priorities are described in Chapter 11, “Processor Abstraction Layer”.

Each external interrupt (INT) in the system is distinguished from other external interrupts by a unique vector number. There are 256 distinct vector numbers in the range 0 - 255. Vector numbers 1 and 3 through 14 are reserved for future use. Vector number 0 (ExtINT) is used to service Intel 8259A-compatible external interrupt controllers. Vector number 2 is used for the Non-Maskable Interrupt (NMI). The remaining 240 external interrupt vector numbers (16 through 255) are available for general operating system use. Table 5-7 summarizes the interrupt priority model.

Table 5-7. Interrupt Priorities, Enabling, and Masking

Priority	Priority Class	Interrupt	Vector Number	Interrupt Delivery Enabled	Interrupt Unmasked Condition
Highest	n/a	INIT	n/a	if PSR.mc is 0	Always
		PMI	0..3	if PSR.ic is 1	Always
		INT	2 (NMI)	if PSR.i is 1[a]	Interrupt is higher priority than all in-service external interrupts
			0 (ExtINT)		TPR.mmi is 0, and interrupt is higher priority than all in-service external interrupts
	15		240..255		TPR.mmi is 0, and interrupt is higher priority than all in-service external interrupts, and Vector Number{7:4} > TPR.mic
	14		224..239		
	13		208..223		
	12		192..207		
	11		176..191		
	10		160..175		
	9		144..159		
	8		128..143		
	7		112..127		
	6		96..111		
	5		80..95		
	4		64..79		
	3		48..63		
	2		32..47		
Lowest	1		16..31		

[a] For Itanium-based code execution external interrupt delivery is enabled if PSR.i is 1. For IA-32 code execution external interrupt delivery is enabled if (PSR.i AND (!CFLAG.if OR EFLAG.if)) is true.

NMI (vector 2) has higher interrupt priority than ExtINT (vector 0), which has higher priority than external interrupt vectors 16 through 255.

External interrupts vectors 16 through 255 are divided into 15 interrupt priority classes. Sixteen different interrupt vectors share a single interrupt priority class, with class 1 being the lowest priority and class 15 being the highest. For these external interrupts, higher number external interrupts have priority over lower number external interrupts, including those within the same priority class.

Vector number 15 is used to indicate that the highest priority pending interrupt in the processor is at a priority level that is currently masked or there are no pending external interrupts. This encoding is referred to as a “spurious” interrupt.

5.8.2 Interrupt Enabling and Masking

Upon receiving an interrupt, the processor holds the interrupt pending internally until interrupt delivery is enabled and, in the case of external interrupts, the interrupt is unmasked. When all of the interrupt enabling and unmasking conditions are satisfied (see Table 5-7), the processor accepts the pending interrupt, interrupts the control flow of the processor, and transfers control to the External Interrupt handler for external interrupts, or to PAL firmware for INITs and PMIs.

Note: The TPR controls the masking of external interrupts. TPR is described in “Task Priority Register (TPR – CR66)” on page 2:106.

The processor provides nested interrupt priority support for external interrupt vectors 0, 2, and 16 through 255 by:

- Automatically masking external interrupts of equal or lower priority than the highest priority external interrupt currently in-service. This raises the in-service external interrupt masking level when each external interrupt begins service by an IVR read.
- Associating EOI writes with the highest priority in-service external interrupt, and removing the in-service indication for this external interrupt. This lowers the in-service masking level to that of the next highest priority currently in-service external interrupt (if any).

This mechanism allows software external interrupt handlers to be interrupted by higher priority external interrupts.

For example, assume software acquires an external interrupt vector 45 by reading IVR. During the service of this interrupt other external interrupts can still be received and are pended. If software sets PSR.i to a 1, pending external interrupts of equal or lower priority than 45 are masked. However, a higher priority pending external interrupt can be accepted by the processor (provided it is not masked by TPR.mmi or TPR.mic). Assuming external interrupt vector 80 is received by the processor, the processor will accept the interrupt by interrupting the control flow of the processor. During the service of this interrupt, external interrupts of equal or lower priority than vector 80 are masked. When EOI is issued by software, the processor will remove the in-service indication for external interrupt vector 80. External interrupt masking will then revert back to the next highest priority in-service external interrupt, vector 45. External interrupt vectors of equal or lower priority than vector 45 would remain masked until EOI is issued by software. The in-service indication for vector 45 is then removed by the write to EOI.

5.8.2.1 Re-enabling External Interrupt Delivery

When emerging from code in which external interrupt delivery is disabled and interruption state collection is turned off, the following minimal code sequence describes the architectural method with which to re-enable interruption collection and enable external interrupts:

```
ssm PSR.ic        // enable interruption collection
;;
srlz.d            // guarantee that interruption collection is enabled
ssm PSR.i         // enable external interrupts
```

The processor does not ensure that enabling external interrupts is immediately observed after the `ssm` PSR.i instruction. Software must perform a data serialization operation after `ssm` PSR.i to ensure that external interrupt delivery is enabled prior to a given point in program execution.

5.8.2.2 External Interrupt Sampling

Assuming that external interrupt delivery is currently disabled (PSR.i is 0), the following minimal code sequence describes the architectural method with which to briefly open the external interrupt window for external interrupt sampling (typically PSR.ic is 1 to enable interruption collection):

```
ssm PSR.i
;;
srlz.d           // external interrupts may be sampled anywhere here
;;
rsm PSR.i
```

The stop following the `srlz.d` instruction in the above code sequence is required to force the Reset System Mask (`rsm`) instruction into a subsequent instruction group. The stop guarantees that the `srlz.d` will open the external interrupt window for at least one cycle before the `rsm` instruction closes it again.

Note: In the above code sequence, the effect of disabling interrupts due to the `rsm` instruction is observed on the next instruction following the `rsm`.

5.8.2.3 Disabling of External Interrupt Delivery and rsm

When the current privilege level is zero, an `rsm` instruction whose mask includes PSR.i may cause external interrupt delivery to be disabled for an implementation-dependent number of instructions, even if the qualifying predicate for the `rsm` instruction is false. Architecturally, the extents of this delivery disable “window” are defined as follows:

1. External interrupt delivery may be disabled for any instructions in the same instruction group as the `rsm`, including those that precede the `rsm` in sequential program order, regardless of the value of the qualifying predicate of the `rsm` instruction.
2. If the qualifying predicate of the `rsm` is true, then external interrupt delivery is disabled immediately following the `rsm` instruction.
3. If the qualifying predicate of the `rsm` is false, then external interrupt delivery may be disabled until the next data serialization operation that follows the `rsm` instruction.

The delivery disable window is guaranteed to be no larger than defined by the above criteria, but it may be smaller, depending on the implementation.

When the current privilege level is non-zero, an `rsm` instruction whose mask includes PSR.i may briefly disable external interrupt delivery, regardless of the value of the qualifying predicate of the `rsm` instruction. However, the implementation guarantees that non-privileged code cannot lock out external interrupts indefinitely (e.g., via an arbitrarily long sequence of `rsm` PSR.i instructions with zero-valued qualifying predicates).

5.8.3 External Interrupt Control Registers

Software interacts with external interrupts by reading and writing the external interrupt control registers (CR64-81). These registers are summarized in Table 5-8, and are used to prioritize and deliver external interrupts, and to assign external interrupt vectors for processor-internal interrupt sources such as interval timer, performance monitoring, and corrected machine check.

The external interrupt control registers can only be accessed at privilege level 0, otherwise a Privileged Operation fault is raised.

Table 5-8. External Interrupt Control Registers

Register	Name	Description
CR64	LID	Local ID
CR65	IVR	External Interrupt Vector Register (read only)
CR66	TPR	Task Priority Register
CR67	EOI	End Of External Interrupt
CR68	IRR0	External Interrupt Request Register 0 (read only)
CR69	IRR1	External Interrupt Request Register 1 (read only)
CR70	IRR2	External Interrupt Request Register 2 (read only)
CR71	IRR3	External Interrupt Request Register 3 (read only)
CR72	ITV	Interval Timer Vector
CR73	PMV	Performance Monitoring Vector
CR74	CMCV	Corrected Machine Check Vector
CR80	LRR0	Local Redirection Register 0
CR81	LRR1	Local Redirection Register 1

5.8.3.1 Local ID (LID – CR64)

The LID register contains the processor's local interrupt identifier. Two fields (id and eid) serve as the processor's physical name for all interrupt messages (external interrupts, INITs, and PMIs). LID is loaded by firmware during platform initialization based on the processor's physical location within the system. Processors receiving an interrupt message on the system bus compare their id/eid fields with the target address for the interrupt message. In case of a match, the processor receives the interrupt and internally holds the interrupt pending.

LID is a read-write register. To ensure that future arriving interrupts see the updated LID value by a given point in program execution, software must perform a data serialization operation after a LID write and prior to that point. The Local ID fields are defined in Figure 5-6 and Table 5-9.

Figure 5-6. Local ID (LID – CR64)

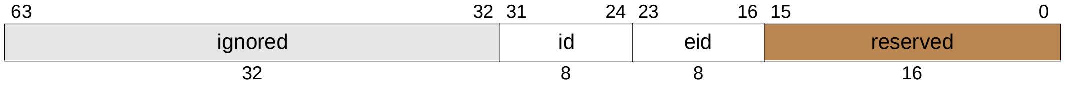


Table 5-9. Local ID Fields

Field	Bits	Description
id/eid	31:16	The low order bits of id correspond to a unique, geographically significant address of the processor on the local system bus. The high order bits of id and the eid field correspond to a unique address of the local system bus within the entire system. These fields are initialized by platform firmware to an implementation-dependent value and should not be modified by software. The two fields corresponds to physical address bits{19:4} of the inter-processor interrupt message.

5.8.3.2 External Interrupt Vector Register (IVR – CR65)

A read of IVR returns the highest priority, pending, unmasked external interrupt vector, independent of the value of PSR.i. The external interrupt vector is an 8-bit encoded number. If there are no pending external interrupts or all external interrupts are currently masked, IVR returns the “spurious” interrupt indication (vector 15). IVR fields are shown in Figure 5-7. See “Interrupt Unmasked Condition” column in Table 5-7 on page 2:102 for masking conditions.

IVR reads also have two atomic side effects:

- The interrupt pending bit in IRR is cleared for the reported external interrupt vector. Subsequent IVR reads will not report the interrupt as pending unless a new interrupt was pended for the specified interrupt vector.
- The processor marks the interrupt vector as being in-service and masks all pending external interrupts with equal or lower priority until software writes the end-of-interrupt (EOI) register for the in-service interrupt.

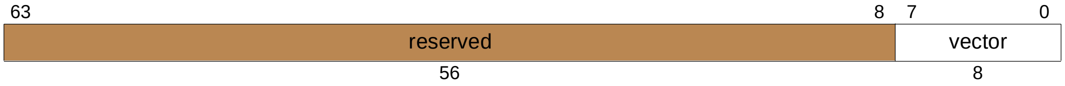
To ensure IVR side effects are observed by a given point in program execution (e.g., before the next IVR read, EOI write, or PSR.i write to enable external interrupt delivery), software must perform a data serialization operation after an IVR read and prior to that point. To ensure that the reported external interrupt vector is correctly masked before the next IVR read, software must perform a data serialization operation after a TPR or EOI write and prior to that IVR read.

Software must be prepared to service any possible external interrupt if it reads IVR, since IVR reads are destructive and removes the highest priority pending external interrupt (if any).

IVR is a read-only register; writes to IVR result in a Illegal Operation fault.

IVR reads do not issue an external INTA cycle. If the interrupt vector must be acquired from an Intel 8259A-compatible external interrupt controller, software should perform a load from the INTA byte. See “Interrupt Acknowledge (INTA) Cycle” on page 2:114 for details.

Figure 5-7. External Interrupt Vector Register (IVR – CR65)



5.8.3.3 Task Priority Register (TPR – CR66)

The processor’s Task Priority Register (TPR) provides the ability to create additional masking of external interrupts based on a “priority class”. The 240 external interrupt vectors (16 - 255) are divided into 15 priority classes of 16 numerically contiguous interrupt vectors each. The value written in TPR.mic masks all external interrupts of equal or lower priority classes.

To ensure that new priority levels are established by a given point in program execution (e.g., before PSR.i is set to 1), software must perform a data serialization operation after a TPR write and prior to that point. A data serialization operation must be performed after TPR is written and before IVR is read to ensure that the reported IVR vector is correctly masked. The TPR fields are described in Figure 5-8 and Table 5-10.

Figure 5-8. Task Priority Register (TPR – CR66)

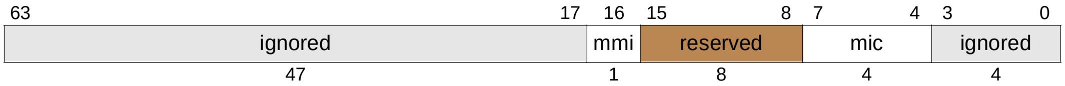


Table 5-10. Task Priority Register Fields

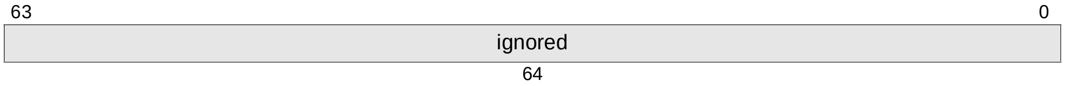
Field	Bits	Description
mic	7:4	Mask Interrupt Class: all external interrupt vectors of equal or lower priority classes then the TPR.mic field are masked. For example, if mic field is 4, interrupt priority classes 1, 2, 3, and 4 are masked. A TPR.mic value of 0 has no masking effect; a value of 15 will mask all external interrupt vectors in the range 16 - 255. TPR.mic has no effect on external interrupt vectors 0 and 2, INITs and PMIs. See "Processor Interrupt Block" on page 2:110.
mmi	16	Mask Maskable Interrupts: When 1, masks all external interrupts other than NMI (vector 2). When 0, external interrupt vectors 16 - 255, are masked by the TPR.mic field.

5.8.3.4 End of External Interrupt Register (EOI – CR67)

A write to the EOI (end-of-external interrupt) register, shown in Figure 5-9, indicates that software has finished servicing the highest priority in-service external interrupt. The processor removes its internal in-service indication for the highest priority currently in-service external interrupt vector. Pending external interrupts are then masked by the next highest priority in-service external interrupt (if any).

Writes to EOI affect the local processor only, and do not propagate to other processors or external interrupt controllers.

Figure 5-9. End of External Interrupt Register (EOI – CR67)



EOI is a read-write register. Reads return 0. Data associated with the EOI writes is ignored.

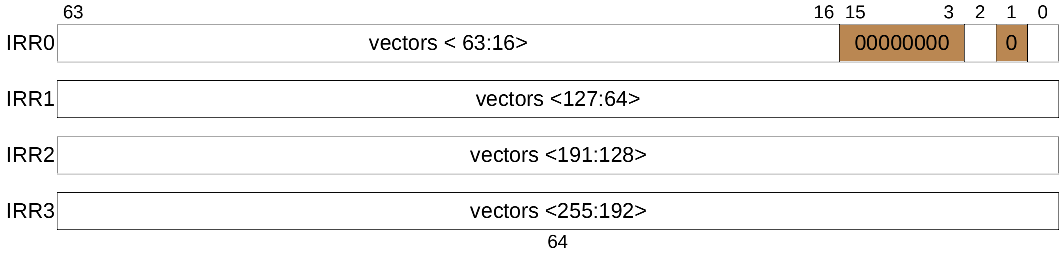
To ensure that the previous in-service interrupt indication has been cleared by a given point in program execution, software must perform a data serialization operation after an EOI write and prior to that point. To ensure that the reported IVR vector is correctly masked before the next IVR read, software must perform a data serialization operation after an EOI write and prior to that IVR read.

5.8.3.5 External Interrupt Request Registers (IRR0-3 – CR68,69,70,71)

Four 64-bit read-only External Interrupt Request Registers (IRR0-3, see Figure 5-10) provide the capability for software to determine the set of pending asynchronous external interrupts. IRR0 contains vectors <63:0> where vector 0 is in bit position 0, IRR1 contains vectors <127:64>, IRR2 contains vectors <191:128>, and IRR3 contains vectors <255:192>. A bit in the IRR, corresponding to the pending interrupt vector number, is set when the processor receives an external interrupt. The IRR bit is cleared when software reads the IVR and the vector number corresponding to the IRR bit value is returned in the IVR. The IRR bit is also cleared when a level-sensitive external interrupt signal is deasserted, effectively removing the pending interrupt.

Since IRR0-3 are read-only registers, writes to these registers result in Illegal Operation faults.

Figure 5-10. External Interrupt Request Register (IRR0-3 – CR68, 69, 70, 71)



5.8.3.6 Interval Timer Vector (ITV – CR72)

ITV specifies the external interrupt vector number for Interval Timer Interrupts. To ensure that subsequent interval timer interrupts reflect the new state of the ITV by a given point in program execution, software must perform a data serialization operation after an ITV write and prior to that point. See Figure 5-11 and Table 5-11 for the definitions of the ITV fields.

Figure 5-11. Interval Timer Vector (ITV – CR72)

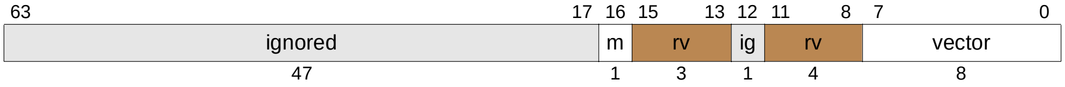


Table 5-11. Interval Timer Vector Fields

Field	Bits	Description
vector	7:0	External interrupt vector number to use when generating an Interval Timer interrupt. Vector values can be 0, 2 or 16-255. All other vectors are ignored and reserved for future use.
m	16	Mask: When 1, occurrences of Interval Timer interrupts are discarded and not pended. When 0, occurrences of Interval Timer interrupts are pended.

5.8.3.7 Performance Monitoring Vector (PMV – CR73)

PMV specifies the external interrupt vector number for Performance Monitoring overflow interrupts. To ensure that subsequent performance monitor interrupts reflect the new state of PMV by a given point in program execution, software must perform a data serialization operation after a PMV write and prior to that point. See Figure 5-12 and Table 5-12 for the definitions of the PMV fields.

Figure 5-12. Performance Monitor Vector (PMV – CR73)

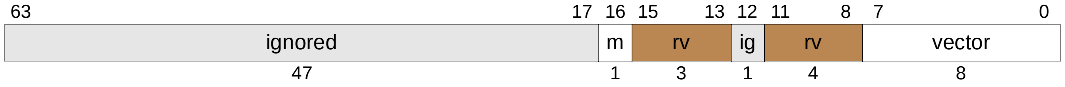


Table 5-12. Performance Monitor Vector Fields

Field	Bits	Description
vector	7:0	Vector number to use when generating a Performance Monitor interrupt. Vector values can be 0, 2, or 16-255. All other vectors are ignored and reserved for future use.
m	16	Mask: When 1, occurrences of Performance Monitor interrupts are discarded and not pended. When 0, occurrences of Performance Monitor interrupts are pended.

5.8.3.8 Corrected Machine Check Vector (CMCV – CR74)

CMCV specifies the external interrupt vector number for Corrected Machine Checks. To ensure that subsequent corrected machine check interrupts reflect the new state of CMCV by a given point in program execution, software must perform a data serialization operation after a CMCV write and prior to that point. See Figure 5-13 and Table 5-13 for the CMCV field definitions.

Figure 5-13. Corrected Machine Check Vector (CMCV – CR74)

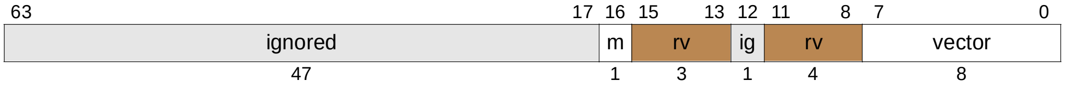


Table 5-13. Corrected Machine Check Vector Fields

Field	Bits	Description
vector	7:0	Vector number to use when generating a Corrected Machine Check. Vector values can be 0, 2, or 16 - 255. All other vectors are ignored and reserved for future use.
m	16	Mask: When 1, occurrences of Corrected Machine Check interrupts are discarded and not pended. When 0, occurrences of Corrected Machine Check interrupts are pended.

5.8.3.9 Local Redirection Registers (LRR0-1 – CR80,81)

Local Redirection Registers (LRR0-1) steer external signal based interrupts that are directly connected to the local processor to a specific external interrupt vector. All processors support two direct external interrupt pins. These external interrupt signals (pins) are referred to as Local Interrupt 0 (LINT0) and Local Interrupt 1 (LINT1).

To ensure that subsequent interrupts from LINT0 and LINT1 reflect the new state of LRR prior to a given point in program execution, software must perform a data serialization operation after an LRR write and prior to that point. The LRR fields are defined in Figure 5-14 and Table 5-14.

Figure 5-14. Local Redirection Register (LRR – CR80,81)

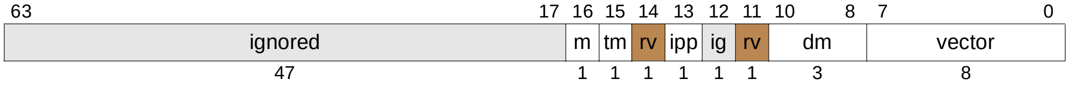


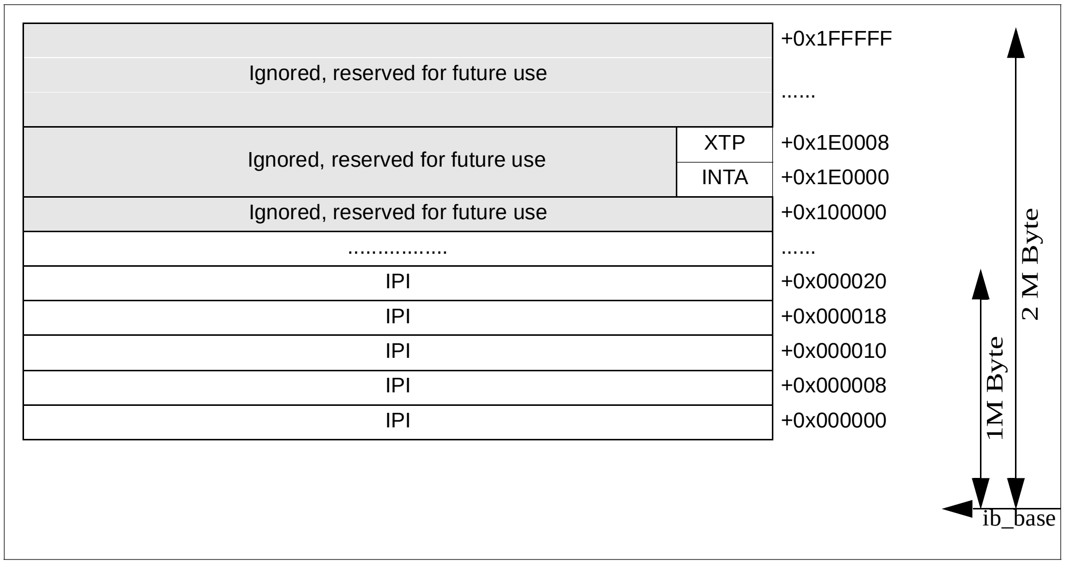
Table 5-14. Local Redirection Register Fields

Field	Bits	Description	
vector	7:0	External interrupt vector number to use when generating an interrupt for this entry. For INT delivery mode, allowed vector values are 0, 2, or 16-255. All other vectors are ignored and reserved for future use. For all other delivery modes this field is ignored.	
dm	10:8	000	INT – pend an external interrupt for the vector number specified by the vector field in LRR. Allowed vector values are 0, 2, or 16-255. All other vector numbers are ignored and reserved for future use.
		001	reserved
		010	PMI – pend a Platform Management Interrupt Vector number 0 for system firmware. The vector field is ignored.
		011	reserved
		100	NMI – pend a Non-Maskable Interrupt. This interrupt is pended at external interrupt vector number 2. The vector field is ignored.
		101	INIT – pend an Initialization Interrupt for system firmware. The vector field is ignored.
		110	reserved
		111	ExtINT – pend an Intel 8259A-compatible interrupt. This interrupt is delivered at external interrupt vector number 0. For details on servicing ExtINT external interrupts see "Interrupt Acknowledge (INTA) Cycle" on page 2:114. The vector field is ignored.
ipp	13	Interrupt Pin Polarity – specifies the polarity of the interrupt signal. When 0, the signal is active high. When 1, the signal is active low.	
tm	15	Trigger Mode – When 0, specifies edge sensitive interrupts. If the m field is 0, assertion of the corresponding LINT pin pends an interrupt for the specified vector corresponding to the dm field. The pending interrupt indication is cleared by software servicing the interrupt. When 1, specifies level sensitive interrupts. If the m field is 0, assertion of the corresponding LINT pin pends an external interrupt for the specified vector. Deassertion of the corresponding LINT pin clears the pending interrupt indication. The processor has undefined behavior if the dm and tm fields specify level sensitive PMIs or INITs.	
m	16	Mask – When 1, edge or level occurrences of the local interrupt pins are discarded and not pended. When 0, edge or level occurrences of local interrupt pins are pended.	

5.8.4 Processor Interrupt Block

Inter-Processor Interrupt (IPI) messages, Interrupt Acknowledge (INTA) cycles, and External Task Priority (XTP) cycles on the processor system bus are initiated by software by accessing a special physical memory range known as the "Processor Interrupt Block". Figure 5-15 defines its memory layout. The entire 2 MByte Processor Interrupt Block is relocatable by a PAL firmware call and must be aligned on a 2 MByte boundary; by default, the block is located at physical address 0x0000 0000 FEE0 0000.

Figure 5-15. Processor Interrupt Block Memory Layout



The Inter-Processor Interrupt region occupies the lower half of the Processor Interrupt Block; by default its physical address range is 0x0000 0000 FEE0 0000 through 0x0000 0000 FEEF FFFF. A processor generates Inter-Processor Interrupts by performing an aligned 8-byte store to this memory region.

The Processor Interrupt Block does not support all forms of memory operations. Unsupported memory accesses result in undefined processor operation.

- When targeted at the inter-processor interrupt delivery region (lower half of the Processor Interrupt Block), the following memory operations are undefined: instruction fetch, RSE accesses, or memory read references (only writes are permitted), references other than aligned 8-byte accesses, and references through any memory attribute other than UC.
- When targeted at the upper half of the Processor Interrupt Block, the following memory operations are undefined: instruction fetches, references other than 1-byte accesses, and references through any memory attribute other than UC.

5.8.4.1 Inter-processor Interrupt Messages

A processor can interrupt any individual processor, including itself**,** by issuing an inter-processor interrupt message (IPI). A processor generates an IPI by storing an 8-byte interrupt command to an 8-byte aligned address in the interrupt delivery region of the Processor Interrupt Block defined in “Processor Interrupt Block” on page 2:110. (If the address is not 8-byte aligned, the processor must either generate an Unaligned Data Reference Fault, see Section “Memory Datum Alignment and Atomicity” on page 2:77, or have undefined behavior). The address being stored to designates the target processor to receive the interrupt. The store address and data format of the inter-processor interrupt message are defined in Figure 5-16 and Figure 5-17. The data fields are defined in Table 5-16. The address processor identifier fields specify the target processor and are defined in Table 5-15.

Figure 5-16. Address Format for Inter-processor Interrupt Messages

63 20	19 12	11 4	3	2 0
ib_base	id	eid	ir	0
	8	8	1	3

Figure 5-17. Data Format for Inter-processor Interrupt Messages

63 11	10 8	7 0
ignored, reserved for future use	dm	vector
53	3	8

Table 5-15. Address Fields for Inter-processor Interrupt Messages

Field	Bits	Description
ir	3	Interrupt Redirection bit. The processor propagates the Interrupt Redirection bit along with the Inter-Processor Interrupt (IPI) message into the external system. When this bit is 0, the external system must send the IPI to the processor specified by the id/eid fields. When this bit is 1 on platforms that support interrupt redirection, the external system may perform interrupt load balancing and send the IPI to a processor with the lowest External Task Priority level. Alternatively, the external system may ignore the Interrupt Redirection bit and send the IPI message to the processor specified by the eid/id fields. Software must always program a valid eid/id field since the external system may or may not redirect the interrupt. If the eid/id field is not programmed with the address of a valid destination processor the IPI message may be lost. See "External Task Priority (XTP) Cycle" on page 2:114 for details on External Task Priority levels. On platforms that do not support interrupt redirection, software must not set the Interrupt Redirection bit to 1. Doing so will result in undefined behavior. Software can consult system specific firmware to determine if the Interrupt Redirection feature is supported by the external system.
id/eid	19:4	Specify the target processor. See Table 5-9 on page 2:105 for a definition of these fields.
ib_base	63:20	Physical Base address of Processor Interrupt Block. This is a PAL relocatable physical address. The default is 0x0000 0000 FEE. See "Processor Interrupt Block" on page 2:110 Based on the processor model some of the high order physical address bits may be reserved.

Table 5-16. Data Fields for Inter-processor Interrupt Messages

Field	Bits	Description	
vector	7:0	Vector number for the interrupt. For INT delivery, allowed vector values are 0, 2, or 16-255. All other vectors are ignored and reserved for future use. For PMI delivery, allowed PMI vector values are 0-3. All other PMI vector values are reserved for use by processor firmware.	
dm	10:8	000	INT – pend an external interrupt for the specified vector to the processor listed in the destination. Allowed vector values are 0, 2, or 16-255. All other vector numbers are ignored and reserved for future use.
		001	Reserved
		010	PMI – pend a PMI interrupt for the specified vector to the processor listed in the destination. Allowed PMI vector values are 0-3. All other PMI vector values are reserved for use by processor firmware.
		011	Reserved
		100	NMI – pend an external interrupt as an NMI (vector 2) to the processor listed in the destination. The vector field is ignored.
		101	INIT – pend an Initialization Interrupt for platform firmware on the processor listed in the destination. The vector field is ignored.
		110	Reserved
		111	ExtINT – pend an Intel 8259A-compatible interrupt. This interrupt is delivered at external interrupt vector number 0. For details on servicing ExtINT external interrupts see "Interrupt Acknowledge (INTA) Cycle" on page 2:114. The vector number field is ignored.
ignored	63:11	Ignored, reserved for future use	

5.8.4.2 Interrupt and IPI Ordering

Interrupt messages from external device(s), or external interrupts routed to the processor's LINT pins, may arrive at one or more processors and become pending in any order. No ordering is enforced by the processor or the platform.

As observed by a receiving processor, IPIs emitted from the same issuing processor may be pended in any order, even when the receiving processor and the issuing processor are the same.

As observed by a receiving processor, IPIs are pended after all prior loads and stores emitted by the same issuing processor are visible if and only if the IPI is issued with a `st.rel` (or proceeded by an `mf`), even when the receiving processor and the issuing processor are the same. For all other cases, no ordering is implied between IPI transactions and prior cacheable or uncached memory references.

As observed by a receiving processor, no ordering is implied between IPIs and subsequent loads/stores from the same issuing processor, even when the receiving processor and the issuing processor are the same. Subsequent loads or stores may become visible before an IPI is seen as pending. Data or instruction serialization operations, memory fences (`mf` or `mf.a`), or `st.rel` do not ensure an IPI is pending at the target processor (including self) by a given point in program execution on the local processor.

5.8.4.3 Interrupt Acknowledge (INTA) Cycle

Intel 8259A-compatible external interrupt controllers can not issue interrupt messages and therefore do not specify an external interrupt vector number when the interrupt request is generated. When accepting an external interrupt, software must inspect the vector number supplied by the IVR register. If the vector matches the vector number assigned to the external controller (can be ExtINT, or any other vector number based on software convention), software must acquire the actual external interrupt vector number from the external interrupt controller by issuing a 1-byte load from the INTA Byte.

The INTA Byte is located within the upper half of the Processor Interrupt Block, at offset 0x1E0000 from the base. A single byte load from the INTA address causes the processor to emit the INTA cycle on the processor system bus. An Intel 8259A-compatible external interrupt controller must respond with the actual interrupt vector number as the data to be loaded. If two INTA cycles are required by the external interrupt controller, the platform must provide this functionality.

Software must issue an EOI to the local processor, to clear the interrupt in-service indication for the vector associated with the external interrupt controller.

5.8.4.4 External Task Priority (XTP) Cycle

Some model-specific system configurations support an External Task Priority Register (XTPR) per processor in external bus logic. A processor’s XTPR can be modified by storing one byte of data to the processor’s XTP Byte address. This generates a special bus transaction required to change the processor’s XTPR within the system. Please refer to system specific documentation for XTPR bit format and field definitions. The processor does not interpret any data stored to the XTP Byte address and all data bits are passed to the external system unmodified.

XTPR is written by operating system code to notify the system that the processor’s current task priority has been changed. Based on this task priority information, system implementations can steer interrupt messages from the I/O subsystems to the processors that have registered the lowest task priority levels. The XTPR register is a system performance “hint”, and need not be updated by operating system code nor be implemented in all system configurations. If the system does not implement the XTPR, it must still accept a processor’s XTP cycle and discard it. Operating system code can issue XTPR updates regardless of external system support.

5.8.5 Edge- and Level-sensitive Interrupts

The processor’s LINT pins directly support edge and level sensitive interrupts, however all other interrupt sources are edge sensitive. A single external interrupt messages is issued only on the assertion of an interrupt by external interrupt controllers or devices, deassertion of an external interrupt sends no interrupt message to the processor. Since the processor removes the pending interrupt when the interrupt is serviced, the processor guarantees exactly-one interrupt acceptance for each external interrupt message. By definition external interrupt messages are edge sensitive.

Level sensitive external interrupts can be supported using edge sensitive interrupt messages as follows:

- Software services the external interrupt generated by an edge interrupt message.

- Software removes the external interrupt request from the requesting device, the device should then deassert its interrupt request line.
- To avoid spurious external interrupts, it is highly recommended that software issue a dummy read from the device to ensure that the interrupt request has been actually been removed before the interrupt is resampled in the next step.
- Software issues a command to the external interrupt controller to resample the interrupt (typically an external interrupt controller end-of-interrupt command). The external interrupt controller must issue another interrupt message back to the processor if service is still required by the processor for a given vector number. For example, if there are other devices still requiring service that are attached to the same level sensitive interrupt request line.

Register Stack Engine 6

The register stack engine (RSE) moves registers between the register stack and the backing store in memory without explicit program intervention. The RSE operates concurrently with the processor and can take advantage of unused memory bandwidth to dynamically issue register spill and fill operations. In this manner, the latency of register spill/fill operations can be overlapped with useful program work. The basic principles of the register stack are discussed in Section 4.1. This chapter presents the internal state, the programming model and the interruption behavior of the register stack engine.

6.1 RSE and Backing Store Overview

The register stack frames are mapped onto a set of physical registers which operate as a circular buffer containing the most recently created frames. The RSE spills and fills these physical registers to/from a backing store in memory. The RSE moves registers between the physical register stack and the backing store without explicit program intervention. As indicated in Figure 6-1, the RSE operates on the physical stacked registers outside of the currently active frame (as defined by CFM). These registers contain the frames of the parent procedures of the current procedure.

As shown in Figure 6-1, the backing store is organized as a stack in memory that grows from lower to higher addresses. The Backing Store Pointer (BSP) application register contains the address of the first (lowest) memory location reserved for the current frame (i.e., the location at which GR32 of the current frame will be spilled). RSE spill/fill activity occurs at addresses below what is contained in the BSP since the RSE spills/fills the frames of the current procedure's parents. The BSPSTORE application register contains the address at which the next RSE spill will occur. The address register which corresponds to the next RSE fill operation, the BSP load pointer, is not architecturally visible. The addresses contained in BSP and BSPSTORE are always aligned to an 8-byte boundary. The backing store contains the local area of each frame. The output area is not spilled to the backing store (unless it later becomes part of a callee's local area). Within each stack frame, lower-addressed registers are stored at lower memory addresses. RSE spills of NaTed stacked general registers are subject to the same memory update constraints as software spills (st8.spill) of NaTed static general registers (see "Register Spill and Fill" on page 2:55).

The RSE also spills/fills the NaT bits corresponding to the stacked registers. The NaT bits corresponding to the static subset must be spilled/filled as necessary by software. The NaT bits are the 65th bit of each general register. The NaT bits for the stacked subset are spilled/filled in groups of 63 corresponding to 63 consecutive physical stacked registers. When the RSE spills a register to the backing store the corresponding NaT bit is copied to the RSE NaT collection (RNAT) application register. Whenever bits 8:3 of BSPSTORE are all ones, the RSE stores RNAT to the backing store. As shown in Figure 6-2, this results in a backing store memory image in which every 63 register values are followed by a collection of NaT bits. Bit 0 of the NaT collection corresponds to the first (lowest addressed) of the 63 register values; bit 62 corresponds to the 63rd register value. Bit 63 of the NaT collection is always written as zero. When the RSE fills a stacked register from the backing store it also fills the register's NaT bit. Whenever bits 8:3 of the RSE backing store load pointer are all ones, the RSE reloads a NaT collection from the backing store. Bit 63 of the NaT collection is ignored when read from the backing store.

Figure 6-1. Relationship Between Physical Registers and Backing Store

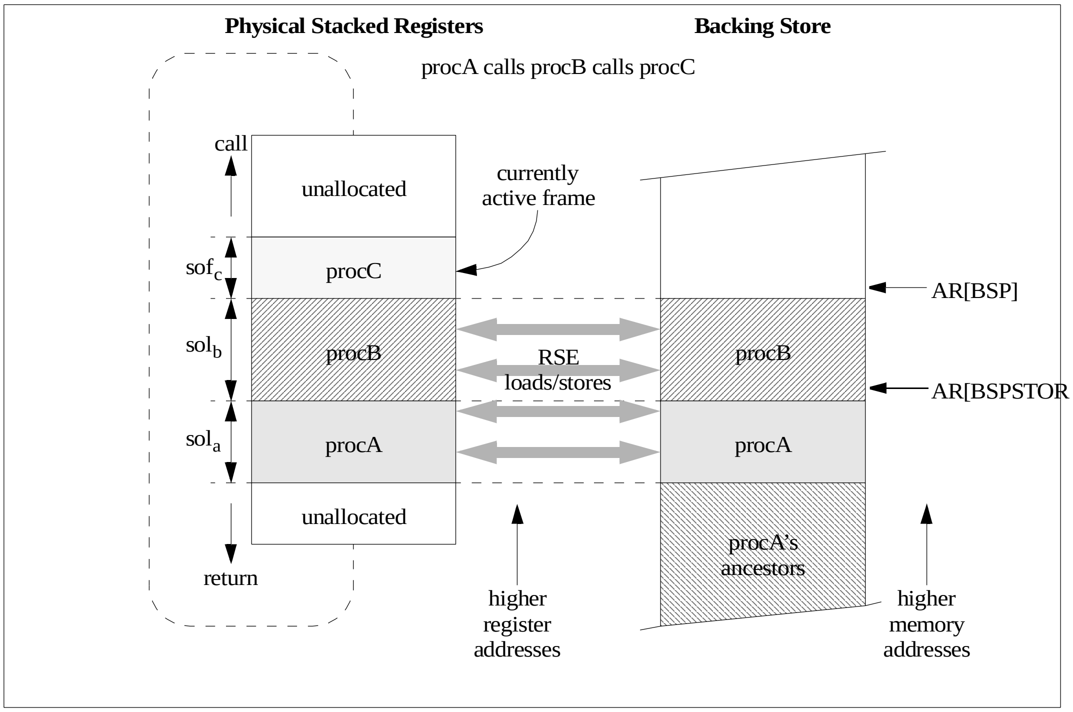
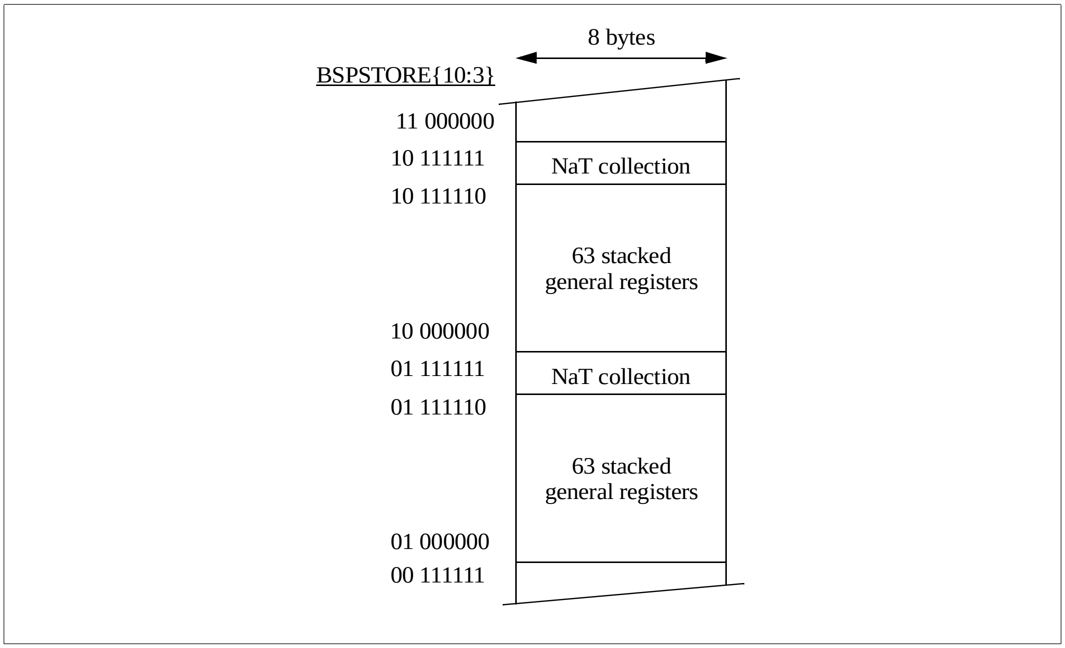


Figure 6-2. Backing Store Memory Format



The RSE operates concurrently and asynchronously with respect to instruction execution by taking advantage of unused memory bandwidth to dynamically perform register spill and fill operations. The algorithm employed by the RSE to determine whether and when to spill/fill is implementation dependent. Software can not depend on the spill/fill algorithm. To ensure that the processor and RSE activities do not interfere with each other, software should not access stacked registers outside of the current stack frame. The architecture guarantees register stack integrity by faulting on writes to out-of-frame registers. Reads from out-of-frame registers may interact with RSE operations and return undefined data values. However, out-of-frame reads are required to propagate NaT bits.

The operation of the RSE is controlled by the Register Stack Configuration (RSC) application register. Activity between the processor and the RSE is synchronized only when `alloc`, `flushrs`, `loadrs`, `br.ret`, or `rfi` instructions actually require registers to be spilled or filled, or when software explicitly requests RSE synchronization by executing a mov to/from RSC, BSPSTORE or RNAT application register instruction.

6.2 RSE Internal State

Table 6-1 describes architectural state that is maintained by the register stack engine. The RSE internal state elements described here are not directly exposed to the programmer as architecturally visible registers. As a consequence, RSE internal state does not need to be preserved across context switches or interruptions. Instead, it is modified as the side-effect of register stack-related instructions. To describe the effects of these instructions a complete definition of the RSE internal state is essential. To distinguish them from architecturally visible resources, all RSE internal state elements are prefixed with “RSE”. Other RSE related resources are architecturally visible and are exposed to software as application registers: RSC, BSP, BSPSTORE, and RNAT.

Table 6-1. RSE Internal State

Name	Description	Corresponds to:
RSE.N_STACKED_PHYS	Number of Stacked Physical registers: Implementation dependent size of the stacked physical register file.	
RSE.BOF	Bottom-of-frame register number: Physical register number of GR32.	AR[BSP]
RSE.StoreReg	RSE Store Register number: Physical register number of next register to be stored by RSE.	AR[BSPSTORE]
RSE.LoadReg	RSE Load Register number: Physical register number one greater than the next register to load (modulo the number of stacked physical registers).	RSE.BspLoad
RSE.BspLoad	Backing Store Pointer for memory loads: 64-bit Backing Store Address 8 bytes greater than the next address to be loaded by the RSE.	RSE.BspLoad
RSE.RNATBitIndex	RSE NaT Collection Bit Index: 6-bit wide RNAT Collection Bit Index (defines which RNAT collection bit gets updated)	AR[BSPSTORE]{8:3}
RSE.CFLE	RSE Current FrameLoad Enable: Control bit that permits the RSE to load registers in the current frame after a `br.ret` or `rfi`.	
RSE.ndirty	Number of dirty registers on the register stack	
RSE.ndirty_words	Number of dirty words on the register stack plus corresponding number of NaT collection registers	AR[BSP] - AR[BSPSTORE]

6.3 Register Stack Partitions

The processor's physical register file provides at least 96 stacked registers. The actual number of stacked registers (RSE.N_STACKED_PHYS) is implementation dependent and must be an even multiple of 16. Figure 6-3 illustrates the circular nature of the physical register file, and shows the correspondence of the registers to the backing store. Figure 6-3 also shows the four partitions of the stacked register file:

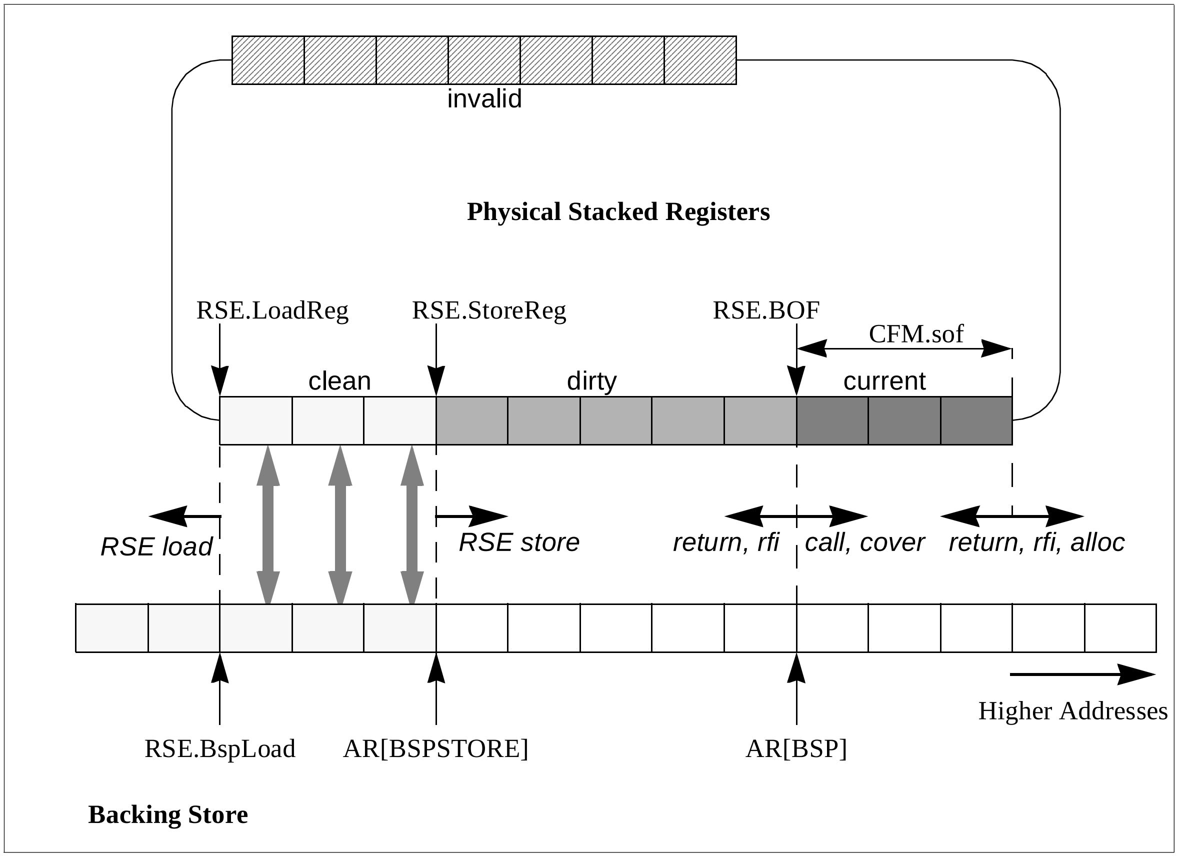
Clean partition (lightly-shaded): registers that contain values from parent procedure frames. The registers in this partition have been successfully spilled to the backing store by the RSE and their contents have not been modified since they were written to the backing store.

Dirty partition (medium-shaded): registers that contain values from parent procedure frames. The registers in this partition have not yet been spilled to the backing store by the RSE. The number of registers contained in the dirty partition (distance between RSE.StoreReg and RSE.BOF) is referred to as RSE.ndirty.

Current frame (shaded dark): stacked registers allocated for computation. The position of the current frame in the physical stacked register file is defined by the Bottom-of-frame register (RSE.BOF). The number of registers in the current frame is defined by the size of frame field in the current frame marker (CFM.sof).

Invalid partition (diagonally striped): registers outside the current frame that do not contain values from parent procedure frames. They are immediately available for allocation into the current frame or for RSE load operations.

Figure 6-3. Four Partitions of the Register Stack



The boundaries between the four register stack partitions are defined by the current frame marker (CFM) and three physical register numbers: a load, store and bottom-of-frame register number. As described in Table 6-1 each of these physical register numbers has a corresponding 64-bit backing store memory address pointer. (For example, AR[BSP] always contains the address where GR[32] of the current frame will be stored.)

Figure 6-3 also shows the effects of various instructions on the partition boundaries. RSE loads use invalid registers. RSE stores use dirty registers. Eager RSE loads and stores grow the clean partition. A `br.call`, `brl.call`, or `cover` instruction can increase the bottom-of-frame pointer (RSE.BOF) which moves registers from the current frame to the dirty partition. An `alloc` may shrink or grow the current frame by updating CFM.sof. A `br.ret` or `rfi` instruction may shrink or grow the current frame by updating both the bottom-of-frame pointer (RSE.BOF) and CFM.sof.

6.4 RSE Operation

The register stack backing store is organized as a stack in memory that grows from lower addresses towards higher addresses. The top of the backing store stack is defined by the Backing Store Pointer (BSP) application register, which points to the first memory location reserved for the current frame. The RSE load and store activities take place at lower addresses, defined relative to BSP by the sizes of the clean and dirty partitions. Although the stack is conceptually infinite in both directions, the effective base of the stack is expected to be the first memory location of the first page allocated to the backing store.

To allow the highest possible degree of concurrent execution, the processor and the RSE operate independently of each other during normal program execution. The RSE distinguishes between **mandatory** and **eager** load/store operations. Mandatory load/store operations occur as the result of `alloc`, `flushrs`, `loadrs`, `br.ret` or `rfi` instructions. Eager operations occur when the RSE is speculatively working ahead of program execution, and it is not known whether this register spill/ fill is actually required by the program.

When the RSE works in the background, it issues eager RSE spill and fill operations to extend the size of the clean partition in both directions—by decreasing the RSE load pointer and loading values from the backing store into invalid registers (eager RSE load), and by saving dirty registers to the backing store and increasing the RSE store pointer (eager RSE store). Allocation of a sufficiently large frame (using `alloc`) or execution of a `flushrs` instruction may cause the RSE to suspend program execution and issue mandatory RSE stores until the required number of registers have been spilled to the backing store. Similarly a `br.ret` or `rfi` back to a sufficiently large frame or execution of a `loadrs` instruction may cause the RSE to suspend program execution and issue mandatory RSE loads until the required number of registers have been restored from the backing store. The RSE only operates in the foreground and suspends program execution whenever forward progress of the program actually requires registers to be spilled or filled.

Table 6-2 describes the RSE operation instructions and state modifications.

Table 6-2. RSE Operation Instructions and State Modification

Affected State	Instruction			
	`alloc` r_1=`ar.pfs,`*i*`,`*l*`,`*o*`,`*r*[a]	`br.call`[a], `brl.call`[a]	`br.ret`[a]	`rfi` **when CR[IFS].v = 1**
AR[BSP]{63:3}	unchanged	AR[BSP]{63:3} + CFM.sol + (AR[BSP]{8:3} + CFM.sol)/63	AR[BSP]{63:3} – AR[PFS].pfm.sol – (62-AR[BSP]{8:3}+ AR[PFS].pfm.sol)/63	AR[BSP]{63:3} – CR[IFS].ifm.sof – (62-AR[BSP]{8:3}+ CR[IFS].ifm.sof)/63
AR[PFS]	unchanged	AR[PFS].pfm = CFM AR[PFS].pec = AR[EC] AR[PFS].ppl = PSR.cpl	unchanged	unchanged
GR[r_1]	AR[PFS]	N/A	N/A	N/A
CFM	CFM.sof = *i*+*l*+*o* CFM.sol = *i*+*l* CFM.sor = *r* >> 3	CFM.sof -= CFM.sol CFM.sol = 0 CFM.sor = 0 CFM.rrb.gr = 0 CFM.rrb.fr = 0 CFM.rrb.pr = 0	AR[PFS].pfm or [b] CFM.sof = 0 CFM.sol = 0 CFM.sor = 0 CFM.rrb.gr = 0 CFM.rrb.fr = 0 CFM.rrb.pr = 0	CR[IFS].ifm

a. These instructions have undefined behavior with an incomplete frame. See “RSE Behavior with an Incomplete Register Frame” on page 2:129.

b. Normal `br.ret` instructions restore CFM with AR[PFS].pfm. However, if a bad PFS value is read by the `br.ret` instruction, all CFM fields are set to zero. See “Bad PFS used by Branch Return” on page 2:126.

6.5 RSE Control

The RSE can be controlled at all privilege levels by means of three instructions (`cover`, `flushrs`, and `loadrs`) and by accessing four application registers (mov to/from RSC, BSP, BSPSTORE and RNAT). This section first presents each of the RSE application registers, and then discusses the three RSE control instructions.

6.5.1 Register Stack Configuration Register

The layout of the Register Stack Configuration application register (RSC) is defined in Section 3.1.8.2. This section describes the semantics of the mode, the privilege level and the byte order fields of the RSC. The loadrs field is described as part of the `loadrs` instruction in Section 6.5.4.

RSE Mode: Two mode bits in the RSC register determine when the RSE generates register spill or fill operations. When both mode bits are zero (enforced lazy mode) the RSE issues only mandatory loads and stores (when an `alloc`, `br.ret`, `flushrs` or `rfi` instruction requires registers to be spilled or filled). Bit 0 of the RSC.mode field enables eager RSE stores and bit 1 enables eager RSE loads. Table 6-3 defines all four possible RSE modes. Please see the processor specific documentation for further information on the RSE modes implemented by the Itanium processor.

Table 6-3. RSE Modes (RSC.mode)

Mode	RSE Loads	RSE Stores	RSC.mode
Enforced Lazy	Mandatory only	Mandatory only	00
Store Intensive	Mandatory only	Eager and Mandatory	01
Load Intensive	Eager and Mandatory	Mandatory only	10
Eager	Eager and Mandatory	Eager and Mandatory	11

The algorithm that decides whether and when to speculatively perform eager register spill or fill operations is implementation dependent. Software may not make any assumptions about the RSE load/store behavior when the RSC.mode is non-zero. Furthermore, access to the BSPSTORE and RNAT application registers and the execution of the `loadrs` instructions require RSC.mode to be zero (enforced lazy mode). If `loadrs`, move to/from BSPSTORE or move to/from RNAT are executed when RSC.mode is non-zero an Illegal operation fault is raised. Eager spill/fill of the RNAT register to/from the backing store is only permitted if the RSE is in store/load intensive or eager mode. In enforced lazy mode, the RSE may spill/fill the RNAT register only if a subsequent mandatory register spill/fill is required.

RSE Privilege Level: When address translation is enabled (PSR.rt is one), the RSE operates at a privilege level defined by two privilege level bits in the Register Stack Configuration register (RSC.pl). All privilege level checks for RSE virtual accesses are performed using the privilege level in RSC.pl. When the RSC is written, the privilege level bits are clipped to the current privilege level of the process, i.e., the numerical maximum of the current privilege level and the privilege level in the source register is written to RSC.pl.

Protection is also checked based on the current entries in the data TLB. The RSE always remains coherent with respect to the data TLB. If a translation that is being used by the RSE is changed or purged, the RSE will immediately begin using the new translation or suffer a TLB miss. Only mandatory loads and stores can cause RSE memory related faults. Details on RSE fault delivery are described in "RSE Interruptions" Although eager RSE loads and stores do not cause interruptions they can, under certain conditions, cause a VHPT walk and TLB insert. Details on when RSE loads and stores can cause a VHPT walk are described in "VHPT Environment" on page 2:56.

The RSE expects its backing store to be mapped to cacheable speculative memory. If RSE spill/fill transactions are performed to non-speculative memory that may contain I/O devices, system behavior is unpredictable.

RSE Byte Order: Because the RSE runs asynchronously with the processor, it may be running on behalf of a context with a different byte order from the current one. Consequently, the RSE defines its own byte ordering bit: RSC.be. When RSC.be is zero, registers are stored in little-endian byte order (least significant bytes to lower addresses). When RSC.be is one, registers are stored in big-endian byte order (most significant bytes to lower addresses). RSC.be also determines the byte order of NaT collections spilled/filled by the RSE. RSC.be may be written by code at any privilege level. Changes to RSC.be should only be made by software when RSC.mode is zero. Failure to do so results in undefined backing store contents.

6.5.2 Register Stack NaT Collection Register

As described in Section 6.1, the RSE is responsible for saving and restoring NaT bits associated with the stacked registers to and from the backing store. The RSE writes its NaT collection register (the RNAT application register) to the backing store whenever BSPSTORE{8:3} = 0x3F (1 NaT collection for every 63 registers). The RNAT acts as a temporary holding area for up to 63 unsaved NaT bits. The RSE NaT collection bit index (RSE.RNATBitIndex) determines which bit of the RNAT register receives the NaT bit of a spilled register as the result of an RSE store. The six-bit wide RSE.RNATBitIndex is always equal to BSPSTORE{8:3}. As a result, RNAT{*x*} corresponds to the register saved at

```
concatenate(BSPSTORE{63:9},x{5:0},0{2:0}).
```

The RSE never saves partial NaT collections to the backing store, so software must save and restore the RNAT application register when switching the backing store pointer. RSE.RNATBitIndex determines which RNAT bits are valid. Bits RNAT{RSE.RNATBitIndex:0} contain defined values, and bits RNAT{62:RSE.RNATBitIndex+1} contain undefined values. Bit 63 of the RNAT application register always reads as zero. Writes to bit 63 of the RNAT application register are ignored. The execution of RSE control instructions `mov` to BSPSTORE and `loadrs` as well as an RSE spill of the RNAT register cause the contents of the RNAT register to become undefined. The RNAT application register can only be accessed when RSC.mode is zero. If RSC.mode is non-zero, accessing the RNAT application register results in an Illegal Operation fault.

6.5.3 Backing Store Pointer Application Registers

The RSE defines two Backing Store Pointer application registers: BSPSTORE and BSP. Since the RSE backing store pointers are always 8-byte aligned, bits {2:0} of the backing store pointers always read as zero. When writing the BSPSTORE application register, bits {2:0} in the presented address are ignored.

The RSE Backing Store Pointer for memory stores (BSPSTORE) is a 64-bit application register that provides the main interface to the three RSE backing store memory pointers: BSP, BSPSTORE and RSE.BspLoad. The BSPSTORE application register can only be accessed when RSC.mode is zero. If RSC.mode is non-zero, accessing BSPSTORE results in an Illegal Operation fault.

Reading BSPSTORE (`mov` from BSPSTORE application register) returns the address of the next RSE store.

Writing BSPSTORE (`mov` to BSPSTORE application register) has side-effects on all three RSE pointers and the NaT collection process. The operation is defined as follows: the BSPSTORE and RSE.BspLoad pointers are both set to the address presented, which forces the size of the clean partition to zero. Writes to the BSPSTORE application register do not change the size of the dirty partition: the BSP pointer is set to the address presented plus the size of the dirty partition plus the size of any intervening NaT collections. The dirty partition is preserved to allow software to change the backing store pointer without having to flush the register stack. Writing BSPSTORE causes the contents of the RNAT register to become undefined. Therefore software must preserve the contents of RNAT prior to writing BSPSTORE. After writing to BSPSTORE, the NaT collection bit index (RSE.RNATBitIndex) is set to bits {8:3} of the presented address. If an unimplemented address in BSPSTORE is used by a mandatory RSE spill or fill, an Unimplemented Data Address fault is raised.

The RSE Backing Store Pointer (BSP) is a 64-bit read-only application register. Writing BSP (`mov` to BSP application register) results in an Illegal Operation fault. Reads from BSP (`mov` from BSP application register) return the address of the top of the register stack in memory. This location is the backing store address to which the current GR32 would be written. Reading BSP does not have any side-effect on any of the internal RSE pointers or the NaT collection process. Therefore, BSP can be read regardless of the RSE mode, i.e., even when RSC.mode is non-zero. Since BSP is determined by BSPSTORE and the size of the dirty partition, it is possible for BSPSTORE to contain an implemented address and for BSP to contain an unimplemented address. BSP reads always return a full 64-bit (possibly unimplemented) address; only a subsequent data memory reference with an unimplemented address will cause an Unimplemented Data Address fault.

Table 6-4 summarizes the effects of the three instructions that access the backing store pointer application registers.

Table 6-4. Backing Store Pointer Application Registers

Affected State	Instruction		
	Read BSP `mov` r_1`=AR[BSP]`	**Read BSPSTORE** `mov` r_1`=AR[BSPSTORE]`	**Write BSPSTORE**[a] `mov AR[BSPSTORE]=r2`
GR[r_1]	AR[BSP]	AR[BSPSTORE]	N/A
AR[BSP]{63:3}	Unchanged	Unchanged	(GR[r_2]{63:3} + RSE.ndirty) + ((GR[r_2]{8:3} + RSE.ndirty)/63)
AR[BSPSTORE]{63:3}	Unchanged	Unchanged	GR[r_2]{63:3}
RSE.BspLoad {63:3}	Unchanged	Unchanged	GR[r_2]{63:3}
AR[RNAT]	Unchanged	Unchanged	UNDEFINED
RSE.RNATBitIndex	Unchanged	Unchanged	GR[r_2]{8:3}

a. Writing to AR[BSPSTORE] has undefined behavior with an incomplete frame. See "RSE Behavior with an Incomplete Register Frame" on page 2:129.

6.5.4 RSE Control Instructions

This section describes the RSE control instructions: `cover`, `flushrs` and `loadrs`. The effects of the three RSE control instructions on the RSE state are summarized in Table 6-5.

The `cover` instruction adds all registers in the current frame to the dirty partition, and allocates a zero-size current frame. As a result AR[BSP] is updated. `cover` clears the register rename base fields in the current frame marker CFM. If PSR.ic is zero, the original value of CFM is copied into CR[IFS].ifm and CR[IFS].v is set to one. The `cover` instruction must be specified as the last instruction in a bundle group otherwise an Illegal Operation fault is taken.

The `flushrs` instruction spills all dirty registers to the backing store. When it completes, RSE.ndirty is defined to be zero, and BSPSTORE equals BSP. Since `flushrs` may cause RSE stores, the RNAT application register is updated. A `flushrs` instruction must be the first instruction in an instruction group otherwise the results are undefined.

The `loadrs` instruction ensures that a specified portion of the backing store below the current BSP is present in the physical stacked registers. The size of the backing store section is specified in the `loadrs` field of the RSC application register (AR[RSC].loadrs). After loadrs completes, all registers and NaT collections between the current BSP and the tear-point (BSP-(RSC.loadrs{13:3} << 3)), and no more than that, are guaranteed to be present and marked as dirty in the stacked physical registers. When `loadrs` completes BSPSTORE and RSE.BspLoad are defined to be equal to the backing store tear-point address. All other physical stacked registers are marked invalid.

- If the tear-point specifies an address below RSE.BspLoad, the RSE issues mandatory loads to restore registers and NaT collections. All registers between the current BSP and the tear-point are marked dirty.

- If the RSE has already loaded registers beyond the tear-point when the `loadrs` instruction executes, the RSE marks clean registers below the tear-point as invalid and marks clean registers above the tear-point as dirty.
- If the tear-point specifies an address greater than BSPSTORE, the RSE marks clean and dirty registers below the tear-point as invalid (in this case dirty registers are lost).

Table 6-5. RSE Control Instructions

Affected State	Instruction		
	cover	flushrs[a]	loadrs[a]
AR[BSP]{63:3}	AR[BSP]{63:3}+ CFM.sof + (AR[BSP]{8:3} + CFM.sof)/63	Unchanged	Unchanged
AR[BSPSTORE]{63:3}	Unchanged	AR[BSP]{63:3}	AR[BSP]{63:3} – AR[RSC].loadrs{13:3}
RSE.BspLoad{63:3}	Unchanged	Model specific[b]	AR[BSP]{63:3} – AR[RSC].loadrs{13:3}
AR[RNAT]	Unchanged	Updated	UNDEFINED
RSE.RNATBitIndex	Unchanged	AR[BSPSTORE]{8:3}	AR[BSPSTORE]{8:3}

CR[IFS]	if (PSR.ic == 0) {
	CR[IFS].ifm = CFM
	CR[IFS].v = 1}
		Unchanged	Unchanged

CFM	CFM.sof = 0
	CFM.sol = 0
	CFM.sor = 0
	CFM.rrb.gr = 0
	CFM.rrb.fr = 0
	CFM.rrb.pr = 0
		Unchanged	Unchanged

a. These instructions have undefined behavior with an incomplete frame. See "RSE Behavior with an Incomplete Register Frame" on page 2:129.

b. In general, eager RSE implementations will preserve RSE.BspLoad during a `flushrs`. Lazy RSE implementations may set RSE.BspLoad to AR[BSPSTORE] after `flushrs` completes or faults.

By specifying a zero RSC.loadrs value `loadrs` can be used to invalidate all stacked registers outside the current frame. `loadrs` causes the contents of the RNAT register to become undefined. The NaT collection index is set to bits {8:3} of the new BSPSTORE. A `loadrs` instruction must be the first instruction in an instruction group otherwise the results are undefined. The following conditions cause `loadrs` to raise an Illegal Operation fault:

- If RSC.mode is non-zero.
- If both CFM.sof and RSC.loadrs are non-zero.
- If RSC.loadrs specifies more words to be loaded than will fit in the stacked physical register file (RSE.N_STACKED_PHYS).

6.5.5 Bad PFS used by Branch Return

On a `br.ret`, if the PFS application register defines an output area which is larger than the number of implemented stacked registers minus the size of dirty partition ((AR[PFS].sof – AR[PFS].sol) > (RSE.N_STACKED_PHYS – RSE.ndirty)), the return will not restore CFM with AR[PFS].pfm (normal behavior); instead, the return sets all fields in the CFM (of the procedure being returned to) to zero.

Typical procedure call and return sequences that preserve PFS values and that do not use cover or loadrs instructions will not encounter this situation.

The RSE will detect the above condition on a br.ret, and update its state as follows:

- The register rename base (RSE.BOF), AR[BSP], and AR[BSPSTORE] are updated as required by the return.
- The CFM (after the return) is forced to zero; i.e., all CFM fields (including CFM.sof and CFM.sol) are set to zero.
- The registers from the returned-from frame and the preserved registers from the returned-to frame are added to the invalid partition of the register stack.
- The dirty partition of the register stack is shrunk by AR[PFS].pfm.sol.
- The clean partition of the register stack remains unchanged. RSE.BspLoad and RSE.LoadReg remain unchanged.
- No other indication is given to software.

Since the size of the current frame is set to zero, the contents of some (possibly all) stacked GRs may be overwritten by subsequent eager RSE operations or by subsequent instructions allocating a new stack frame and then targeting a stacked GR. Therefore, explicit register stack management sequences that manipulate PFS, use the cover instruction, or use the loadrs instruction must avoid this situation by executing one of the two following code sequences prior to a br.ret:

- Use a flushrs instruction prior to the br.ret. This preserves all dirty registers to memory, and sets RSE.ndirty to zero, which avoids the condition.
- Use a loadrs instruction with an AR[RSC].loadrs value in the following range:

  AR[RSC].loadrs <= 8*(ndirty_max + ((62 – AR[BSP]{8:3} + ndirty_max) / 63)),
  where ndirty_max = (RSE.N_STACKED_PHYS – (AR[PFS].sof – AR[PFS].sol))

This adjusts the size of the dirty partition appropriately to avoid the condition. A loadrs with RSC.loadrs=0 works on all processor models, regardless of the number of implemented stacked physical registers. Note that loadrs may cause registers in the dirty partition to be lost.

6.6 RSE Interruptions

Although the RSE runs asynchronously to processor execution, RSE related interruptions are delivered synchronously with the instruction stream. These RSE interruptions are a direct consequence of register stack-related instructions such as: alloc, br.ret, rfi, flushrs, loadrs, or mov to/from BSP, BSPSTORE, RSC, PFS, IFS, or RNAT. Register spills and fills that are executed by the RSE in the background (eager RSE loads or stores) do not raise interruptions. If a faulting/trapping register spill or fill operation is required for software to make forward progress (mandatory RSE load or store) then the RSE will raise an interruption.

Mandatory RSE stores occur in the context of alloc and flushrs instructions only. Any faults raised by these instructions are delivered on the issuing instruction. Faults raised by mandatory RSE loads caused by a loadrs are delivered on the issuing instruction. Mandatory RSE loads which fault while restoring the frame for a br.ret or rfi deliver the fault on the target instruction, and the ISR.ir (incomplete register frame) bit is set. When a mandatory RSE load faults, AR[BSPSTORE] points to a backing store location above the faulting address reported in CR[IFA]. This allows handlers that service RSE load faults to use the backing store switch routine described in "Switch from Interrupted Context" on page 2:131.

The `br.ret` and the `rfi` instructions set the RSE Current Frame Load Enable bit (RSE.CFLE) to one if the register stack frame being returned to is not entirely contained in the stacked register file. This enables the RSE to restore registers for the current frame of the target instruction. When RSE.CFLE is set, instruction execution is stalled until the RSE has completely restored the current frame or an interruption occurs. This is the only time that the RSE issues any memory traffic for the current frame. Interruption delivery clears RSE.CFLE which allows an interruption handler to execute in the presence of an incomplete frame (e.g., to handle the fault raised by the mandatory RSE load). The RSE.CFLE bit is RSE internal state and is not architecturally visible.

Table 6-6 summarizes RSE raised interruptions.

Table 6-6. RSE Interruption Summary

Instruction	Interruption	Description
`alloc`	Illegal Operation fault	Malformed `alloc` immediate.
`alloc`	Reserved Register/Field fault	`alloc` instruction which attempted to change the size of the rotating region when one or more of the RRB values in CFM were non-zero.

`alloc,`
`flushrs,`
`loadrs`
	Unimplemented Data Address fault	AR[BSPSTORE] contains an unimplemented address.

	Data Nested TLB fault	
	Alternate Data TLB fault	
	VHPT Data fault	
	Data TLB fault	
	Data Page Not Present fault	
	Data NaT Page Consumption fault	AR[BSPSTORE] pointed to a NaTVal data page.
	Data Key Miss fault	
	Data Key Permission fault	
	Data Access Rights fault	
	Data Dirty Bit fault	
	Data Access Bit fault	
	Data Debug fault	

`br.call,`
`brl.call`
	No RSE related interruptions	

`br.ret`	No RSE load related faults	RSE load related faults are delivered on target instruction.
`rfi`	No RSE related interruptions	RSE load related faults are delivered on target instruction.
Target of `br.ret` or `rfi`	IR Unimplemented Data Address fault	Mandatory RSE load targeted an unimplemented address.
	IR Data Nested TLB fault	`br.ret` with PSR.ic = 0 or `rfi` executed when IPSR.ic = 0.
	IR Alternate Data TLB fault	
	IR VHPT Data TLB fault	
	IR Data TLB fault	
	IR Data Page Not Present fault	
	IR Data NaT Page Consumption fault	RSE.BspLoad pointed at a NaTPage.
	IR Data Key Miss fault	
	IR Data Key Permission fault	
	IR Data Access Rights fault	
	IR Data Access Bit fault	
	IR Data Debug fault	

6.7 RSE Behavior on Interruptions

When the processor raises an interruption, the current register stack frame remains unchanged. If PSR.ic is one, the valid bit in the Interruption Function State register (IFS.v) is cleared. When the IFS.v bit is clear, the contents of the interruption frame marker field (IFS.ifm) are undefined.

While an interruption handler is running and the RSE is in store/load intensive or eager mode, the RSE continues spilling/filling registers to/from the backing store on behalf of the interrupted context as long as the registers are not part of the current frame as defined by CFM.

A sequence of mandatory RSE loads or stores (from `alloc`, `br.ret`, `flushrs`, `loadrs` and `rfi`) can be interrupted by an external interrupt.

When PSR.ic is 0, faults taken on mandatory RSE operations may not be recoverable.

6.8 RSE Behavior with an Incomplete Register Frame

The current register frame is considered **incomplete** when one of the mandatory RSE loads after a br.ret or a rfi faults, leaving BSPSTORE pointing to a location above BSP (i.e., RSE.ndirty_words is negative). The frame becomes complete when RSE.ndirty_words becomes non-negative, either by executing a cover instruction, or by handling the fault and completing the original sequence of mandatory RSE loads.

When the current frame is incomplete the following instructions have undefined behavior: `alloc`, `br.call`, `brl.call`, `br.ret`, `flushrs`, `loadrs`, and move to BSPSTORE. Software must guarantee that the current frame is complete before executing these instructions.

6.9 RSE and ALAT Interaction

The ALAT (see "Data Speculation" on page 2:57) uses physical register addresses to track advanced loads. RSE.BOF may only change as the result of a `br.call` (by CFM.sol), `cover` (by CFM.sof), `br.ret` (by AR[PFM].sol) or `rfi` (by CR[IFS].ifm.sof when CR[IFS].v =1). This ensures, for ALAT invalidation purposes, that hardware does not update virtual to physical register address mapping, unless explicitly instructed to do so by software.

When software performs backing store switches that could cause program values to be placed in different physical registers, then the ALAT must be explicitly invalidated with the `invala` instruction. Typically this happens as part of a process or thread context switch, longjmp or call stack unwind, when software re-writes AR[BSPSTORE], but cannot guarantee that RSE.BOF was preserved.

A stacked register is said to be **deallocated** when an `alloc`, `br.ret`, or `rfi` instruction changes the top of the current frame such that the register is no longer part of the current frame. Once a stacked register is deallocated, its value, its corresponding NaT bit, and its ALAT state are undefined. If that register is subsequently made part of the current frame again (either via another `alloc` instruction, or via a `br.ret` or `rfi` to a previous frame that contained that register), the value stored in the register, the NaT bit for the register, and the corresponding ALAT entry for the register remain undefined.

RSE stores do not invalidate ALAT entries. Therefore, software cannot use the ALAT to trace RSE stores to the backing store.

Note: While an implementation is allowed to remove entries from the ALAT at any time, performance considerations strongly encourage not invalidating ALAT entries due to RSE stores.

6.10 Backing Store Coherence and Memory Ordering

RSE loads and stores are coherent with respect to the processor's data cache at all times. The backing store below BSPSTORE is defined to be consistent with the register stack (the memory image contains consecutive register values and NaT collections). Addresses below BSPSTORE are not modified by the RSE until `br.ret`, `rfi` or a move to BSPSTORE causes BSP to drop below the original BSPSTORE value. The RSE never writes to a memory address greater than or equal to BSP.

In order for software to modify a value in the backing store and guarantee that it be loaded by the RSE, software must first place the RSE into enforced lazy mode (RSC.mode=0), and read BSP and BSPSTORE to determine the location of the RSE store pointer. If the location to be modified lies between BSPSTORE and BSP, software must issue a `flushrs`, update the backing store location in memory, and issue a `loadrs` instruction with the RSC.loadrs set to zero (this invalidates the current contents of the physical stacked registers, except the current frame, which forces the RSE to reload registers from the backing store). If the location to be modified lies below BSPSTORE, unnecessary memory traffic can be avoided as follows: software must read the RNAT application register, update the backing store location in memory, rewrite BSPSTORE with the original value, and then rewrite RNAT.

RSE loads and stores are weakly ordered. The `flushrs` and `loadrs` instructions do not include an implicit memory fence. Turning on and off the RSE does not affect memory ordering. To ensure ordering of RSE loads and stores on a multiprocessor system, software is required to issue explicit memory fence (`mf`) instructions.

6.11 RSE Backing Store Switches

The implementation of system calls, operating system context switches, user-level thread packages, debugging software, and certain types of exception handling (e.g., setjmp/longjmp, structured exception handling and call stack unwinding) require explicit user-level control of the RSE and/or knowledge of the backing store format in memory. Therefore, the RSE and the backing store can be controlled at all privilege levels.

Three RSE backing store switches are described here:

1. Switching from an interrupted context (as part of exception handler or interrupt bubble-up code)
2. Returning to a previously interrupted context

3. Non-preemptive, synchronous backing store switch (covers system calls, user-level thread and operating system context switches)

Failure to follow these sequences may result in undefined RSE and processor behavior.

6.11.1 Switch from Interrupted Context

To switch from the backing store of an interrupted context to a new backing store:

1. Read and save the RSC and PFS application registers.
2. Issue a `cover` instruction for the interrupted frame.
3. Read and save the IFS control register.
4. Place RSE in enforced lazy mode by clearing both RSC.mode bits.
5. Read and save the BSPSTORE and RNAT application registers.
6. Write BSPSTORE with the new backing store address.
7. Read and save the new BSP to calculate the number of dirty registers.
8. Select the desired RSE setting (mode, privilege level and byte order).

6.11.2 Return to Interrupted Context

To return to the backing store of an interrupted context:

1. Allocate a zero-sized frame.
2. Subtract the BSPSTORE value written in step 6 of Section 6.11.1 from the BSP value read in step 7 of Section 6.11.1, and deposit the difference into RSC.loadrs along with a zero into RSC.mode (to place the RSE into enforced lazy mode).
3. Issue a `loadrs` instruction to insure that any registers from the interrupted context which were saved on the new stack have been loaded into the stacked registers.
4. Restore BSPSTORE from the interrupted context (saved in step 5 of Section 6.11.1).
5. Restore RNAT from the interrupted context (saved in step 5 of Section 6.11.1).
6. Restore PFS and IFS from the interrupted context (saved in steps 1 and 3 of Section 6.11.1).
7. Restore RSC from the interrupted context (saved in step 1 of Section 6.11.1). This restores the setting of the RSE mode bits as well as privilege level and byte order.
8. Issue an `rfi` instruction (IFS.ifm will become CFM).

6.11.3 Synchronous Backing Store Switch

A non-preemptive, synchronous backing store switch at any privilege level can be accomplished as follows:

1. Read and save the RSC, BSP and PFS application registers.
2. Issue a `flushrs` instruction to flush the dirty registers to the backing store.
3. Place RSE in enforced lazy mode by clearing both RSC.mode bits.

4. Read and save the RNAT application register.
5. Invalidate the ALAT using the `invala` instruction when switching from code that does not restore RSE.BOF to its original setting. A different RSE.BOF will cause program values in the new context to be placed in different physical registers. See "RSE and ALAT Interaction" on page 2:129 for details.
6. Write the new context's BSPSTORE (was BSP after `flushrs` when switching out).
7. Write the new context's PFS and RNAT.
8. Write the new context's RSC which will set the RSE mode, privilege level and byte order.

6.12 RSE Initialization

At processor reset the RSE is defined to be in enforced lazy mode, i.e., the RSC.mode bits are both zero. The RSE privilege level (RSC.pl) is defined to be zero. RSE.BOF points to physical register 32. The values of AR[PFS].pfm and CR[IFS].ifm are undefined. The current frame marker (CFM) is set as follows: sof=96, sol=0, sor=0, rrb.gr=0, rrb.fr=0, and rrb.pr=0. This gives the processor access to 96 stacked registers.

The RSE performs no spill/fill operations until either an `alloc`, `br.ret`, `rfi`, `flushrs` or `loadrs` require a mandatory RSE operation, or software explicitly enables eager RSE operations. Software must provide the RSE with a valid backing store address in the BSPSTORE application register prior to causing any RSE spill/fill operations. Failure to initialize BSPSTORE results in undefined behavior.

Debugging and Performance Monitoring 7

Processors based on the Itanium architecture provide comprehensive debugging and performance monitoring facilities for both IA-32 and Itanium instructions. This chapter describes the debug registers, performance monitoring registers and their programming models. The debugging facilities include several data and instruction break point registers, single step trap, breakpoint instruction fault, taken branch trap, lower privilege transfer trap, instruction and data debug faults. The performance monitoring facilities include two sets of registers to configure and collect various performance-related statistics.

7.1 Debugging

Several Data Breakpoint Registers (DBR) and Instruction Breakpoint Registers (IBR) are defined to hold address breakpoint values for data and instruction references. In addition the following debugging facilities are supported:

- **Single Step trap** – When PSR.ss is 1, successful execution of each Itanium instruction results in a Single Step trap. When PSR.ss is 1 or EFLAG.tf is 1, successful execution of each IA-32 instruction results in an IA_32_Exception(Debug) single step trap. After the trap, IIP and IPSR.ri point to the next instruction to be executed. IIPA and ISR.ei point to the trapped instruction. See "Single Stepping" for complete single stepping behavior.
- **Break Instruction fault** – execution of a `break` instruction results in a Break Instruction fault. IIM is loaded with the immediate operand from the instruction. IIM values are defined by software convention. `break` can be used for profiling, debugging and entry into the operating system (although Enter Privileged Code (`epc`) is recommended since it has lower overhead). Execution of the IA-32 INT 3 (break) instruction results in a IA_32_Exception(Debug) trap.
- **Taken Branch trap** – When PSR.tb is 1, a Taken Branch trap occurs on every taken Itanium branch instruction. When PSR.tb is 1, a IA_32_Exception(Debug) taken branch trap occurs on every taken IA-32 branch instruction (CALL, Jcc, JMP, RET, LOOP). This trap is useful for debugging and profiling. After the trap, IIP and IPSR.ri point to the branch target instruction and IIPA and ISR.ei point to the trapping branch instruction.
- **Lower Privilege Transfer trap** – When PSR.lp bit is 1, and an Itanium branch demotes the privilege level (numerically higher), a Lower Privilege Transfer trap occurs. This trap allows for auditing of privilege demotions, for example to remove permissions which were granted to higher privilege code. After the trap, IIP and IPSR.ri point to the branch target and IIPA and ISR.ei point to the trapping branch instruction. IA-32 instructions can not raise this trap.
- **Instruction Debug faults** – When PSR.db is 1, any Itanium instruction memory reference that matches the parameters specified by the IBR registers results in an Instruction Debug fault. Instruction Debug faults are reported even if Itanium instructions are nullified due to a false predicate. If PSR.id is 1, Itanium Instruction Debug faults are disabled for one instruction. The successful execution of an Itanium instruction clears PSR.id. When PSR.db is 1, any IA-32 instruction memory reference that matches the parameters specified by the IBR registers results in an IA_32_Exception(Debug) fault. If PSR.id is 1 or EFLAG.rf is 1, IA-32 Instruction

Debug faults are disabled for one instruction. The successful execution of an IA-32 instruction clears the PSR.id and EFLAG.rf bits.

- **Data Debug faults** – When PSR.db is 1, any Itanium data memory reference that matches the parameters specified by the DBR registers results in a Data Debug fault. Data Debug faults are only reported if the qualifying predicate is true. Data Debug faults can be deferred on speculative loads by setting DCR.dd to 1. If PSR.dd is 1, Data Debug faults are disabled for one instruction or one mandatory RSE memory reference. When PSR.db is 1, any IA-32 data memory reference that matches the parameters specified by the DBR registers results in a IA_32_Exception(Debug) trap. IA-32 data debug events are traps, not faults as defined for the Itanium instruction set. The reported trap code returns the match status of the first 4 DBR registers that matched during the execution of the IA-32 instruction. See "IA-32 Trap Code" on page 2:189 for trap code details. Zero, one or more DBR registers may be reported as matching.

7.1.1 Data and Instruction Breakpoint Registers

Instruction or data memory addresses that match the Instruction or Data Breakpoint Registers (IBR/DBR) shown in Figure 7-1 and Figure 7-2 and Table 7-1 result in an Instruction or Data Debug fault. IA-32 Instruction or data memory addresses that match the Instruction or Data Breakpoint Registers (IBR/DBR) result in an IA-32_Exception(Debug) fault or trap. Even numbered registers contain breakpoint addresses, odd registers contain breakpoint mask conditions. At least 4 data and 4 instruction register pairs are implemented on all processor models. Implemented registers are contiguous starting with register 0.

Figure 7-1. Data Breakpoint Registers (DBR)

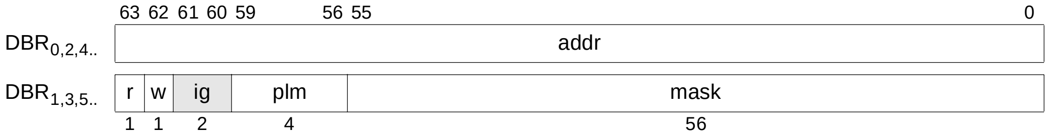
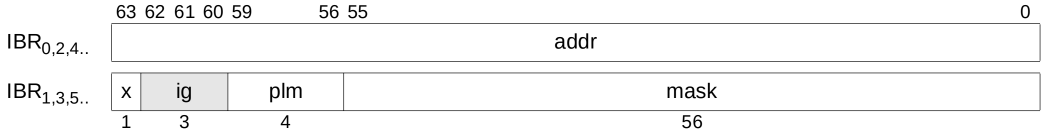


Figure 7-2. Instruction Breakpoint Registers (IBR)



When executing Itanium instructions, instruction and data memory addresses presented for matching are always in the implemented address space. Programming an unimplemented physical address into an IBR/DBR guarantees that physical addresses presented to the IBR/DBR will never match. Similarly, programming an unimplemented virtual address into an IBR/DBR guarantees that virtual addresses presented to the IBR/DBR will never match.

Table 7-1. Debug Breakpoint Register Fields (DBR/IBR)

Field	Bits	Description
addr	63:0	Match Address – 64-bit virtual or physical breakpoint address. Addresses are interpreted as either virtual or physical based on PSR.dt, PSR.it or PSR.rt. Data breakpoint addresses trap on load, store, semaphore, and mandatory RSE memory references. For Intel® Itanium™ instruction set references, IBR.addr{3:0} is ignored in the address match. For IA-32 instruction references, IBR.addr{31:0} are used in the match and IBR.addr{63:32} must be zero to match. All 64 bits are implemented on all processors regardless of the number of implemented address bits.
mask	55:0	Address Mask – determines which address bits in the corresponding address register are compared in determining a breakpoint match. Address bits whose corresponding mask bits are 1, must match for the breakpoint to be signaled, otherwise the address bit is ignored. Address bits{63:56} for which there are no corresponding mask bits are always compared. For IA-32 instruction references, IBR.mask{55:32} are ignored. All 56 bits are implemented on all processors regardless of the number of implemented address bits.
plm	59:56	Privilege Level Mask – enables data breakpoint matching at the specified privilege level. Each bit corresponds to one of the 4 privilege levels, with bit 56 corresponding to privilege level 0, bit 57 with privilege level 1, etc. A value of 1 indicates that the debug match is enabled at that privilege level.
w	62	Write match enable – When DBR.w is 1, any non-nullified mandatory RSE store, IA-32 or Intel® Itanium™ store, semaphore, probe.w.fault or probe.rw.fault to an address matching the corresponding address register causes a breakpoint.
r	63	Read match enable – When DBR.r is 1, any non-nullified IA-32 or Intel® Itanium™ load, mandatory RSE load, semaphore, lfetch.fault, probe.r.fault or probe.rw.fault to an address matching the corresponding address register causes a breakpoint. When DBR.r is 1, a VHPT access that matches the DBR (except those for a `tak` instruction) will cause an Instruction/ Data TLB Miss fault. If DBR.r and DBR.w are both 0, that data breakpoint register is disabled.
x	63	Execute match enable – When IBR.x is 1, execution of an IA-32 instruction or Intel® Itanium™ instruction in a bundle at an address matching the corresponding address register causes a breakpoint. If IBR.x is 0, that instruction breakpoint register is disabled. Instruction breakpoints are reported even if the qualifying predicate is false.
ig	62:60	Ignored

Four privileged instructions, defined in Table 7-2, allow access to the debug registers. Register access is indirect, where the debug register number is determined by the contents of a general register. DBR/IBR registers can only be accessed at privilege level 0, otherwise a Privileged Operation fault is raised.

Table 7-2. Debug Instructions

Mnemonic	Description	Operation	Instr Type	Serialization Required
mov dbr[r_3] = r_2	Move to data breakpoint register	DBR[GR[r_3]] ← GR[r_2]	M	data
mov r_1 = dbr[r_3]	Move from data breakpoint register	GR[r_1] ← DBR[GR[r_3]]	M	none
mov ibr[r_3] = r_2	Move to instruction breakpoint register	IBR[GR[r_3]] ← GR[r_2]	M	inst
mov r_1 = ibr[r_3]	Move from instruction breakpoint register	GR[r_1] ← IBR[GR[r_3]]	M	none

break *imm*	Breakpoint Instruction fault	if (PSR.ic) IIM ← *imm*
		fault(Breakpoint_Instruction)
			B/I/M	none

Changes to debug registers and PSR are not necessarily observed by following instructions. Software should issue a data serialization operation to ensure modifications to DBR, PSR.db, PSR.tb and PSR.lp are observed before a dependent instruction is executed. For register changes to IBR and PSR.db that affect fetching of subsequent instructions, software must issue an instruction serialization operation.

On some implementations, a hardware debugger may use two or more of these registers pairs for its own use. When a hardware debugger is attached, as few as 2 DBR pairs and as few as 2 IBR pairs may be available for software use. Software should be prepared to run with fewer than the implemented number of IBRs and/or DBRs if the software is expected to be debuggable with a hardware debugger. When a hardware debugger is not attached, at least 4 IBR pairs and 4 DBR pairs are available for software use.

Any debug registers used by an attached hardware debugger are allocated from the highest register numbers first (e.g. if only 2 DBR pairs are available to software, the available registers are DBR[0-3]).

Note: When a hardware debugger is attached and is using two or more debug registers pairs, the processor does not forcibly partition the registers between software and hardware debugger use; that is, the processor does not prevent software from reading or modifying any of the debug registers being used by the hardware debugger. However, if software modifies any of the registers being used by the hardware debugger, processor and/or hardware debugger operation may become undefined, or the processor and/or hardware debugger may crash.

7.1.2 Debug Address Breakpoint Match Conditions

For virtual memory accesses, breakpoint address registers contain the virtual addresses of the debug breakpoint. For physical accesses, the addresses in these registers are treated as a physical address. Software should be aware that debug registers configured to fault on virtual references, may also fault on a physical reference if translations are disabled. Likewise a debug register configured for physical references can fault on virtual references that match the debug breakpoint registers.

The range of addresses detected by the DBR and IBR registers for memory references by Itanium instructions is defined as:

- Instruction and single or multi-byte aligned data memory references that access any memory byte specified by the IBR/DBR address and mask fields results in an Instruction/Data Debug fault regardless of datum size. Implementations must only report a Debug fault if the specified aligned byte(s) are referenced.
- Floating-point load double/integer pair, floating-point spill/fill and 10-byte operands are treated as 16-byte datums for breakpoint matching, if the accesses are aligned. Floating-point load single pair operands are treated as 8-byte datums for breakpoint matching, if the accesses are aligned.
- If data memory references are unaligned, multi-byte memory references that access any memory byte specified by DBR address and mask fields result in a breakpoint Data Debug fault regardless of datum size. Processor implementations may also report additional breakpoint Data Debug faults for addresses not specifically specified by the DBR registers. Debugging software should perform a byte by byte breakpoint analysis of each address accessed by multi-byte unaligned datums to detect true breakpoint conditions.

Address breakpoint Data Debug faults are not reported for the Flush Cache (`fc`), non-faulting `probe`, non-faulting `lfetch`, insert TLB (`itc, itr`), purge TLB (`ptc, ptr`), or translation access (`thash, ttag, tak, tpa`) instructions. Accesses by the RSE to a debug region are checked, but the Data Debug fault is not reported until a subsequent `alloc`, `br.ret`, `rfi`, `loadrs`, or `flushrs` which requires that the faulting load or store actually occur.

The range of addresses detected by the DBR and IBR registers for IA-32 memory references is defined as:

- Instruction memory references where the first byte of the IA-32 instruction match the IBR address and mask fields results in an IA-32_Exception(Debug) fault. Subsequent bytes of a multiple byte IA-32 instruction are not compared against the IBR registers for breakpoints. The upper 32-bits of the IBR addr field must be zero to detect IA-32 instruction memory references.
- IA-32 single or multi-byte data memory references that access any memory byte specified by the DBR address and mask fields results in an IA-32_Exception(Debug) trap regardless of datum size and alignment. The processor ensures that all data breakpoint traps are precisely reported. Data breakpoint traps are reported if and only if any byte in the IA-32 data memory reference matches the DBR address and mask fields. No spurious data breakpoint events are generated for IA-32 data memory operands that are unaligned, nor are breakpoints reported if no bytes of the operand lie within the address range specified by the DBR address and mask fields.

7.2 Performance Monitoring

Performance monitors allow processor events to be monitored by programmable counters or give an external notification (such as a pin or transaction) on the occurrence of an event. Monitors are useful for tuning application, operating system and system performance. Two sets of performance monitor registers are defined. Performance Monitor Configuration (PMC) registers are used to control the monitors. Performance Monitor Data (PMD) registers provide data values from the monitors. The performance monitors can record performance values from either the IA-32 or Itanium instruction set.

As shown in Figure 7-3, all processor implementations provide at least four performance counters (PMC/PMD[4]..PMC/PMD[7] pairs), and four performance counter overflow status registers (PMC[0]..PMC[3]). Performance monitors are also controlled by bits in the processor status register (PSR), the default control register (DCR) and the performance monitor vector register (PMV). Processor implementations may provide additional implementation-dependent PMC and PMD registers to increase the number of “generic” performance counters (PMC/PMD pairs). The remainder of the PMC and PMD register set is implementation dependent.

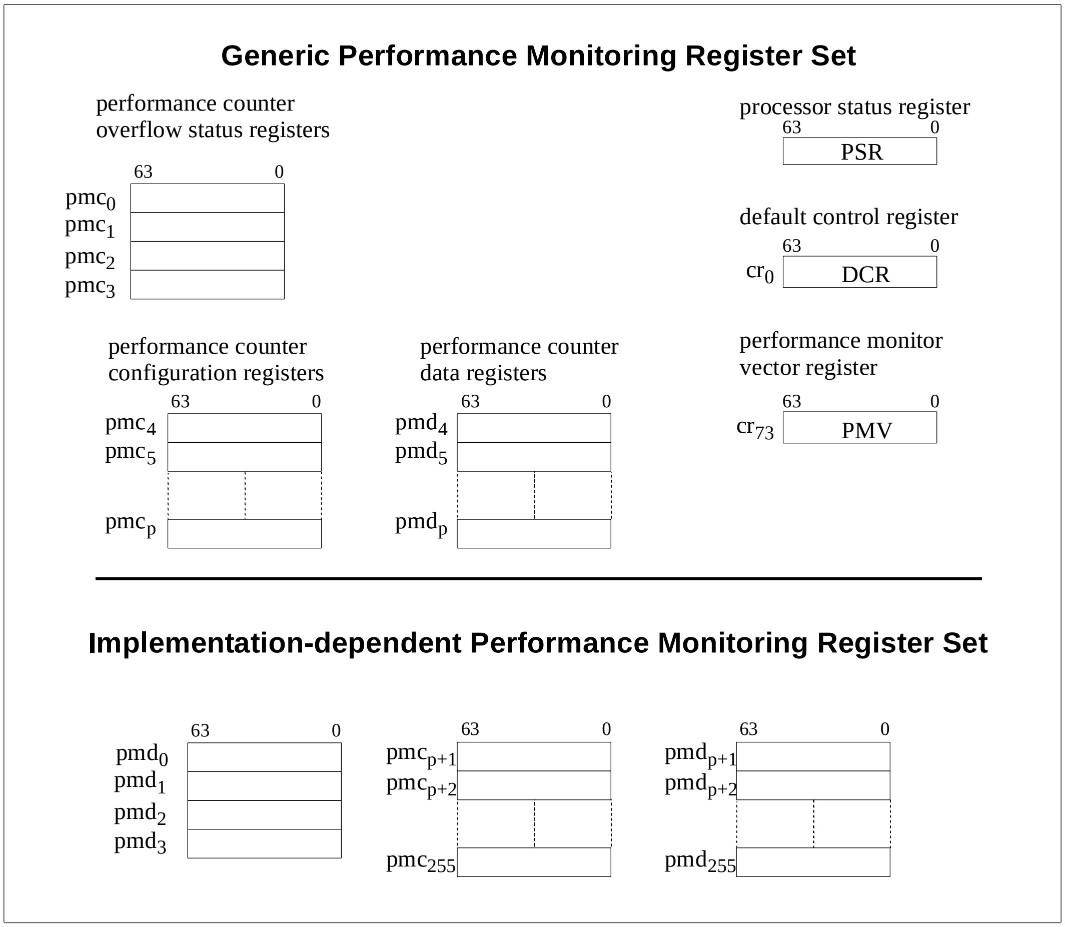
Event collection for implementation-dependent performance monitors is not specified by the architecture. Enabling and disabling functions are implementation dependent. For details, consult processor specific documentation.

Processor implementations may not populate the entire PMC/PMD register space. Reading of an unimplemented PMC or PMD register returns zero. Writes to unimplemented PMC or PMD registers are ignored; i.e., the written value is discarded.

Writes to PMD and PMC and reads from PMC are privileged operations. At non-zero privilege levels, these operations result in a Privileged Operation fault, regardless of the register address.

Reading of PMD registers by non-zero privilege level code is controlled by PSR.sp. When PSR.sp is one, PMD register reads by non-zero privilege level code return zero.

Figure 7-3. Performance Monitor Register Set


Generic Performance Monitoring Register Set
performance counter overflow status registers
63
0
pmc_0
pmc_1
pmc_2
pmc_3
processor status register
63
0
PSR
default control register
63
0
cr_0
DCR
performance counter configuration registers
63
0
pmc_4
pmc_5
pmc_p
performance counter data registers
63
0
pmd_4
pmd_5
pmd_p
performance monitor vector register
63
0
cr_{73}
PMV
Implementation-dependent Performance Monitoring Register Set
63
0
pmd_0
pmd_1
pmd_2
pmd_3
63
0
pmc_{p+1}
pmc_{p+2}
pmc_{255}
63
0
pmd_{p+1}
pmd_{p+2}
pmd_{255}

7.2.1 Generic Performance Counter Registers

Generic performance counter registers are PMC/PMD pairs that contiguously populate the PMC/PMD name space starting at index 4. At least 4 performance counter register pairs (PMC/PMD[4]..PMC/PMD[7]) are implemented in all processor models. Each counter can be configured to monitor events for any combination of privilege levels and one of several event metrics. The number of performance counters is implementation specific. The figures and tables use the symbol "p" to represent the index of the last implemented generic PMC/PMD pair. The bit-width W of the counters is also implementation specific. A counter overflow interrupt occurs when the counter wraps; i.e., a carry out from bit W-1 is detected. Figure 7-4 and Figure 7-5 show the fields in PMD and PMC respectively, while Table 7-3 and Table 7-4 describe the fields in PMD and PMC respectively.

Figure 7-4. Generic Performance Counter Data Registers (PMD[4]..PMD[p])

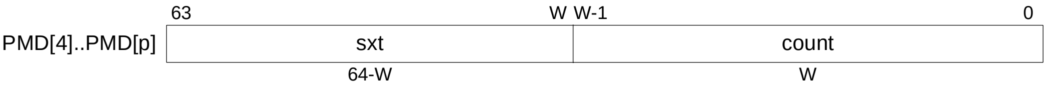


Table 7-3. Generic Performance Counter Data Register Fields

Field	Bits	Description
sxt	63:W	Writes are ignored. Reads return the value of bit W-1, so count values appear as sign extended.
count	W-1:0	Event Count. The counter is defined to overflow when the count field wraps (carry out from bit W-1).

Figure 7-5. Generic Performance Counter Configuration Register (PMC[4]..PMC[p])

	63 16	15 8	7	6	5	4	3 0
PMC[4]..PMC[p]	implementation specific	es	ig	pm	oi	ev	plm
	48	8	1	1	1	1	4

Table 7-4. Generic Performance Counter Configuration Register Fields (PMC[4]..PMC[p])

Field	Bits	Description
plm	3:0	Privilege Level Mask – controls performance monitor operation for a specific privilege level. Each bit corresponds to one of the 4 privilege levels, with bit 0 corresponding to privilege level 0, bit 1 with privilege level 1, etc. A bit value of 1 indicates that the monitor is enabled at that privilege level. Writing zeros to all plm bits effectively disables the monitor. In this state, the corresponding PMD register(s) do not preserve values, and the processor may choose to power down the monitor.
ev	4	External visibility – When 1, an external notification (such as a pin or transaction) is provided whenever the monitor overflows. Overflow occurs when a carry out from bit W-1 is detected.
oi	5	Overflow interrupt – When 1, a Performance Monitor Interrupt is raised and the performance monitor freeze bit (PMC[0].fr) is set when the monitor overflows. When 0, no interrupt is raised and the performance monitor freeze bit (PMC[0].fr) remains unchanged. Overflow occurs when a carry out from bit W-1 is detected. See "Performance Monitor Overflow Status Registers (PMC[0]..PMC[3])" for details on configuring interrupt vectors.
pm	6	Privileged monitor – When 0, the performance monitor is configured as a user monitor, and enabled by PSR.up. When PMC.pm is 1, the performance monitor is configured as a privileged monitor, enabled by PSR.pp, and the corresponding PMD can only be read by privileged software.
ig	7	ignored
es	15:8	Event select – selects the performance event to be monitored. Actual event encodings are implementation dependent. Some processor models may not implement all event select (es) bits. At least one bit of es must be implemented on all processors. Unimplemented es bits are ignored.
implem. specific	63:16	Implementation specific bits – Reads from implemented bits return implementation-dependent values. For portability, software should write what was read; i.e., software may not use these bits as storage. Hardware will ignore writes to unimplemented bits.

Event collection is controlled by the Performance Monitor Configuration (PMC) registers and the processor status register (PSR). Four PSR fields (PSR.up, PSR.pp, PSR.cpl and PSR.sp) and the performance monitor freeze bit (PMC[0].fr) affect the behavior of all generic performance monitor registers. Finer, per monitor, control of generic performance monitors is provided by two PMC register fields (PMC[i].plm, PMC[i].pm). Event collection for a generic monitor is enabled under the following constraints:

- Generic Monitor Enable[i] =(not PMC[0].fr) and PMC[i].plm[PSR.cpl] and ((not (PMC[i].pm) and PSR.up) or (PMC[i].pm and PSR.pp))

Generic performance monitor data registers (PMD[i]) can be configured to be user readable (useful for user level sampling and tracing user level processes) by setting the PMC[i].pm bit to 0. All user-configured monitors can be started and stopped synchronously by the user mask instructions (`rum` and `sum`) by altering PSR.up. User-configured monitors can be secured by setting PSR.sp to 1. A user-configured secured monitor continues to collect performance values; however, reads of PMD, by non-privileged code, return zeros until the monitor is unsecured.

Monitors configured as privileged (PMC[i].pm is 1) are accessible only at privilege level 0; otherwise, reads return zeros. All privileged monitors can be started and stopped synchronously by the system mask instructions (`rsm` and `ssm`) by altering PSR.pp. Table 7-5 summarizes the effects of PSR.sp, PMC[i].pm, and PSR.cpl on reading PMD registers.

Updates to generic PMC registers and PSR bits (up, pp, is, sp, cpl) require implicit or explicit data serialization prior to accessing an affected PMD register. The data serialization ensures that all prior PMD reads and writes as well as all prior PMC writes have completed.

Table 7-5. Reading Performance Monitor Data Registers

PSR.sp	PMC[i].pm	PSR.cpl	PMD Reads Return
0	0	0	PMD value
0	1	0	PMD value
1	0	0	PMD value
1	1	0	PMD value
0	0	>0	PMD value
0	1	>0	0
1	0	>0	0
1	1	>0	0

Generic PMD counter registers may be read by software without stopping the counters. The processor guarantees that software will see monotonically increasing counter values. Software must accept a level of sampling error when reading the counters due to various machine stall conditions, interruptions, and bus contention effects, etc. The level of sampling error is implementation specific. More accurate measurements can be obtained by disabling the counters and performing an instruction serialize operation for instruction events or data serialize operation for data events before reading the monitors. Other (non-counter) implementation-dependent PMD registers can only be read reliably when event monitoring is frozen (PMC[0].fr is one).

For accurate PMD reads of disabled counters, data serialization (implicit or explicit) is required between any PMD read and a subsequent `ssm` or `sum` (that could toggle PSR.up or PSR.pp from 0 to 1), or a subsequent `epc`, demoting `br.ret` or branch to IA-32 (`br.ia`) (that could affect PSR.cpl or PSR.is). Note that implicit post-serialization semantics of `sum` do not meet this requirement.

Table 7-6 defines the instructions used to access the PMC and PMD registers.

Table 7-6. Performance Monitor Instructions

Mnemonic	Description	Operation	Instr Type	Serialization Required
mov pmd[r_3] = r_2	Move to performance monitor data register	PMD[GR[r_3]] ← GR[r_2]	M	data/inst
mov r_1 = pmd[r_3]	Move from performance monitor data register	GR[r_1] ← PMD[GR[r_3]]	M	none

Table 7-6. Performance Monitor Instructions (Continued)

Mnemonic	Description	Operation	Instr Type	Serialization Required
mov pmc[r_3] = r_2	Move to performance monitor configure register	PMC[GR[r_3]] ← GR[r_2]	M	data/inst
mov r_1 = pmc[r_3]	Move from performance monitor configure register	GR[r_1] ← PMC[GR[r_3]]	M	none

7.2.2 Performance Monitor Overflow Status Registers (PMC[0]..PMC[3])

Performance monitor interrupts may be caused by an overflow from a generic performance monitor or an implementation-dependent event from a model-specific monitor. The four performance monitor overflow registers (PMC[0]..PMC[3]) shown in Figure 7-6 indicate which monitor caused the interruption.

Each of the 252 overflow bits in the performance monitoring overflow status registers (PMC[0]..PMC[3]) corresponds to a generic performance counter pair or to an implementation-dependent monitor. For generic performance counter pairs, overflow status bit PMC[i/64]{i%64} corresponds to generic counter pair PMC/PMD[i], where 4<=i<=p, and p is the index of the last implemented generic PMC/PMD pair.

When a generic performance counter pair (PMC/PMD[n]) overflows and its overflow interrupt bit (PMC[n].oi) is 1, or an implementation-dependent monitor wants to report an event with an interruption, then the processor:

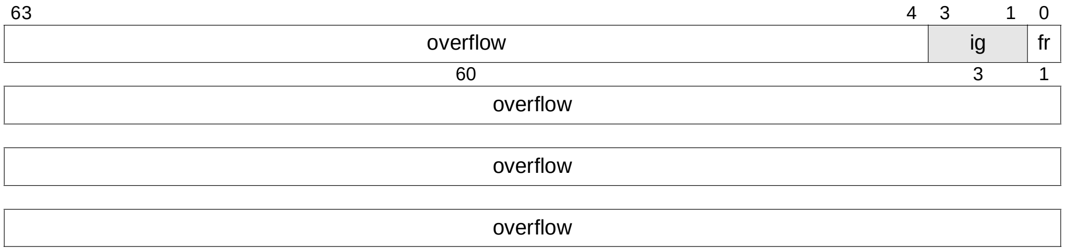
- Sets the corresponding overflow status bit in PMC[0]..PMC[3] to one, and
- Sets the freeze bit in PMC[0] which suspends event monitoring.

When a generic performance counter pair (PMC/PMD[n]) overflows, and its overflow interrupt bit (PMC[n].oi) is 0, the corresponding overflow status register bit is set to one. However, in this case of counter wrap without interrupt, the freeze bit in the PMC[0] is left unchanged, and event monitoring continues.

If control register bit PMV.m is one, a performance monitoring overflow interrupt is disabled from being pended. When PMV.m is zero, the interruption is received and held pending. (Further masking by the PSR.i, TPR and in-service masking can keep the interrupt from being raised.) Figure 7-6 shows the Performance Monitor Overflow Status registers.

Implementation-dependent PMD registers 0-3 cannot report events in the overflow registers; those 4 bit positions are used for other purposes.

Figure 7-6. Performance Monitor Overflow Status Registers (PMC[0]..PMC[3])



If the PMC[0] freeze bit is set (either by a performance counter overflow or an explicit software write), the processor suspends all event monitoring, i.e., counters do not increment, and overflow bits as well as model-specific monitoring are frozen. Writing a zero to the freeze bit resumes event monitoring.

Table 7-7. Performance Monitor Overflow Register Fields (PMC[0]..PMC[3])

Register	Field	Bits	Description
PMC[0]	fr	0	Performance Monitor “freeze” bit. This bit is volatile state, i.e., it is set by the processor whenever: • a generic performance monitor overflow occurs and its overflow interrupt bit (PMC[n].oi) is set to one. • a model-specific performance monitor signals an interrupt. The freeze bit can also be set by software to enable or disable all event monitoring. If the freeze bit is one, event monitoring is disabled. If the freeze bit is zero, event monitoring is enabled.
PMC[0]	ig	3:1	Ignored
PMC[0]..PMC[3]	overflow	implemented monitors	Bit vector indicating which performance monitor overflowed. Overflow status bits are sticky, they are set to 1 by the processor if the corresponding PMD overflows; otherwise they are left unchanged. Multiple overflow status bits may be set, independent of whether counter overflow causes an interrupt or not.
		unimplemented monitors	Ignored

Multiple overflow bits may be set, if counters overflow concurrently. The overflow bits and the freeze bit are sticky; i.e., the processor sets them to one but never resets them to zero. It is software’s responsibility to reset the overflow and freeze bits.

The overflow status bits are populated only for implemented counters. Overflow bits of unimplemented counters read as zero and writes are ignored.

7.2.3 Performance Monitor Events

The set of monitored events is implementation specific. All processor models are required to provide at least two events: the number of retired instructions, and the number of processor clock cycles. Events may be monitorable only by a subset of the available counters. PAL calls provide an implementation-independent interface that provides information on the number of implemented counters, their bit-width, the number and location of other (non-counter) monitors, etc.

7.2.4 Implementation-independent Performance Monitor Code Sequences

This section describes implementation-independent code sequences for servicing overflow interrupts and context switches of the performance monitors. For forward compatibility, the code sequences outlined in Section 7.2.4.1 and Section 7.2.4.2 use PAL-provided implementation-specific information to collect/preserve data values for all implemented counters.

7.2.4.1 Performance Monitor Interrupt Service Routine

When a performance counter register overflows and, for generic performance counters, the PMC[n].oi bit is set, the processor suspends event collection, and sets the freeze bit in PMC[0]. Event monitoring remains frozen until software clears the freeze bit. Performance monitor interrupts may be caused by an overflow of any of the counters. The processor indicates which performance monitor overflowed in the performance monitor overflow status registers (PMC[0]..PMC[3]). If multiple counters overflow concurrently, multiple overflow bits will be set to one. For forward compatibility, event collection interrupt handlers should follow the implementation-independent overflow interrupt service routine outlined in Figure 7-7.

After a context switch from a context which had performance monitoring enabled to an unmonitored context, the freeze bit will be set (see Section 7.2.4.2). A pending overflow interrupt which was targeted at a monitored process may not be delivered until a non-monitored process is running. A bogus interrupt is one where the freeze bit is zero or performance monitoring is disabled in the PSR.

Figure 7-7. Performance Monitor Interrupt Service Routine (Implementation Independent)

```
//Assumes PSR.up and PSR.pp are switched to zero together
if ((PMC[0].fr==1) && (PSR.up == 1) || (PSR.pp == 1)){
   // freeze bit is set. Search for interrupt.
   for (i=0; i< 4; i++) {
      if (PMC[i] != 0) {
         startbit = (i==0) ? 4 : 0;
         for (j=startbit; j < 64 ; j++) {
            if (PMC[i]{j}) {
               counter_id = 64*i + j;
               if (counter_id > PAL_GENERIC_PMCPMD_PAIRS) {
                  Implementation_Specific_Update(counter_id);
               }
               else { // Generic PMC/PMD counter
                  if (PMC[counter_id].oi)
                     ovflcount[counter_id] += 1;
               }
            }
         } // scan overflow bits
      }
   }
}
// Either ignore bogus interrupt or clear PMC[3]..PMC[1]
// and PMC[0] last (clears freeze bit)
for (i=3; i>=0; i--) { PMC[i] = 0; }
rfi
```

7.2.4.2 Performance Monitor Context Switch

The context switch routine described in Figure 7-8 defines the implementation-independent context switching of Itanium performance monitors. Using bit masks provided by PAL (PAL`PMCmask`, PAL`PMDmask`) the routine can generically save/restore the contents of all implementation-specific performance monitoring registers. If the outgoing context is monitored (PSR.pp or PSR.up are set), then in addition to preserving all PMC and PMD registers, if the context switch routine determines (by reading the freeze bit) that the outgoing context has a pending performance monitor interrupt, software preserves the outgoing context's overflow status registers (PMC[0]..PMC[3]). The context switch handler then restores the performance monitor freeze bit which resets event collection for the new context. Sometime into the incoming (possibly unmonitored) context, the performance overflow interrupt service routine will run, but by looking at the status of the freeze bit software can determine whether this interrupt can be ignored (for details refer to Section 7.2.4.1).

When switching back to the original context (that originally caused the counter overflow), the previously saved freeze bit can be inspected. If it was set (meaning there was a pending performance monitor interrupt), then the context switch routine posts an interrupt message to the incoming context's processor at the performance monitor vector specified by the PMV register. This will result in a new performance monitor overflow interrupt in the correct context. Essentially, the interrupt message is "replaying" the overflow interrupt that was missed because of the context switch.

Figure 7-8. Performance Monitor Overflow Context Switch Routine

```
// in context or thread switch

if (outgoing process is monitored (PSR.up or PSR.pp are set)) {
   1. Turn-off counting and ignore interrupts for context switch
      of counters.
      1a)   if not already done, raise interrupt priority above
            perf. mon overflow vector
      1b)   read and preserve PSR.up, PSR.pp, PSR.sp
      1c)   clear PSR.up, clear PSR.pp
      1d)   srlz.d
   2. Check for pending interrupt: Preserve Interrupt State
      2a)   read and preserve PMC[0]..PMC[3]
   3. Set freeze bit
      This ensures that PMD registers remain stable for context
      switch. Also, for restoration of incoming context, if PSR
      of the incoming process enables PSR.up or PSR.pp, the
      counters won’t start up, until they have been completely
      restored.
      3a) write one to freeze bit (PMC[0].fr=1)
      3b) srlz.d
   4. Preserve PMC/PMD contents
      4a) For each PMC whose PALPMCmask bit is set, preserve PMC.
      4b) For each PMD whose PALPMDmask bit is set, preserve PMD.
}

.... continue context switch ......

// Now in incoming process/thread
if (incoming process is monitored (PSR.up or PSR.pp are set)) {
   // Note that the context switch itself already restored PSR
   // with the original values of PSR.pp, PSR.up and PSR.sp
   // (inverse of step 1b above). Event counting is disabled,
   // because PMC[0].fr is one (step 3a above).

   5. Restore PMC/PMD contents (inverse of step 4 above)
      5a) For each PMC whose PALPMCmask bit is set, reload PMC.
      5b) For each PMD whose PALPMDmask bit is set, reload PMD.

   6. Restore Interrupt State (inverse of step 2 and 1a above)
      6a)   if (preserved freeze bit was set) {
               send myself a performance monitor interrupt
               (store to interrupt address)
            }
      6b)   Restore PMC[3], PMC[2], PMC[1], and finally PMC[0].
            Write PMC[0] last, which restores the state of the
            performance monitor freeze bit.
      6c)   srlz.d
      6d)   lower interrupt priority below perf. mon overflow
            vector
}
```

Interruption Vector Descriptions 8

Chapter 5 describes the interruption mechanism and programming model for the Itanium architecture. This chapter describes the IVA-based interruption handlers. "Interruption Vector Descriptions" describes all the Itanium IVA-based interruption vectors and "IA-32 Interruption Vector Definitions" describes all of the IA-32 interrupt vectors. PAL-based interruptions are described in Chapter 11, "Processor Abstraction Layer". Note that unless otherwise noted, references to "interruption" in this chapter refer to IVA-based interruptions. See "Interruption Definitions" on page 2:79.

8.1 Interruption Vector Descriptions

The section lists all the Itanium interruption vectors. It describes the interruption vectors and the parameters that are defined when the vector is entered.

If an interruption is independent of the executing instruction set (including IA-32), such as an external interrupt or TLB fault, common Itanium interruption vectors are used. For exceptions and intercept conditions that are specific to the IA-32 instruction set three IA-32 specific vectors are used; IA-32_Exception, IA-32_Interrupt, and IA-32_Intercept.

Table 8-1 defines which interruption resources are written, are left unmodified, or are undefined for each interruption vector. The individual vector descriptions below list interruption specific resources for each vector.

See "IVA-based Interruption Handling" on page 2:85 for details on how the processor handles an interruption. See "Interruption Control Registers" on page 2:29 for the definition of bit fields within the interruption resources.

8.2 ISR Settings

For each of the interruption vectors, a figure depicts the ISR setting. These figures show the value that hardware writes into the ISR for the corresponding interruption.

Table 8-2 provides an overview of ISR settings for all of the interruption vectors.

For some of the vectors, certain bits will always be 0 (or 1) simply because no instruction that would set that bit differently can ever end up on that vector. For example, ISR.sp is always 0 in the Break Instruction vector because ISR.sp is only set by speculative loads, and speculative loads can never take a Break Instruction fault.

After interruption from the IA-32 instruction set, the following ISR bits will always be zero - ISR.ni, ISR.na, ISR.sp, ISR.rs, ISR.ir, ISR.ei, and ISR.ed.

ISR.code settings for non-access instructions are described in “Non-access Instructions and Interruptions” on page 2:87.

Table 8-3 on page 2:151 provides an overview of ISR.code field on all Itanium traps.

8.3 Interruption Vector Definition

Table 8-1. Writing of Interruption Resources by Vector

Interruption Resource	IIP, IPSR, IIPA, IFS.v		IFA		ITIR		IHA		IIM		ISR	
PSR.ic at time of interruption	0	1	0	1	0	1	0	1	0	1	0	1
Interruption Vector												
Alternate Data TLB vector	n/a[a]	W[b]	n/a	W	n/a	W	n/a	x[c]	n/a	x	n/a	W
Alternate Instruction TLB vector	-[d]	W	-	W	-	W	x	x	x	x	W	W
Break Instruction vector	-	W	x	x	x	x	x	x	-	W	W	W
Data Access Rights vector	-	W	-	W	-	W	x	x	x	x	W	W
Data Access-Bit vector	-	W	-	W	-	W	x	x	x	x	W	W
Data Key Miss vector	-	W	-	W	-	W	x	x	x	x	W	W
Data Nested TLB vector	-	n/a	-	n/a	-	n/a	-	n/a	x	n/a	-	n/a
Data TLB vector	n/a	W	n/a	W	n/a	W	n/a	W	n/a	x	n/a	W
Debug vector	-	W	-	W	x	x	x	x	x	x	W	W
Dirty-Bit vector	-	W	-	W	-	W	x	x	x	x	W	W
Disabled FP-Register vector	-	W	x	x	x	x	x	x	x	x	W	W
External Interrupt vector	-	W	x	x	x	x	x	x	x	x	W	W
Floating-point Fault vector	-	W	x	x	x	x	x	x	x	x	W	W
Floating-point Trap vector	-	W	x	x	x	x	x	x	x	x	W	W
General Exception vector	-	W	x	x	x	x	x	x	x	x	W	W
IA-32 Exception Vector	n/a	W	n/a	x	n/a	x	n/a	x	n/a	x	n/a	W
IA-32 Intercept Vector	n/a	W	n/a	x	n/a	x	n/a	x	n/a	W	n/a	W
IA-32 Interrupt Vector	n/a	W	n/a	x	n/a	x	n/a	x	n/a	x	n/a	W
Instruction Access Rights vector	-	W	-	W	-	W	x	x	x	x	W	W
Instruction Access-Bit vector	-	W	-	W	-	W	x	x	x	x	W	W
Instruction Key Miss vector	-	W	-	W	-	W	x	x	x	x	W	W
Instruction TLB vector	-	W	-	W	-	W	-	W	x	x	W	W
Key Permission vector	-	W	-	W	-	W	x	x	x	x	W	W
Lower-Privilege Transfer Trap vector	-	W	x	x	x	x	x	x	x	x	W	W
NaT Consumption vector												
- reg	-	W	-	x	x	x	x	x	x	x	W	W
- data/instr	-	W	-	W	x	x	x	x	x	x	W	W
Page Not Present vector	-	W	-	W	-	W	x	x	x	x	W	W
Single Step Trap vector	-	W	x	x	x	x	x	x	x	x	W	W
Speculation vector	-	W	x	x	x	x	x	x	-	W	W	W
Taken Branch Trap vector	-	W	x	x	x	x	x	x	x	x	W	W
Unaligned Reference vector	-	W	-	W	x	x	x	x	x	x	W	W
Unsupported Data Reference vector	-	W	-	W	x	x	x	x	x	x	W	W
VHPT Translation vector	n/a	W	n/a	W	n/a	W	n/a	W	n/a	x	n/a	W

[a] “n/a” indicates that this cannot happen.
[b] “W” indicates that the resource is written with a new value.

c. “x” indicates that the resource may or may not be written; whether it is written and with what value is implementation specific.
d. “-” indicates that the resource is not written.

Table 8-2. ISR Values on Interruption

Vector / Interruption	ed	ei[a]	so	ni[b]	ir[c]	rs[d]	sp[e]	na[f]	r	w	x
Alternate Data TLB vector											
Alternate Data TLB fault	ed[k]	ri	so	ni[l]	0	rs	sp	na	r	w	0
IR Alternate Data TLB fault	0	ri	0	ni[l]	1	1	0	0	1	0	0
Alternate Instruction TLB vector											
Alternate Instruction TLB fault	0	ri	0	ni	0	0	0	0	0	0	1
Break Instruction vector											
Break Instruction fault	0	ri	0	ni	0	0	0	0	0	0	0
Data Access Rights vector											
Data Access Rights fault	ed[k]	ri	so	ni	0	rs	sp	na	r	w	0
IR Data Access Rights fault	0	ri	0	ni	1	1	0	0	1	0	0
Data Access-Bit vector											
Data Access Bit fault	ed[k]	ri	so	ni	0	rs	sp	na	r	w	0
IR Data Access Bit fault	0	ri	0	ni	1	1	0	0	1	0	0
Data Key Miss vector											
Data Key Miss fault	ed[k]	ri	so	ni	0	rs	sp	na	r	w	0
IR Data Key Miss fault	0	ri	0	ni	1	1	0	0	1	0	0
Data Nested TLB vector[g]											
Data Nested TLB fault	-	-	-	-	-	-	-	-	-	-	-
IR Data Nested TLB fault	-	-	-	-	-	-	-	-	-	-	-
Data TLB vector											
Data TLB fault	ed[k]	ri	so	ni[l]	0	rs	sp	na	r	w	0
IR Data TLB fault	0	ri	0	ni[l]	1	1	0	0	1	0	0
Debug vector											
Data Debug fault	ed[k]	ri	0	ni	0	rs	sp	na	r	w	0
Instruction Debug fault	0	ri	0	ni	0	0	0	0	0	0	1
IR Data Debug fault	0	ri	0	ni	1	1	0	0	1	0	0
Dirty-Bit vector											
Data Dirty Bit fault	ed[k]	ri	so	ni	0	rs	0	na[h]	r	1	0
Disabled FP-Register vector											
Disabled Floating-Point Register fault	0	ri	0	ni	0	0	sp	0	r	w	0
External Interrupt vector											
External Interrupt	0	ri	0	ni	ir[i]	0	0	0	0	0	0
Floating-point Fault vector											
Floating-Point Exception fault	0	ri	0	ni	0	0	0	0	0	0	0
Floating-point Trap vector											
Floating-Point Exception trap	0	ei	0	ni	0	0	0	0	0	0	0
General Exception vector											
Disabled ISA Transition fault	0	ri	0	ni	0	0	0	0	0	0	0
Illegal Dependency fault	0	ri	0	ni	0	0	0	0	0	0	0
Illegal Operation fault	0	ri	0	ni	0	0	0	0	0	0	0
IR Unimplemented Data Address fault	0	ri	0	ni	1	1	0	0	1	0	0
Privileged Operation fault	0	ri	0	ni	0	0	0	na	0	0	0

Table 8-2. ISR Values on Interruption (Continued)

Vector / Interruption	ed	ei[a]	so	ni[b]	ir[c]	rs[d]	sp[e]	na[f]	r	w	x
Privileged Register fault	0	ri	0	ni	0	0	0	0	0	0	0
Reserved Register/Field fault	0	ri	0	ni	0	0	0	0	0	0	0
Unimplemented Data Address fault	0	ri	0	ni	0	rs	0	na[j]	r	w	0
IA-32 Exception vector	0	0	0	0	0	0	0	0	0	0	x
IA-32 Intercept vector	0	0	0	0	0	0	0	0	r	w	0
IA-32 Interrupt vector	0	0	0	0	0	0	0	0	0	0	0
Instruction Access Rights vector											
Instruction Access Rights fault	0	ri	0	ni	0	0	0	0	0	0	1
Instruction Access-Bit vector											
Instruction Access Bit fault	0	ri	0	ni	0	0	0	0	0	0	1
Instruction Key Miss vector											
Instruction Key Miss fault	0	ri	0	ni	0	0	0	0	0	0	1
Instruction TLB vector											
Instruction TLB fault	0	ri	0	ni	0	0	0	0	0	0	1
Key Permission vector											
Data Key Permission fault	ed[k]	ri	so	ni	0	rs	sp	na	r	w	0
Instruction Key Permission fault	0	ri	0	ni	0	0	0	0	0	0	1
IR Data Key Permission fault	0	ri	0	ni	1	1	0	0	1	0	0
Lower-Privilege Transfer Trap vector											
Lower-Privilege Transfer trap	0	ei	0	ni	ir	0	0	0	0	0	0
Unimplemented Instruction Address trap	0	ei	0	ni	ir	0	0	0	0	0	0
NaT Consumption vector											
Data NaT Page Consumption fault	0	ri	so	ni	0	rs	0	na	r	w	0
Instruction NaT Page Consumption fault	0	ri	0	ni	0	0	0	0	0	0	1
IR Data NaT Page Consumption fault	0	ri	0	ni	1	1	0	0	1	0	0
Register NaT Consumption fault	0	ri	0	ni	0	0	0	na	r	w	0
Page Not Present vector											
Data Page Not Present fault	ed[k]	ri	so	ni	0	rs	sp	na	r	w	0
Instruction Page Not Present fault	0	ri	0	ni	0	0	0	0	0	0	1
IR Data Page Not Present fault	0	ri	0	ni	1	1	0	0	1	0	0
Single Step Trap vector											
Single Step trap	0	ei	0	ni	ir	0	0	0	0	0	0
Speculation vector											
Speculative Operation fault	0	ri	0	ni	0	0	0	0	0	0	0
Taken Branch Trap vector											
Taken Branch trap	0	ei	0	ni	ir	0	0	0	0	0	0
Unaligned Reference vector											
Unaligned Data Reference fault	ed	ri	0	ni	0	0	sp	0	r	w	0
Unsupported Data Reference vector											
Unsupported Data Reference fault	ed	ri	0	ni	0	0	0	0	r	w	0
VHPT Translation vector											
IR VHPT Data fault	0	ri	0	ni[l]	1	1	0	0	1	0	0
VHPT Data fault	ed[k]	ri	so	ni[l]	0	rs	sp	na	r	w	0
VHPT Instruction fault	0	ri	0	ni	0	0	0	0	0	0	1

a. ISR.ei is equal to IPSR.ri for all faults and external interrupts (1 for faults and interrupts on the L+X instruction of an MLX). For traps, ISR.ei points at the excepting instruction (2 for traps on the L+X instruction of an MLX).
b. If ISR.ni is 1, the interruption occurred either when PSR.ic was 0 or was in-flight.
c. ISR.ri captures the value of RSE.CFLE at the time of an interruption.
d. ISR.rs is 1 for interruptions caused by mandatory RSE fills/spills and 0 for all others.
e. ISR.sp is 1 for interruptions caused by speculative loads and zero for all others.
f. ISR.na is 1 for interruptions caused by non-access instructions and zero for all others.
g. ISR is not written.
h. A faulting `probe.w.fault` or `probe.rw.fault` can cause a Dirty Bit fault on a non-access instruction.
i. ISR.ir is 1 if an external interrupt was taken when mandatory RSE fills caused by a `br.ret` or `rfi` were re-loading the current register stack frame.
j. A faulting `lfetch.fault` or `probe.fault` to an unimplemented address will set ISR.na to 1.
k. ISR.ed is 0 if the interruption was caused by a mandatory RSE fill or spill.
l. If PSR.ic was 0 when the interruption was taken, these faults do not occur, but a Data Nested TLB fault is taken.

Table 8-3 provides the definition for the ISR.code field on all Itanium traps. Hardware will always deliver the highest priority enabled trap. Software must look at the ISR.code bit vector to determine if any lower priority trap occurred at the same time as the trap being processed.

Table 8-3. ISR.code Fields on Intel® Itanium™ Traps

Field	Bit Range	Description
fp	0	Floating-Point Exception trap
lp	1	Lower-Privilege Transfer trap
tb	2	Taken Branch trap
ss	3	Single Step trap
ui	4	Unimplemented Instruction Address trap
fp trap code	7	IEEE O (overflow) exception (Parallel FP-LO)
fp trap code	8	IEEE U (underflow) exception (Parallel FP-LO)
fp trap code	9	IEEE I (inexact) exception (Parallel FP-LO)
fp trap code	10	FPA, Added one to significand when rounding (Parallel FP-LO)
fp trap code	11	IEEE O (overflow) exception (Normal or Parallel FP-HI)
fp trap code	12	IEEE U (underflow) exception (Normal or Parallel FP-HI)
fp trap code	13	IEEE I (inexact) exception (Normal or Parallel FP-HI)
fp trap code	14	FPA, Added one to significand when rounding (Normal or Parallel FP-HI).

Table 8-4. Interruption Vectors Sorted Alphabetically

Vector Name	Offset	Page
Alternate Data TLB vector	0x1000	2:158
Alternate Instruction TLB vector	0x0c00	2:157
Break Instruction vector	0x2c00	2:165
Data Access Rights vector	0x5300	2:170
Data Access-Bit vector	0x2800	2:164
Data Key Miss vector	0x1c00	2:161
Data Nested TLB vector	0x1400	2:159
Data TLB vector	0x0800	2:156
Debug vector	0x5900	2:177
Dirty-Bit vector	0x2000	2:162
Disabled FP-Register vector	0x5500	2:173
External Interrupt vector	0x3000	2:166

Table 8-4. Interruption Vectors Sorted Alphabetically (Continued)

Vector Name	Offset	Page
Floating-Point Fault vector	0x5c00	2:180
Floating-Point Trap vector	0x5d00	2:181
General Exception vector	0x5400	2:171
IA-32 Exception vector	0x6900	2:185
IA-32 Intercept vector	0x6a00	2:186
IA-32 Interrupt vector	0x6b00	2:187
Instruction Access Rights vector	0x5200	2:169
Instruction Access-Bit vector	0x2400	2:163
Instruction Key Miss vector	0x1800	2:160
Instruction TLB vector	0x0400	2:155
Key Permission vector	0x5100	2:168
Lower-Privilege Transfer Trap vector	0x5e00	2:182
NaT Consumption vector	0x5600	2:174
Page Not Present vector	0x5000	2:167
Single Step Trap vector	0x6000	2:184
Speculation vector	0x5700	2:176
Taken Branch Trap vector	0x5f00	2:183
Unaligned Reference vector	0x5a00	2:178
Unsupported Data Reference vector	0x5b00	2:179
VHPT Translation vector	0x0000	2:153

Name **VHPT Translation vector (0x0000)**

Cause The hardware VHPT walker encountered a TLB miss while attempting to reference the virtually addressed hashed page table for a memory reference (including IA-32).

Interruptions on this vector:

IR VHPT Data fault
VHPT Instruction fault
VHPT Data fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

IHA – The virtual address in the hashed page table which the hardware VHPT walker was attempting to reference.

ITIR – The ITIR contains default translation information for the virtual address contained in the IHA. The access key field within this register is set to the region id value from the region register selected by the virtual address in the IHA**.** The ITIR.ps field is set to the RR.ps field from the selected region register. All other fields are set to 0.

If the fault is due to a VHPT data fault for both original instruction and data references:

- IFA – The faulting address that the hardware VHPT walker was attempting to resolve.
- ISR – The ISR bits are set to reflect the original access on whose behalf the VHPT walker was operating. If the original operation was a non-access instruction then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0. For mandatory RSE fill or spill references, ISR.ed is always 0. The ISR.ni bit is 0 if PSR.ic was 1 when the interruption was taken, and is 1 if PSR.ic was in-flight. For IA-32 memory references the ISR.code, ni, ed, ei, ir, rs, sp, and na bits are always 0. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

If the fault is due to a VHPT instruction fault:

- IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits or, if the hardware VHPT walker was attempting to resolve a TLB miss, the virtual address of the translation.
- ISR – The ISR bits are set based on the original instruction fetch that the VHPT walker was attempting to resolve. The defined ISR bits are specified below. The ISR.ni bit is 0 if PSR.ic was 1 when the interruption was taken, and is 1 if PSR.ic was in-flight. For IA-32 memory references the ei and ni bits are always 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes This fault can only occur when PSR.ic is 1 or in-flight, and the VHPT walker is enabled for the referenced region. Refer to "VHPT Environment" on page 2:56 for details on VHPT enabling.

The original IFA address will be needed by the operating system page fault handler in the case where the page containing the VHPT entry has not yet been allocated. When the translation for the VHPT is available the handler must first move the address contained in the IHA to the IFA prior to the TLB insert.

Name **Instruction TLB vector (0x0400)**

Cause The instruction TLB entry needed by an instruction fetch (including IA-32) is absent, and the hardware VHPT walker could not find the translation in the VHPT, or the hardware VHPT walker is enabled but not implemented on this processor.

Interruptions on this vector:

Instruction TLB fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

IHA – The virtual address of the hashed page table entry which corresponds to the reference that raised this fault.

ITIR – The ITIR contains default translation information for the original instruction address. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below. The ISR.ni bit is 0 if PSR.ic was 1 when the interruption was taken, and is 1 if PSR.ic was in-flight. The ISR.ei and ni bits are always 0 for IA-32 memory references.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes This fault can only occur when PSR.ic is 1 or in-flight, the VHPT hardware walker is enabled for the referenced region, the PSR.it bit is 1, and the fetched instruction bundle is to be executed. Refer to "VHPT Environment" on page 2:56 for details on VHPT enabling.

The hardware VHPT walker may have failed due to an unimplemented page size, tag mismatch, illegal entry, or it may have terminated before reading the data. Software must be able to handle the case where the VHPT walker fails.

Name Data TLB vector (0x0800)

Cause

For memory references (including IA-32), the data TLB entry needed by the data access is absent, and the hardware VHPT walker could not find the translation in the VHPT, or the hardware VHPT walker is not implemented on this processor.

Interruptions on this vector:

IR Data TLB fault
Data TLB fault

Parameters

IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

IHA – The virtual address of the hashed page table entry which corresponds to the reference that raised this fault.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The address of the data being referenced.

ISR – If the interruption was due to a non-access operation then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0. For mandatory RSE fill or spill references, ISR.ed is always 0. The ISR.ni bit is 0 if PSR.ic was 1 when the interruption was taken, and is 1 if PSR.ic was in-flight. The ISR.code, ed, ei, ir, rs, sp and na bits are always 0 for IA-32 memory references. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

Notes

The fault can only occur on an IA-32 or Itanium load, store, semaphore, or non-access operation when PSR.dt is 1, and the VHPT hardware walker is enabled for the referenced region. This fault can only occur on a mandatory RSE load/store operation if PSR.rt is 1, and the VHPT hardware walker is enabled for the referenced region. Refer to “VHPT Environment” on page 2:56 for details on VHPT enabling.

The hardware VHPT walker may have failed due to an unimplemented page size, tag mismatch, illegal entry, or it may have terminated before reading the data. Software must be able to handle the case where the VHPT walker fails. The Data TLB fault is only taken if PSR.ic is 1 or in-flight, otherwise a Data Nested TLB fault is taken.

Name **Alternate Instruction TLB vector (0x0c00)**

Cause The instruction TLB entry needed by an instruction fetch (including IA-32) is absent, and the hardware VHPT walker was not enabled for this address.

Interruptions on this vector:

Alternate Instruction TLB fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the original instruction address. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.

ISR – For Itanium memory references, the ISR.ei bits are set to indicate which instruction caused the exception and ISR.ni is set to 0 if PSR.ic was 1 when the interruption was taken, and set to 1 if PSR.ic was 0 or in-flight. For IA-32 memory references the ISR.ei and ni bits are 0. The defined ISR bits are specified below.

The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes This fault can only occur when the VHPT walker is disabled for the referenced region, and the fetched instruction bundle is to be executed. Refer to "VHPT Environment" on page 2:56 for details on VHPT enabling.

Name **Alternate Data TLB vector (0x1000)**

Cause For memory references (including IA-32), the data TLB entry needed by data access is absent, and the hardware VHPT walker was not enabled for this address.

Interruptions on this vector:

IR Alternate Data TLB fault
Alternate Data TLB fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The address of the data being referenced.

ISR – If the interruption was due to a non-access operation then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0. For mandatory RSE fill or spill references, ISR.ed is always 0. The ISR.ni bit is 0 if PSR.ic was 1 when the interruption was taken, and is 1 if PSR.ic was in-flight. For IA-32 memory references the ISR.code, ed, ei, ir, rs, sp and na bits are 0. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

Notes The fault can only occur on an IA-32 or Itanium load, store, semaphore, or non-access operation when PSR.dt is 1, and the VHPT hardware walker is disabled for the referenced region. This fault can only occur on a mandatory RSE load/store operation if PSR.rt is 1, and the VHPT hardware walker is disabled for the referenced region. The Alternate Data TLB fault is only taken if PSR.ic is 1 or in-flight, otherwise a Data Nested TLB fault is taken. Refer to "VHPT Environment" on page 2:56 for details on VHPT enabling.

Name **Data Nested TLB vector (0x1400)**

Cause For memory references, the data TLB entry needed for a data reference is absent and PSR.ic is 0. Note: Data Nested TLB faults cannot occur during IA-32 instruction set execution, since PSR.ic must be 1.

Interruptions on this vector:

IR Data Nested TLB fault
Data Nested TLB fault

Parameters IIP, IPSR, IIPA, IFS, ISR are **unchanged** from their previous values; they contain information relating to the original interruption.

ITIR – is **unchanged** from the previous value.

IFA – is **unchanged** from the previous value and contains the original address of the data being referenced.

Notes This fault can only occur when PSR.dt is 1 and PSR.ic is 0 on a load, store, semaphore, or non-access instruction, or when PSR.rt is 1 and PSR.ic is 0 on a RSE mandatory load/store operation. Since the operating system is in control of the code executing at the time of the nested fault, it can by convention know which register contains the address that raised the nested event. As the PSR.ic bit is 0 on a nested fault, the IFA contains the original data address if the original interruption was caused by a data TLB fault. If the translation table entry required by the nested miss handler has not yet been allocated, then the address in the IFA will be passed to the operating system page fault handler. If the translation for the entry is available then the general register containing the nested fault address must be moved to the IFA prior to the insert. The ISR contains the ISR for the original faulting instruction, and not the ISR for the instruction that caused the nested fault.

Name **Instruction Key Miss vector (0x1800)**

Cause For instruction fetches (including IA-32), the PSR.it bit is 1, the PSR.pk bit is 1, and the access key from the TLB entry for the address of the executing instruction bundle does not match any of the valid protection keys.

Interruptions on this vector:

Instruction Key Miss fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the original instruction address. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. For IA-32 memory references the ISR.ei and ni bits are 0. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Name

Data Key Miss vector (0x1c00)

Cause

For memory references (including IA-32), the PSR.dt bit is 1, the PSR.pk bit is 1, and the access key from the TLB entry for the address referenced by a load, store, probe, or semaphore operation does not match any of the valid protection keys. The RSE may cause this fault if PSR.rt is 1, the PSR.pk bit is 1, and the access key from the TLB entry for the address referenced by an RSE mandatory load or store operation does not match any of the valid protection keys.

Interruptions on this vector:

IR Data Key Miss fault
Data Key Miss fault

Parameters

IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – Faulting data address.

ISR – If the interruption was due to a non-access operation then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0. For mandatory RSE fill or spill references, ISR.ed is always 0. For IA-32 memory references, the ISR.code, ed, ei, ni, ir, rs, sp, and na bits are 0. The value for the ISR bits depend on the type of access performed and are specified below.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">0</td><td colspan="8">0</td><td colspan="12">0</td><td colspan="4">code{3:0}</td></tr>
</table>

<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">0</td><td>ed</td><td colspan="2">ei</td><td>so</td><td>ni</td><td>ir</td><td>rs</td><td>sp</td><td>na</td><td>r</td><td>w</td><td>0</td></tr>
</table>

Notes

`Probe` and the faulting variant of `lfetch` are the only non-access instructions that will cause a data key miss fault.

Name **Dirty-Bit vector (0x2000)**

Cause IA-32 or Itanium store or semaphore operations to a page with the dirty-bit (TLB.d) equal to 0 in the data TLB.

Interruptions on this vector:

Data Dirty Bit fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – Faulting data address.

ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE spill references, ISR.ed is always 0. For IA-32 memory references, ISR.ed, ei, ni, and rs are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	0	rs	0	na	r	1	0

Notes Dirty Bit fault can only occur in these situations:

- When PSR.dt is 1 on an IA-32 or Itanium store or semaphore operation
- When PSR.dt is 1 on a `probe.w.fault` or `probe.rw.fault`
- When PSR.rt is 1 on an RSE mandatory store operation

For `probe.w.fault` or `probe.rw.fault` the ISR.na bit is set.

Only an IA-32 or Itanium semaphore, or `probe.rw.fault` operation would set ISR.r on a dirty bit fault.

Software is invoked to update the dirty bit in the data TLB entry and the Page table. The PSR.da bit can be used to suppress this fault for one executed instruction or one mandatory RSE store operation.

Name **Instruction Access-Bit vector (0x2400)**

Cause For instruction fetches (including IA-32), the access bit (TLB.a) in the TLB entry for this page is 0, and an instruction on the page is referenced.

Interruptions on this vector:

Instruction Access Bit fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. For IA-32 memory references the ISR.ei and ni bits are 0. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes The fault can only occur when PSR.it is 1 on an instruction reference (including IA-32). Software uses this fault for memory management page replacement algorithms. The PSR.ia bit can be used to suppress this fault for one executed instruction.

Name **Data Access-Bit vector (0x2800)**

Cause For data memory references (including IA-32), the access bit (TLB.a) in the TLB entry for this page is 0, and the page is referenced.

Interruptions on this vector:

IR Data Access Bit fault
Data Access Bit fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – Faulting data address.

ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0. For IA-32 memory references, ISR.code, ed, ei, ni, ir, rs, na and sp are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

Notes These faults can only occur in these situations:

- When PSR.dt is 1 on an IA-32 or Itanium load, store, or semaphore operation
- When PSR.dt is 1 on a `probe.fault`
- When PSR.dt is 1 on an `lfetch.fault`
- When PSR.rt is 1 on an RSE mandatory load/store operation

For `probe.fault` or `lfetch.fault` the ISR.na bit is set.

Software uses this fault for memory management page replacement algorithms. The PSR.da bit can be used to suppress this fault for one executed instruction or one mandatory RSE memory reference.

Name **Break Instruction vector (0x2c00)**

Cause An attempt is made to execute an Itanium `break` instruction.

Interruptions on this vector:

Break Instruction fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

IIM – Is updated with the break instruction immediate value.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">0</td><td colspan="8">0</td><td colspan="16">0</td></tr>
</table>

<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">0</td><td>0</td><td colspan="2">ei</td><td>0</td><td>ni</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Notes This fault cannot be raised by IA-32 instructions.

Name **External Interrupt vector (0x3000)**

Cause There are unmasked external interrupts pending from external devices, other processors, or internal processor events and:

- PSR.i is 1, while executing Itanium instructions
- PSR.i is 1 and (CFLAG.if is 0 or EFLAG.if is 1), while executing IA-32 instructions

IPSR.is indicates which instruction set was executing at the time of the interruption.

Interruptions on this vector:

External Interrupt

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

IVR – Highest priority unmasked pending external interrupt vector number. If there are no unmasked pending interrupts the “spurious” interrupt vector (15) is reported.

ISR – The ISR.ei bits are set to indicate which instruction was to be executed when the external interrupt event was taken. The defined ISR bits are specified below. For external interrupts taken in the IA-32 instruction set, ISR.ei, ni and ir bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	ir	0	0	0	0	0	0

Notes: Software is expected to avoid situations which could cause ISR.ni to be 1.

Name Page Not Present vector (0x5000)

Cause The bundle or IA-32 instruction being executed resides on a page for which the P-bit (TLB.p) in the instruction TLB entry is 0, or the data being referenced resides on a page for which the P-bit in the data TLB entry is 0.

Interruptions on this vector:

IR Data Page Not Present fault
Instruction Page Not Present fault
Data Page Not Present fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

If the fault is due to a data page not present fault for both instruction and data original references:

- IFA – The virtual address of the data being referenced.
- ISR – If the interruption was due to a non-access operation then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0. The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0. For IA-32 memory references, ISR.code, ed, ei, ni, ir, rs, sp and na bits are 0.

31–24	23–16	15–4	3–0
0	0	0	code{3:0}

63–44	43	42–41	40	39	38	37	36	35	34	33	32
0	ed	ei	so	ni	ir	rs	sp	na	r	w	0

If the fault is due to an instruction page not present fault:

- IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.
- ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below. For IA-32 memory references the ISR.ei and ni bits are 0.

31–24	23–16	15–0
0	0	0

63–44	43	42–41	40	39	38	37	36	35	34	33	32
0	0	ei	0	ni	0	0	0	0	0	0	1

Notes This fault can only occur when PSR.it is 1 on an instruction reference, when PSR.dt is 1 on a load, store, semaphore, or non-access operation, or when PSR.rt is 1 on a RSE mandatory load/store operation.

Name **Key Permission vector (0x5100)**

Cause Data access (including IA-32): The PSR.dt bit is 1, the PSR.pk bit is 1 and read or write permission is disabled by the matching protection register on a load, store, or semaphore operation. The RSE may cause this fault if PSR.rt is 1, the PSR.pk bit is 1 and read or write permission is disabled by the matching protection register on an RSE mandatory load/store operation. Instruction access (including IA-32): The PSR.it bit is 1, the PSR.pk bit is 1 and execute permission is disabled by the matching protection register.

Interruptions on this vector:

IR Data Key Permission fault
Instruction Key Permission fault
Data Key Permission fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register.The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

If the fault is due to a data key permission fault:

- IFA – Faulting data address.
- ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0. For IA-32 memory references, the ISR.code, ed, ei, ni, ir, rs, sp bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

If the fault is due to an instruction key permission fault:

- IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.
- ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below. For IA-32 memory references, ISR.ei and ni are set to 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes For `probe.fault` or `lfetch.fault` the ISR.na bit is set.

Name **Instruction Access Rights vector (0x5200)**

Cause For instruction fetches (including IA-32), the PSR.it bit is 1, and the access rights for this page do not allow execution or do not allow execution at the current privilege level.

Interruptions on this vector:

Instruction Access Rights fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below. For IA-32 memory references, ISR.ei and ni bits are 0.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">0</td><td colspan="8">0</td><td colspan="16">0</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">0</td><td>0</td><td colspan="2">ei</td><td>0</td><td>ni</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
</table>

Notes This fault does not occur if PSR.it is 0.

Name **Data Access Rights vector (0x5300)**

Cause For memory references (including IA-32), the PSR.dt bit is 1, and the access rights for this page do not allow read access or do not allow read access at the current privilege level for load and semaphore operations. The PSR.dt bit is 1, and the access rights for this page do not allow write access or do not allow write access at the current privilege level for store and semaphore operations.

The PSR.rt bit is 1, and the access rights for this page do not allow read access or do not allow read access at the current privilege level for the RSE mandatory load operation. The PSR.rt bit is 1, and the access rights for this page do not allow write access or do not allow write access at the current privilege level for the RSE mandatory store operation.

Interruptions on this vector:

IR Data Access Rights fault
Data Access Rights fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – Faulting data address.

ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0. For IA-32 memory references, ISR.code, ed, ei, ni, ir, rs, and sp bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

Notes For `probe.fault` or `lfetch.fault` the ISR.na bit is set.

Name **General Exception vector (0x5400)**

Cause An attempt is being made to execute an illegal operation, privileged instruction, access a privileged register, unimplemented field, unimplemented register, unimplemented address, or take an inter-instruction set branch when disabled.

Interruptions on this vector:

IR Unimplemented Data Address fault
Illegal Operation fault
Illegal Dependency fault
Privileged Operation fault
Disabled Instruction Set Transition fault
Reserved Register/Field fault
Unimplemented Data Address fault
Privileged Register fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. For IA-32 instruction set faults, ISR.ei, ni, na, sp, rs, ir, ed bits are always 0.

- If the fault was caused by a non-access instruction, ISR.code{3:0} specifies which non-access instruction. See “Non-access Instructions and Interruptions” on page 2:87.
- ISR.code{7:4} = 0: Illegal Operation fault. Cannot be raised by IA-32 instructions.
    - An attempt is being made to execute an illegal operation. Illegal operations include:
        - Attempts to execute instructions containing reserved major opcodes, reserved sub-opcodes, or reserved instruction fields, writing GR 0, FR 0 or FR 1, writing a read-only register, or accessing a reserved register.
        - Attempts to execute a reserved template encoding. An `rfi` to a reserved template encoding preserves IPSR.ri and will set ISR.ei to IPSR.ri.
        - Attempts to execute a bundle of template MLX when PSR.ri == 2. This can only be caused by doing an `rfi` with an improper setting of IPSR.ri. In this case, IPSR.ri and ISR.ei will both be 2.
        - Attempts to write outside the current register stack frame.
        - Attempts to specify the same GR, when the instruction has two GR targets (e.g., post-increment).
        - If the instruction has two PR targets, and specifies the same PR for both. Predicated off unconditional compares, `fclass`, `tbit`, and `tnat` instructions take this fault, even when their qualifying predicate is zero.
        - Register bank conflict on a floating-point load pair instruction.
        - An access to BSPSTORE or RNAT is performed with a non-zero RSC.mode, or a `loadrs` is performed with a non-zero RSC.mode.
        - A `loadrs` is performed with a non-zero CFM.sof and a non-zero RSC.loadrs, or a `loadrs` causes more registers to be loaded from memory than can fit in the physical stacked register file.
        - Attempts to predicate a `br.ia` instruction or to execute `br.ia` when AR[BSPSTORE] != AR[BSP].
        - Attempts to execute `epc` if PFS.ppl is less than PSR.cpl.
        - Attempts to access interruption registers if PSR.ic is 1.

- Attempts to execute an `itc` or `itr` instruction if PSR.ic is 1.

- ISR.code{7:4} = 1: Privileged Operation fault. Cannot be raised by IA-32 instructions.
- ISR.code{7:4} = 2: Privileged Register fault. Cannot be raised by IA-32 instructions.
- ISR.code{7:4} = 3: Reserved Register/Field fault, Unimplemented Data Address fault or IR Unimplemented Data Address fault. Cannot be raised by IA-32 instructions. For Unimplemented Data Address fault:
  - If ISR.rs = 0: A data memory reference to an unimplemented address has occurred.
  - If ISR.rs = 1: A mandatory RSE reference to an unimplemented address has occurred.

  For details, refer to "Reserved and Ignored Registers and Fields" on page 2:19 and "Unimplemented Address Bits" on page 2:61.
- ISR.code{7:4} = 4: Disabled Instruction Set Transition fault. An instruction set transition was attempted while PSR.di was 1. This fault can be raised by either the Itanium `br.ia` instruction or the IA-32 `jmpe` instruction. IPSR.is indicates the faulting instruction set.
- ISR.code{7:4} = 8: Illegal Dependency fault. Cannot be raised by IA-32 instructions. The processor has detected a resource dependency violation.

If the fault is due to an Illegal Operation fault or Illegal Dependency fault:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0								code{7:4}				0			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	0

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0								code{7:4}				code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	ir	rs	0	na	r	w	0

Name

Disabled FP-Register vector (0x5500)

Cause

An attempt is made to reference a floating-point register set that is disabled.

When PSR.dfl is 1, execution of any IA-32 FP, SSE or MMX instructions raises a Disabled FP Register Low Fault (regardless of whether FR2 - FR31 are actually referenced).

When PSR.dfh is 1, execution of the first IA-32 instruction following a `br.ia` or `rfi` raises a Disabled FP Register High fault.

If concurrent IA-32 Disabled FP Register High and Low faults are generated, the Disabled FP Register High fault takes precedence and is reported in the ISR code, the Disabled FP Register Low fault is discarded and not reported in the ISR code.

Interruptions on this vector:

Disabled Floating-Point Register fault

Parameters

IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ISR – The defined ISR bits are specified below.

- ISR.code{0} = 1: FR2 - FR31 disabled and access attempted.
- ISR.code{1} = 1: FR32 - FR127 disabled and access attempted.

For IA-32 references, ISR.ei, ni, sp, r, and w bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0														code	

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	sp	0	r	w	0

Name **NaT Consumption vector (0x5600)**

Cause A non-speculative operation (including IA-32) (e.g., load, store, control register access, instruction fetch etc.) read a NaT source register, NaTVal source register, or referenced a NaTPage.

Interruptions on this vector:

IR Data NaT Page Consumption fault
Instruction NaT Page Consumption fault
Register NaT Consumption fault
Data NaT Page Consumption fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

If the fault is due to a Data NaT Page Consumption fault or an IR Data NaT Page Consumption fault:

A non-speculative Itanium integer/FP instruction or instruction fetch or IA-32 data memory reference accessed a page with the NaTPage memory attribute.

- IFA – faulting data address.
- ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0. For the IA-32 instruction set, ISR.ed, ei, ni, ir, rs and na bits are 0. For `probe.fault` or `lfetch.fault` the ISR.na bit is set.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0								2				code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		so	ni	ir	rs	0	na	r	w	0

If the fault is due to an Instruction NaT Page Consumption fault:

A non-speculative Itanium integer/FP instruction or instruction fetch accessed a page with the NaTPage memory attribute.

- IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.
- ISR – The value for the ISR bits depend on the type of access performed and are specified below. For the IA-32 instruction set, ISR.ni and ei bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0								2				0			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

If the fault is due to an Register NaT Consumption fault:

A non-speculative Itanium instruction reads a NaT'ed GR or an FR containing NaTVal. An IA-32 integer instruction reads a NaT'ed GR. For IA-32 instructions behavior of NaT and NaTVal values is model specific, see Section 6.4.3, "NaT/NaTVal Response for IA-32 Instructions" on page 1:126 for details.

- ISR – The value for the ISR bits depend on the type of access performed and are specified below. For the IA-32 instruction set, ISR.ed, ei, ni, ir, rs, r, w, and na bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0								1				code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	na	r	w	0

Name **Speculation vector (0x5700)**

Cause A `chk.a`, `chk.s`, or `fchkf` instruction needs to branch to recovery code, and the branching behavior is unimplemented by the processor. This fault cannot be raised by IA-32 instructions.

Interruptions on this vector:

Speculative Operation fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

IIM – contains the immediate value from the `chk.s`, `chk.a`, or `fchkf` instruction.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The type of instruction which caused the fault is encoded in the lower four bits of the ISR.code field.

- If ISR.code{3:0} = 0: `chk.a` general register speculation fault.
- If ISR.code{3:0} = 1: `chk.s` general register speculation fault.
- If ISR.code{3:0} = 2: `chk.a` floating-point speculation fault.
- If ISR.code{3:0} = 3: `chk.s` floating-point speculation fault.
- If ISR.code{3:0} = 4: `fchkf` fault.

The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	0

Name **Debug vector (0x5900)**

Cause A debug fault has occurred. Either the instruction address matches the parameters set up in the instruction debug registers, or the data address of a load, store, semaphore, or mandatory RSE fill or spill matches the parameters set up in the data debug registers. All IA-32 instruction set debug events are delivered on the IA_32_Exception(Debug) vector; see Chapter 9, "IA-32 Interruption Vector Descriptions". IA-32 instructions can not raise this fault, IA-32 debug events are delivered on the IA-32_Exception(Debug) vector.

Interruptions on this vector:

IR Data Debug fault
Instruction Debug fault
Data Debug fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

If the fault is due to a data debug fault or an IR Data Debug fault:

- IFA – The address of the data being referenced.
- ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		0	ni	ir	rs	sp	na	r	w	0

If the fault is due to an instruction debug fault:

- IFA – Faulting instruction fetch address.
- ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes On an instruction reference this fault is suppressed if the PSR.db bit is 0 or if the PSR.id bit is 1. On a data reference this fault is suppressed if the PSR.db bit is 0 or if the PSR.dd bit is 1. The only non-access data operations which can cause a debug fault are the faulting variants of `lfetch` and `probe`.

Name **Unaligned Reference vector (0x5a00)**

Cause If PSR.ac is 1, and the data address being referenced by an Itanium instruction is not aligned to the natural size of the load, store, or semaphore operation, or a data reference is made to a misaligned datum not supported by the implementation. See “Memory Access Instructions” on page 2:51. For IA-32 data memory references, an IA_32_Exception(Alignment Check) fault is raised; see Chapter 9, "IA-32 Interruption Vector Descriptions". IA-32 instructions can not raise this fault, IA-32 unaligned events are delivered on the IA-32_Exception(Alignment_Check) vector.

If the data reference specified is both unaligned to the natural datum size and unsupported, then an Unaligned Data Reference fault is taken.

Interruptions on this vector:

Unaligned Data Reference fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

IFA – The address of the data being referenced.

ISR – The value for the ISR bits depend on the type of access performed and are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		0	ni	0	0	sp	0	r	w	0

Name **Unsupported Data Reference vector (0x5b00)**

Cause An attempt was made to:

- Execute a `fetchadd`, `cmpxchg`, `xchg`, or unsupported 10-byte memory reference (`ldfe` or `stfe`) instruction to a page that is neither cacheable with write-back write policy nor a NaTPage.
- Execute a `fetchadd` instruction to a page that is an uncacheable exported (UCE) page and the processor model does not support exporting of `fetchadd` instructions.

See “Effects of Memory Attributes on Memory Reference Instructions” on page 2:72 for details. IA-32 instructions can not raise this fault, IA-32 locked faults are delivered on the IA-32_Intercept(Lock) vector.

If the data reference specified is both unaligned to the natural datum size and unsupported, then an Unaligned Data Reference fault is taken.

IA-32 data memory references that require an external atomic lock when DCR.lc is 1, raise an IA_32_Intercept(Lock) fault; see Chapter 9, "IA-32 Interruption Vector Descriptions".

Interruptions on this vector:

Unsupported Data Reference fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

IFA – The address of the data being referenced.

ISR – The value for the ISR bits depend on the type of access performed and are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		0	ni	0	0	0	0	r	w	0

For `ldfe` and `stfe` instructions, the processor may optionally set both ISR.r and ISR.w to 1, although this is not recommended.

Name **Floating-point Fault vector (0x5c00)**

Cause A floating-point exception fault has occurred. IA-32 numeric instructions can not raise this fault, IA-32 floating point faults are delivered on the IA-32_Exception(Floating-Point) vector.

Interruptions on this vector:

Floating-Point Exception fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception.

ISR.code contains information about the FP exception fault. The ISR.code field has eight bits defined. See Chapter 5 for details.

- ISR.code{0} = 1: IEEE V (invalid) exception (Normal or Parallel FP-HI)
- ISR.code{1} = 1: Denormal/Unnormal operand exception (Normal or Parallel FP-HI)
- ISR.code{2} = 1: IEEE Z (divide by zero) exception (Normal or Parallel FP-HI)
- ISR.code{3} = 1: Software assist (Normal or Parallel FP-HI)
- ISR.code{4} = 1: IEEE V (invalid) exception (Parallel FP-LO)
- ISR.code{5} = 1: Denormal/Unnormal operand exception (Parallel FP-LO)
- ISR.code{6} = 1: IEEE Z (divide by zero) exception (Parallel FP-LO)
- ISR.code{7} = 1: Software assist (Parallel FP-LO)

The defined ISR bits are specified below:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0								code{7:0}							

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	0

Name **Floating-point Trap vector (0x5d00)**

Cause A floating-point exception trap has occurred. IA-32 numeric instructions can not raise this trap.

Interruptions on this vector:

Floating-Point Exception trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception.

ISR.code contains information about the type of FP exception and IEEE information. The ISR code field contains a bit vector (see Table 8-3 on page 2:151) for all traps which occurred in the just-executed instruction. The defined ISR bits are specified below:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0	fp trap code								0	0	0	ss	0	0	1

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	0

Name **Lower-Privilege Transfer Trap vector (0x5e00)**

Cause Two trapping conditions transfer control to this vector:

- An attempt is made to execute an instruction at an unimplemented address, resulting in an Unimplemented Instruction Address trap. See “Unimplemented Address Bits” on page 2:61.
- The PSR.lp bit is 1, and a branch lowers the privilege level.

IA-32 instructions can not raise this trap.

Interruptions on this vector:

Unimplemented Instruction Address trap
Lower-Privilege Transfer trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

Note: Please see “Interruption Instruction Bundle Pointer (IIP – CR19)” on page 2:30 for a further clarification of the IIP value for an unimplemented instruction address trap.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The ISR.code contains a bit vector (see Table 8-3 on page 2:151) for all traps which occurred in the just-executed instruction. The defined ISR bits are specified below.

If the trap is due to an Unimplemented Instruction Address trap:

31–24	23–16	15	14–7	6	5	4	3	2	1	0
0	0	0	fp trap code	0	0	1	ss	tb	lp	fp

63–44	43	42–41	40	39	38	37	36	35	34	33	32
0	0	ei	0	ni	ir	0	0	0	0	0	0

If the trap is due to a Lower-Privilege Transfer trap:

31–24	23–16	15–7	6	5	4	3	2	1	0
0	0	0	0	0	0	ss	tb	1	0

63–44	43	42–41	40	39	38	37	36	35	34	33	32
0	0	ei	0	ni	ir	0	0	0	0	0	0

Notes The Unimplemented Instruction Address trap can be the result of an inline instruction fetch, a taken or not-taken branch or an `rfi`. The lower privilege transfer trap is only taken on a branch demotion, and not an `rfi` return.

Name **Taken Branch Trap vector (0x5f00)**

Cause A taken branch was executed, and the PSR.tb bit is 1. IA-32 instructions can not raise this trap, IA-32 taken branch traps are delivered on the IA-32_Exception(Debug) vector.

The Taken Branch trap is not taken on an `rfi` instruction.

Interruptions on this vector:

Taken Branch trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

Note: Please see "Interruption Instruction Bundle Pointer (IIP – CR19)" on page 2:30 for a further clarification of the IIP value for an unimplemented instruction address trap.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The ISR.code contains a bit vector (see Table 8-3 on page 2:151) for all traps which occurred in the just-executed instruction. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0									0	0	0	ss	1	0	0

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	ir	0	0	0	0	0	0

Name **Single Step Trap vector (0x6000)**

Cause An instruction was successfully executed, and the PSR.ss bit is 1. For IA-32 instruction set, this condition is delivered on the IA_32_Exception(Debug) vector; see Chapter 9, "IA-32 Interruption Vector Descriptions". IA-32 instructions can not raise this trap, IA-32 single step events are delivered on the IA-32_Exception(Debug) vector.

The Single Step trap is not taken on an `rfi` instruction.

Interruptions on this vector:

Single Step trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The ISR.code contains a bit vector (see Table 8-3 on page 2:151) for all traps which occurred in the just-executed instruction. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0									0	0	0	1	0	0	0

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	ir	0	0	0	0	0	0

Name **IA-32 Exception vector (0x6900)**

Cause A fault or trap was raised while executing from the IA-32 instruction set.

Interruptions on this vector:

IA-32 Instruction Debug fault
IA-32 Code Fetch fault
IA-32 Instruction Length > 15 bytes fault
IA-32 Device Not Available fault
IA-32 FP Error fault
IA-32 Segment Not Present fault
IA-32 Stack Exception fault
IA-32 General Protection fault
IA-32 Divide by Zero fault
IA-32 Alignment Check fault
IA-32 Bound fault
IA-32 INTO trap
IA-32 Breakpoint (INT 3) trap
IA-32 Data Breakpoint trap
IA-32 Taken Branch trap
IA-32 Single Step trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

IFA – is undefined. The faulting IA-32 address is contained in IIPA.

ISR – ISR.vector contains the IA-32 exception vector number. ISR.code contains the IA-32 error code for faults or a trap code listing concurrent trap events for traps.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								vector								error_code/trap_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	0		0	0	0	0	0	0	0	0	x

Notes See Chapter 9, "IA-32 Interruption Vector Descriptions" for complete details on each IA-32 Exception and for error code and trap code definition.

Name **IA-32 Intercept vector (0x6a00)**

Cause An intercept fault or trap was raised while executing from the IA-32 instruction set. This vector handles all the IA-32 intercepts described in Chapter 9, "IA-32 Interruption Vector Descriptions".

Interruptions on this vector:

IA-32 Invalid Opcode fault
IA-32 Instruction Intercept fault
IA-32 Locked Data Reference fault
IA-32 System Flag Intercept trap
IA-32 Gate Intercept trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

IIM – 64-bit information describing the cause of the intercept.

ISR – ISR.vector contains a number specifying the type of intercept. ISR.code contains the IA-32 specific intercept information or a trap code listing concurrent trap events for traps.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								intercept_number								intercept_code/trap_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	0		0	0	0	0	0	0	r	w	0

Notes See Chapter 9, "IA-32 Interruption Vector Descriptions" for complete details on each IA-32 Intercept and for the intercept code and trap code definition.

Name **IA-32 Interrupt vector (0x6b00)**

Cause An IA-32 software interrupt trap was executed. This vector handles all the IA-32 software interrupts described in Chapter 9, "IA-32 Interruption Vector Descriptions".

Interruptions on this vector:

IA-32 Software Interrupt (INT) trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:147 for a detailed description.

ISR – ISR.vector contains the IA-32 defined interruption vector number. ISR.code contains a trap code listing concurrent trap events.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								vector								trap_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	0		0	0	0	0	0	0	0	0	0

Notes See Chapter 9, "IA-32 Interruption Vector Descriptions" for complete details on this vector and the trap code definition.

IA-32 Interruption Vector Descriptions 9

This section gives detailed description of all possible IA-32 exceptions, interrupts and intercepts that can occur during IA-32 instruction set execution in the Itanium System Environment. Interruption resources not noted below are undefined after the interruption. For all cases where an interruption is taken out of the IA-32 instruction set, IPSR.is is set to 1.

9.1 IA-32 Trap Code

The following trap code is defined for concurrent traps reported during IA-32 instruction set execution. There is a bit for every possible concurrent trap condition.

Figure 9-1. IA-32 Trap Code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								b3	b2	b1	b0	ss	tb	0	

Figure 9-2. IA-32 Trap Code

Bit	Name	Description
2	tb	taken branch trap, set if an IA-32 branch is taken and branch traps are enabled (PSR.tb is 1).
3	ss	single step trap, set after the successful execution of every IA-32 instruction if PSR.ss or EFLAG.tf is 1.
4-7	b0 to b3	Data breakpoint trap due to a match with the corresponding Intel® Itanium™ data breakpoint registers. Each bit indicates a match with the corresponding DBR registers; b0=DBR0/1, b1=DBR2/3, b2=DBR4/5, b3=DBR6/7. Zero, one or more bits may be set. These bits accumulate data breakpoint register matches that occurred during the duration of executing one IA-32 instruction. In order to be reported, the DBR register address and mask registers must precisely match the IA-32 data memory reference address, and the DBR read, write bits match the type of memory transaction, and the DBR privilege level mask match the value in PSR.cpl.

9.2 IA-32 Interruption Vector Definitions

Following are the definitions of IA-32 exceptions, interrupts and intercepts that can occur during IA-32 instruction set execution in the Itanium system environment.

Name **IA_32_Exception (Divide) - Divide Fault**

Cause IA-32 IDIV or DIV instruction attempted a divide by zero operation. Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this fault.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector - 0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (Debug) - Code Breakpoint Fault**

Cause The Itanium architecture debug facilities triggered an IA-32 code breakpoint fault on a IA-32 instruction fetch and PSR.id and EFLAG.rf are 0. Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this fault.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector - 1

ISR.x - 1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								1								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	1

Name **IA_32_Exception (Debug) - Data Breakpoint, Single Step, Taken Branch Trap**

Cause The Itanium architecture debug facilities triggered an IA-32 data breakpoint, single-step or branch trap. In the Itanium System Environment, IA-32 Mov SS or Pop SS single step and data breakpoint traps are NOT deferred to the next instruction. Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this trap.

Parameters IIPA - virtual address of the trapping IA-32 instruction (zero extended to 64-bits) if there was a taken branch trap. Otherwise, if there was no taken branch trap (data breakpoint and/or single step) IIPA is set to the same value as IIP.

IIP - next Itanium instruction address or the virtual IA-32 instruction address zero extended to 64-bits.

ISR.vector - 1

ISR.code - Trap Code, indicates Concurrent Single Step, Taken Branch, Data Breakpoint Trap events

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								1								trap_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (Break) - INT 3 Trap**

Cause IA-32 breakpoint instruction (INT 3) triggered a trap. Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this trap.

Parameters IIPA - trapping virtual IA-32 instruction address zero extended to 64-bits

IIP - next virtual IA-32 instruction address zero extended to 64-bits

ISR.vector - 3

ISR.code -Trap Code, indicates Concurrent Single Step condition

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								3								trap_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (Overflow) - Overflow Trap**

Cause IA-32 INTO instruction execution when EFLAG.of is set to one. Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this trap.

Parameters IIPA - trapping virtual IA-32 instruction address zero extended to 64-bits

IIP - next virtual IA-32 instruction address zero extended to 64-bits

ISR.vector - 4

ISR.code - Trap Code, indicates Concurrent Single Step

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								4								trap_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (Bound) - Bounds Fault**

Cause Failed IA-32 Bound check instruction. Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this fault.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector - 5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								5								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (InvalidOpcode) - Invalid Opcode Fault**

Cause All IA-32 invalid opcode faults are delivered to the IA-32_Intercept(Instruction) handler, including IA-32 illegal, unimplemented opcodes, MMX technology and Streaming SIMD Extension instructions if CR0.EM is 1, and Streaming SIMD Extension instructions if CR4.fxsr is 0. All illegal IA-32 floating-point opcodes result in an IA-32_Intercept(Instruction) regardless of the state of CR0.em.

Name **IA_32_Exception (DNA) - Device Not Available Fault**

Cause The processor executed an IA-32 ESC or floating-point instruction with CR0.em is 1. Or an IA-32 WAIT, ESC, floating-point instruction, MMX technology or Streaming SIMD Extension instruction is executed and CR0.ts bit is 1.

Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this fault.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector - 7

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								7								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **Double Fault**

Cause IA-32 Double Faults (IA-32 vector 8) are not generated by the processor in the Itanium System Environment.

Name **Invalid TSS Fault**

Cause IA-32 Invalid TSS Faults (IA-32 vector 10) are not generated in the Itanium System Environment.

Name **IA_32_Exception (NotPresent) - Segment Not Present Fault**

Cause Generated when the processor detects the Present-bit of the memory segment descriptor is zero during an IA-32 segment load or far control transfer instructions. Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this fault and error codes.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of **t**he faulting IA-32 instruction zero extended to 64-bits.

ISR.vector - 11

ISR.code - IA-32 defined error code. See *Intel Architecture Software Developer's Manual*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								11								error_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (StackFault) - Stack Fault**

Cause IA-32 defined set of stack segment fault conditions detected during stack segment load operations or memory references relative to the stack segment, refer to the *Intel Architecture Software Developer's Manual* for a complete list of all IA-32 faulting conditions. Stack faults can also be generated when the processor detects an inconsistent stack segment register descriptor value during an IA-32 stack reference instruction (e.g. PUSH, POP, CALL, RET,). See section "Segment Descriptor and Environment Integrity" for a list of possible inconsistent register descriptor conditions.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector - 12

ISR.code - IA-32 defined ErrorCode. Zero if an inconsistent register descriptor is detected during a memory reference relative to the stack segment.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								12								error_code or zero															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (GPFault) - General Protection Fault**

Cause IA-32 defined set of data and code segment fault conditions detected during data or code segment load operations or memory references relative to code or data segments, refer to the *Intel Architecture Software Developer's Manual* for a complete list of all IA-32 General Protection Fault conditions. General Protection faults can also be generated when the processor detects an inconsistent code or data segment register descriptor value during an IA-32 code fetch or data memory reference. See section "Segment Descriptor and Environment Integrity" for a list of possible inconsistent register descriptor conditions.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector - 13

ISR.code-IA-32 defined ErrorCode. Zero if an inconsistent register descriptor is detected during a memory reference relative to a code or data segment.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								13								error_code or zero															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **Page Fault**

Cause IA-32 defined page faults (IA-32 vector 14) can not be generated in the Itanium System Environment.

Name **IA_32_Exception (FPError) -Pending Floating-point Error**

Cause An unmasked IA-32 floating-point exception is delivered on the next non-control IA-32 floating-point, MMX technology, WAIT, or `jmpe` instruction trigger delivery of this exception. Floating-point errors are delivered regardless of the state of CR0.ne in the Itanium System Environment. IA-32 numeric exception delivery is not triggered by Itanium numeric exceptions or the execution of Itanium numeric instructions. Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this fault.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of the faulting IA-32 instruction zero extended to 64-bits.

FSR, FIR, FDR and FCR contain the IA-32 floating-point environment and exception information

ISR.vector - 16

.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								16								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (AlignmentCheck) - Alignment Check Fault**

Cause An IA-32 instruction performed an unaligned data memory reference while PSR.ac is 1, or EFLAG.ac is 1 and CR0.am is 1 and the effective privilege level is 3. Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this fault.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of the faulting IA-32 instruction zero extended to 64-bits.

IFA - referenced virtual data address (byte granular) zero extended to 64-bits

ISR.vector - 17

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								17								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **Machine Check**

Cause IA-32 Machine Check (IA-32 vector 18) is not generated in the Itanium System Environment.

Name **IA_32_Exception (StreamingSIMD) -Streaming SIMD Extension Numeric Error Fault**

Cause An unmasked IA-32 Streaming SIMD Extension numeric error occurred. Numeric faults generated on Streaming SIMD Extension instructions are reported precisely on the faulting Streaming SIMD Extension instruction. Streaming SIMD Extension instructions do NOT trigger the report of any pending IA-32 floating-point exceptions. Streaming SIMD Extension instructions always ignore CR0.ne and the IGNNE pin. Refer to the *Intel Architecture Software Developer's Manual* for a complete definition of this fault.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector - 19

.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								19								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Interrupt (Vector #N) - Software Trap**

Cause The IA-32 INT n instruction forces an IA-32 interrupt trap. The IA-32 IDT is not consulted nor are any values pushed onto a memory stack.

Parameters IIPA - trapping virtual IA-32 instruction address (points to the INT instruction) zero extended to 64-bits

IIP - next virtual IA-32 instruction address zero extended to 64-bits

ISR.vector - vector number

ISR.code - TrapCode, Indicates Concurrent Single Step Trap condition

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								vector								trap_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Intercept (Instruction) - Instruction Intercept Fault**

Cause Execution of unimplemented IA-32 opcodes, illegal opcodes or sensitive privileged IA-32 operating system instructions results in an instruction intercept. Intercepted opcodes include (but are not limited to); CLTS, HLT, INVD, INVLPG, IRET, LIDT, LGDT, LLDT, LMSW, LTR, MOV to CRs, MOV to/from DRs, RDMSR, RSM, SIDT, SGDT, SLDT, SMSW, WBINVD, WRMSR, and all other unimplemented and illegal opcode patterns. If CR0.em is 1, execution of all IA-32 MMX technology and IA-32 Streaming SIMD Extension instructions results in this intercept. If CR4.FXSR is 0, execution of all IA-32 Streaming SIMD Extension instructions results in this intercept. All illegal IA-32 floating-point opcodes result in an IA-32_Intercept(Instruction) regardless of the state of CR0.em. Intercepted opcodes are nullified and alter no architectural state.

Parameters IIP - virtual IA-32 instruction address zero extended to 64-bits, points to the first byte of the intercepted IA-32 opcode (including prefixes).

IIPA -virtual address of the faulting IA-32 instruction zero extended to 64-bits.

IIM - Opcode bytes, contains the first 8-bytes of the IA-32 instruction following all prefix bytes. All prefix bytes are decoded and presented as a bitmask in the Intercept Code along with the prefix length in bytes. Opcode bytes are loaded into IIM in the same format as encountered in memory and as defined in the *Intel Architecture Software Developer's Manual*. The lowest memory address byte is placed in byte 0 of IIM, higher memory address bytes are placed in increasingly higher numbered bytes within IIM.

The 8-byte opcode loaded into IIM is stripped of the following prefixes; lock, repeat, address size, operand size, and segment override prefixes (opcode bytes 0xF3, 0xF2, 0xF0, 0x2E, 0x36, 0x3E, 0x26, 0x64, 0x65, 0x66, and 0x67). The 0x0F opcode series prefix is not stripped from the opcode bytes loaded into IIM. The opcode loaded into IIM includes all IA-32 opcode components, including 1 to 3 bytes of opcode, mod r/m bytes, sib bytes and any possible immediates and/or displacements.

If the opcode loaded in IIM is less than 8-bytes, the remainder higher order numbered bytes are set to 0. If the opcode is larger than 8-bytes, bytes after the 8th byte (following all stripped prefixes) are not reported. If required, emulation code must retrieve the extra opcode bytes by reading from the memory locations specified by IIP.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
byte3								byte2								byte1								byte0							

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
byte7								byte6								byte5								byte4							

ISR.vector - 0, indicates instruction intercept.

ISR.code - Intercept Code indicates prefixes and prefix lengths.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								0								intercept_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Figure 9-3 defines intercept codes for IA-32 instruction set intercepts. Intercept code fields are defined by Table 9-1 and Table 9-2 on page 2:210.

Figure 9-3. IA-32 Intercept Code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
len				0		seg			sp	np	rp	lp	as	os	0

Table 9-1. Intercept Code Definition

Bit	Name	Description
1	os	Operand Size - (OperandSize Prefix XOR CSD.d bit). When 1, indicates the effective operand size is 32-bits, when 0, 16-bits.
2	as	Address Size - (AddressSize Prefix XOR CSD.d bit). When 1, indicates the effective address size is 32-bits, when 0, 16-bits.
3	lp	Lock Prefix - If 1, indicates a lock prefix is present.
4	rp	REP or REPE/REPZ Prefix - If 1, indicates a REP/REPE/REPZ prefix is in effect.
5	np	REPNE/REPNZ Prefix - If 1, indicates a REPNE/REPNZ prefix is in effect.
6	sp	Segment Prefix - If 1, indicates a Segment Override prefix is present.
7:9	seg	Segment Value - Segment Prefix Override value, see Figure 9-2 for encodings. If there is no segment prefixes this field is undefined.
12:15	len	Length of Prefixes - Length of all prefix (in bytes) stripped from IIM. If there are no prefixes this field has a value of zero.

Table 9-2. Segment Prefix Override Encodings

Seg Value	Segment Prefix
0	ES Segment Override
1	CS Segment Override
2	SS Segment Override
3	DS Segment Override
4	FS Segment Override
5	GS Segment Override
6	reserved
7	reserved

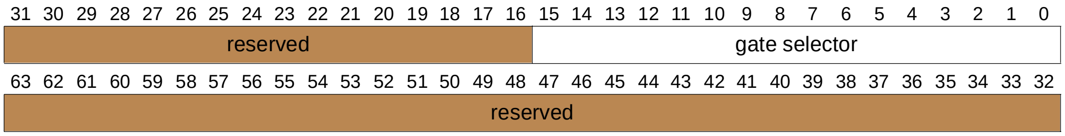
Name **IA_32_Intercept (Gate) - Gate Intercept Trap**

Cause If an IA-32 control transfer is initiated through a GDT/LDT descriptor that transfers control through a Call Gate, Task Gate or Task Segment this interception trap is generated.

Parameters IIPA - trapping virtual IA-32 instruction address zero extended to 64-bits

IIP - next sequential virtual IA-32 instruction address zero extended to 64-bits

IFA - Gate Selector. The gate selector is loaded in IFA{15:0}.



IIM - Gate, Task Gate or Task Segment Descriptor. The descriptor loaded in IIM adheres to the IA-32 GDT/LDT memory format, where byte 0 of the descriptor is in IIM{7:0}.

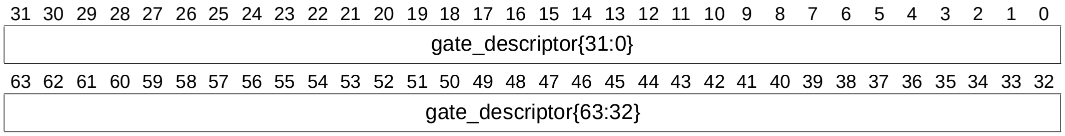


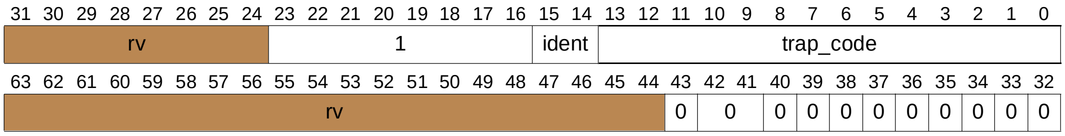
Table 9-3. Gate Intercept Trap Code Identifier

Instruction	ISR.code{15:14}
CALL	00
JMP	01

ISR.vector - 1, indicates gate interception.

ISR.code - TrapCode, Indicates Concurrent Data Debug, taken Branch, and Single Step Events

ISR.code{15:14} - indicates whether CALL or JMP generated the trap. See Table 9-3 for details.



Name **IA_32_Intercept (SystemFlag) - System Flag Trap**

Parameters System Flag Intercept Traps are generated for the following conditions:

CLI, STI, POPF, POPFD instructions. If the EFLAG.if bit changes state and CFLG.ii is 1, or EFLAG.tf or EFLAG.ac change state, a System Flag intercept notification trap is delivered after the instruction completes. IIM contains the previous value of EFLAG before the trapping instruction executed. If IA-32 code does not have IOPL or CPL permission to modify the EFLAG bits, no intercept is generated. This intercept trap condition can be used to provide virtual interrupt services, and delay enabling of interrupts after the STI instruction.

MOV SS, POP SS instructions. After these instructions complete execution, a System Flag intercept notification trap is delivered. This intercept trap condition can be used to inhibit interrupts, and code breakpoints between Mov/Pop SS and the next instruction and to inhibit Single Step and Data Breakpoint traps on the Mov, or Pop SS instruction.

IIP - next virtual IA-32 instruction address zero extended to 64-bits

IIPA - trapping virtual IA-32 instruction address zero extended to 64-bits

IIM - contains the previous EFLAG value before the trapping instruction

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">old EFLAG</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="32">reserved</td></tr>
</table>

ISR.vector - 2

ISR.code - Trap Code, indicates Concurrent Single Step Trap, Debug trap condition.

ISR.code{15:14} indicates which instruction generated the trap.

Table 9-4. System Flag Intercept Instruction Trap Code Instruction Identifier

Instruction	ISR.code{15:14}
CLI	00
STI	01
POPF, POPFD	10
MOV/POP SS	11

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">rv</td><td colspan="8">2</td><td colspan="2">ident</td><td colspan="14">trap_code</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">rv</td><td>0</td><td colspan="2">0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Name **IA_32_Intercept (Lock) - Locked Data Reference Fault**

Cause For IA-32 locked operations, if the DCR.lc bit is 1, and an atomic operation to made to non-write-back memory or to unaligned write-back memory that would result in a read-modify-write sequence being performed externally under an external bus lock, the processor raises a Locked Data Reference fault.

Parameters IIP - faulting virtual IA-32 instruction address zero extended to 64-bits

IIPA - virtual address of the faulting IA-32 instruction zero extended to 64-bits

IFA - faulting virtual data address (byte granular) zero extended to 64-bits

ISR.vector - 4

ISR.code - 0

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
rv	4	0

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44	43	42 41	40	39	38	37	36	35	34	33	32
rv	0	0	0	0	0	0	0	0	1	1	0

Itanium™-based Operating System Interaction Model with IA-32 Applications 10

This section describes the IA-32 system execution model from the perspective of an Itanium-based operating system interfacing with IA-32 code, while operating in the Itanium System Environment. The main features covered are:

- IA-32 system and control register behavior
- IA-32 virtual memory support
- IA-32 fault and trap handling
- IA-32 instruction behavior

10.1 Instruction Set Transitions

Instruction set transitions are defined in "Instruction Set Modes". 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` to IA-32 target results in a Disabled Instruction Set Transition Fault, and the operation is nullified.

The processor also transitions into an Itanium-based operating system when IA-32 privileged system resources are accessed, on an interruption, or when the following conditions are detected:

- Instruction Interception - IA-32 system level privileged instructions are executed
- System Flag Interception - Various EFLAG system flags are modified, (e.g. AC, TF and IF-bits)
- Gate Interception - control transfers are made through call gate, or transfers through a task switch (TSS segment or Task Gate).

All software interrupts, external interrupts, faults, traps and machine checks transition the processor to the Itanium instruction set, regardless of the state of PSR.di. IA-32 defined exceptions and software interrupts are delivered to Itanium-based interruption handlers.

10.2 System Register Model

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 software calling conventions for these registers.

- **Shared**: Shared registers contain values that have similar functionality in either instruction set. For example, all Itanium control registers, debug registers are used for memory references (including IA-32). The stack pointer (ESP) and instruction pointer (IP) are also shared.
- **Unmodified**: These registers are not altered by IA-32 execution. Itanium-based code can rely on these values not being modified during IA-32 instruction set execution. The register will have the have the same contents when entering the IA-32 instruction set and when exiting the IA-32 instruction set.
- **Undefined**: Registers marked as undefined may be used as scratch areas for execution of IA-32 instructions. Software can not rely on the value of these registers across an instruction set transition.

Table 10-1. IA-32 System Register Mapping

Intel® Itanium™ Reg	IA-32 Reg	Convention	Size	Description
Application Registers				
EFLAG	EFLAG	IA-32 state	32	IA-32 System/Arithmetic flags, writes of some bits are conditioned by PSR.cpl and EFLAG.iopl.
CSD	CSD		64	IA-32 code segment (register format)
SSD	SSD			IA-32 stack segment (register format)
CFLG	CR0/CR4		64	IA-32 control flags, CR0=CFLG{31:0}, CR4=CFLG{63:32}[a], writable at PSR.cpl=0 only.
Kernel Registers				
KR0	IOBASE[b]	IA-32 state	64	IA-32 virtual I/O port Base register
KR1	TSSD[c]			IA-32 TSS descriptor (register format)
KR2	CR3/CR2[d]			IA-32 CR2=KR2{63:32}, CR3=KR2{31:0}
KR3-7		unmodified		Intel® Itanium™ preserved registers
Banked General Registers				
GR16-31		unmodified		Preserved for operating system use
Control Registers				
DCR		unmodified, shared		Controls instruction set execution (including IA-32)
IFA, IIP, IPSR, ISR, IIM, IIPA, ITTR, IHA, IFS, IVA		shared	64	Intel® Itanium™ interruption registers may be overwritten on any TLB fault, interruption or exception encountered during IA-32 or Intel® Itanium™ instruction set execution.
PTA		shared	64	Shared page table base for memory references (including IA-32)
ITM		shared		shared Intel® Itanium™ interruption/timer resources
LID, IVR, TPR, EOI, IRR0, IRR1, IRR2, IRR3, ITV, PMV, LRR0, LRR1, CMCV		shared	64	Intel® Itanium™ external interrupt control registers are used to generate, prioritize and delivery external interrupts during IA-32 or Intel® Itanium™ instruction set execution.

Table 10-1. IA-32 System Register Mapping (Continued)

Intel® Itanium™ Reg	IA-32 Reg	Convention	Size	Description
Translation Resources				
TRs		shared		All Intel® Itanium™ virtual memory registers can be used for memory references (including IA-32).
TCs				
RRs				
PKRs				
Debug Registers				
IBRs	dr0-3, dr7	shared	64	Intel® Itanium™ debug registers are used memory references (including IA-32).
DBRs	dr0-3, dr7			
Performance Monitors				
PMCs		shared	64	Intel® Itanium™ performance monitors measure performance events (including IA-32).
PMDs		shared	64	reflect performance monitor results of execution (including IA-32)

a. IA-32 MOV from CR0 and CR4 return the value in the CFLG register.

b. The IOBase register is used by IN/OUT instructions. If IN/OUT operations are disabled via CFLG.io, this register can be used for other values.

c. The TSSD registers are used by IN/OUT instructions for I/O permission via CFLG.io. If access to the TSS is disabled, these registers can be used for other values.

d. The Mov from CR2,CR3 instructions return the value contained in KR2.

10.3 IA-32 System Segment Registers

System Descriptors are maintained in an unscrambled format shown in Figure 10-1 that 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-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-based software can also directly load descriptor registers provided they are properly unscrambled by software. When Itanium-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 *Intel Architecture Software Developer's Manual*.

Figure 10-1. IA-32 System Segment Register Descriptor Format (LDT, GDT, TSS)

63	62 60	59	58 57	56	55 52	51 32	31 0
g	ig	p	dpl	s	stype	lim{19:0}	base{31:0}

Table 10-2. IA-32 System Segment Register Fields (LDT, GDT, TSS)

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. This value is ignored for Intel® Itanium™ instruction set memory references.
lim	51:32	Segment Limit. Contains the maximum effective address value within the segment. See the *Intel Architecture Software Developer's Manual* for details and segment limit fault conditions.
stype	55:52	Segment Type identifier. See the *Intel Architecture Software Developer's Manual* for encodings and definition.

Table 10-2. IA-32 System Segment Register Fields (LDT, GDT, TSS) (Continued)

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 an IA-32 memory reference uses this segment an IA_Exception(GPFault) is generated.
ig	62:60	Ignored - For the LDT/GDT/TSS descriptors reads of this field return the last value written by Itanium™-based code. Reads of this field return zero if written by IA-32 descriptor loads.These field is ignored by the processor during IA-32 instruction set execution. This field may have a future use and should be set to zero by software.
g	63	Segment Limit Granularity. If 1, scales the segment limit by lim=(lim<<12) | 0xFFF for IA-32 instruction set memory references.

System segment selectors and descriptors for GDT and LDT are maintained in Itanium general registers to support segment register loads used extensively by segmented 16-bit code. On the transition into the IA-32 instruction set, GDT/LDT descriptor table must be initialized if IA-32 code will perform protected mode segment register loads or far control transfers.

Within the IA-32 System Environment, GDT and LDT are considered privileged operating system segmentation resources. However, in the Itanium System Environment, applications can transition between the IA-32 and Itanium instruction set and bypass IA-32 segmentation. Itanium user level instructions can also directly modify all selectors and descriptors including GDT and LDT. An operating system should either protect memory with virtual memory management mechanisms defined by the Itanium architecture or disabled application level instruction set transitions. Within the Itanium System Environment, GDT/LDT memory spaces must be mapped into user space, since supervisor overrides for accesses to GDT/LDT are disabled.

The TSSD descriptor points to the I/O Permission Bitmap. If CFLG.io is 1, IN, INS, OUT, and OUTS consult the TSSD I/O permission bitmap as defined in the *Intel Architecture Software Developer's Manual*. If CFLG.io is 0, the TSSD I/O permission bitmap is not checked. See section Section 10.7 "I/O Port Space Model" for details on I/O port permission and for TLB based access control. The TSSD register is not used within the Itanium System Environment to support task switches, or interlevel control transfers. If the TSSD is used for I/O Permissions, Itanium-based operating system software must ensure that a valid 286 or 386 Task State Descriptor is loaded, otherwise IN/OUT operations to the TSSD I/O permission bitmap will result in undefined behavior.

The IDT descriptor is not supported or defined within the Itanium System Environment.

10.3.1 IA-32 Current Privilege Level

PSR.cpl is the current privilege level of the processor for instruction execution (including IA-32). PSR.cpl is used by the processor for all IA-32 descriptor segmentation and paging permission checks. PSR.cpl is a secured register. Typical IA-32 processors used SSD.dpl as the official privilege level of the processor. Since, SSD.dpl is not secured from user modification, processor implementations must base all privilege checks and state backups based on PSR.cpl.

10.3.2 IA-32 System EFLAG Register

The EFLAG (AR24) register is made 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. 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. System flags are typically managed by an operating system and are used to control the overall operations of the processor. System flags are broken into two categories, system flags that control IA-32 instruction set execution behavior and virtualizable system flags. The NT system flag shown in bold font in Figure 10-2 is virtualized.

Figure 10-2. IA-32 EFLAG Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved (set to 0)										id	vip	vif	ac	vm	rf	0	nt	iopl		of	df	if	tf	sf	zf	0	af	0	pf	1	cf

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved (set to 0)																															

System flags AC, TF, RF, VIF, VIP, IOPL and VM directly control the execution of IA-32 instructions. These bits do not control any Itanium instructions. See Table 10-3 for a complete definition these bits.

The NT bit does not directly control the execution of any IA-32 or Itanium instructions. All IA-32 instructions that modify this bit is intercepted (e.g. IRET, Task Switches)

When Itanium-based software loads this application register (AR24), a Reserved Register/Field fault will be raised if a non-zero value is written into bits listed as reserved.

10.3.2.1 Virtualized Interrupt Flag

To provide for virtualization of IA-32 code, the IF bit is virtualizable in the context of an operating system. Interrupts are enabled for IA-32 instructions, if `(PSR.i and (~CFLG.if or EFLAG.if))` is true. For Itanium-based code, interrupts are enabled if PSR.i is 1.

An optional System Flag intercept trap can be generated if CFLG.ii is 1, and the IF-flag changes state due to IA-32 code executing CLI, STI, or POPF. See "IA-32 Control Registers" for CFLG details. Using this model, virtualization code can set CFLG.if to 0 and CFLG.ii to 0, IA-32 instruction set modifications of EFLAG.if does not affect actual interrupt masking, therefore no notification events need be sent to virtualizing software. When virtualization code, detects and queues an external interrupt for delivery into a virtualized IA-32 operating system/application, it can set CFLG.ii to1 to force notification the next time the IF-bit changes state, indicating IA-32 code is either opening or closing the interrupt window. Setting CFLG.if to 1, allows for direct IA-32 control of interrupt masking.

Virtualization of the IF flag is independent of VME extensions. Both mechanisms can be used independently, see the *Intel Architecture Software Developer's Manual* for the complete VME definition.

Table 10-3. IA-32 EFLAG Field Definition

EFLAG[a]	Bits	Description
EFLAG.cf	0	IA-32 Carry Flag. See the *Intel Architecture Software Developer's Manual* for details.
	1	Ignored - Writes are ignored, reads return one for IA-32 and Intel® Itanium™ instructions.
	3,5, 15	Ignored - Writes are ignored, reads return zero for IA-32 and Intel® Itanium™ instructions. Software should set this bits to zero.
EFLAG.pf	2	IA-32 Parity Flag. See the *Intel Architecture Software Developer's Manual* for details.
EFLAG.af	4	IA-32 Aux Flag. See the *Intel Architecture Software Developer's Manual* for details.
EFLAG.zf	6	IA-32 Zero Flag. See the *Intel Architecture Software Developer's Manual* for details.
EFLAG.sf	7	IA-32 Sign Flag. See the *Intel Architecture Software Developer's Manual* for details.
EFLAG.tf	8	IA-32 Trap Flag- In the Intel® Itanium™ System Environment, IA-32 instruction single stepping is enabled when EFLAG.tf is 1 or PSR.ss is 1. EFLAG.tf does not control single stepping for Intel® Itanium™ instruction set execution. When single stepping is enabled, the processor generates a IA-32_Exception(Debug) trap event after the successful execution of each IA-32 instruction. If EFLAG.tf is modified by the POPF or POPFD instruction an IA-32_Intercept(SystemFlag) trap is raised. See the *Intel Architecture Software Developer's Manual* for details on this bit.
EFLAG.if	9	IA-32 Interruption Flag. In the Intel® Itanium™ System Environment, when PSR.i and (~CFLG.if or EFLAG.if) is 1, external interrupts are enabled during IA-32 instruction set execution, otherwise external interrupts are held pending. If CFLG.if is 1, modification of the EFLAG.if directly affects external interrupt enabling. If CFLG.if is 0, EFLAG.if does not affect interrupt enabling. The IF-bit does not affect external interrupt enabling for Intel® Itanium™ instructions nor NMI interrupts. The IF bit can be modified by IA-32 and Itanium™-based code only when PSR.cpl is less than or equal to EFLAG.iopl. If PSR.cpl is greater than EFLAG.iopl, writes to the IF-bit are silently ignored. If CFLG.ii is 1, successful modification of the IF-bit by CLI, STI, or POPF results in an IA-32_Intercept(SystemFlag) trap, otherwise the IF-bit is modified without interception. Modification of this bit by Intel® Itanium™ instructions does not result in an intercept. See the *Intel Architecture Software Developer's Manual* for details on this bit.
EFLAG.df	10	IA-32 Direction Flag. See *Intel Architecture Software Developer's Manual* for details.
EFLAG.of	11	IA-32 Overflow Flag. See *Intel Architecture Software Developer's Manual* for details.
EFLAG.iopl	13:12	IA-32 In/Out Privilege Level, controls accessibility by IA-32 IN/OUT instructions to the I/O port space and permission to modify the IF-bit for Intel® Itanium™ and IA-32 instructions. If PSR.cpl > IOPL, permission is denied for IA-32 IN/OUT instructions, and modifications of EFLAG.if by either IA-32 or Intel® Itanium™ instructions are ignored. IOPL can only be modified by IA-32 or Intel® Itanium™ instructions executing at privilege level 0, otherwise modifications of this bit are silently ignored. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments. See the *Intel Architecture Software Developer's Manual* for details on this bit.
EFLAG.nt	14	IA-32 Nested Task switch. In the IA-32 System Environment, indicates a nested task flag when chaining interrupted and called IA-32 tasks. IA-32 task switches are not directly supported in the Intel® Itanium™ System Environment, since IRET, interruptions, calls, and jumps through task gates are always intercepted. EFLAG.nt can be modified by the POPF or POPFD instruction in both system environments. Modification of EFLAG.nt by POPF and POPFD does not result in a System Flag Intercept. See the *Intel Architecture Software Developer's Manual* for details on this bit.

Table 10-3. IA-32 EFLAG Field Definition (Continued)

EFLAG[a]	Bits	Description
EFLAG.rf	16	IA-32 Resume Flag. In the Intel® Itanium™ System Environment, when EFLAG.rf or PSR.id is 1, code breakpoint faults are temporarily disabled for one IA-32 instruction, so that IA-32 instructions can be restarted after a code breakpoint fault without causing another code breakpoint fault. EFLAG.rf does not affect Intel® Itanium™ Instruction Debug faults. After the successful execution of each IA-32 instruction, PSR.id and EFLAG.rf are cleared to zero. On entry into the IA-32 instruction set via `rfi` or `br.ia,` EFLAG.rf and PSR.id is not cleared until the successful completion of the first (target) IA-32 instruction. `jmpe` clears the PSR.id and the EFLAG.rf bit. EFLAG.rf is set to 1 if a repeat string sequence (REP MOVS, SCANS, CMPS, LODS, STOS, INS, OUTS) takes an external interrupt, trap or fault before the final iteration. EFLAG.rf and PSR.id are set to 0 after the last iteration. For all other cases, external interrupts, faults, traps, and intercept conditions EFLAG.rf is unmodified. The RF-bit can be modified by Intel® Itanium™ instructions running at any privilege level. IA-32 instructions cannot directly modify the RF-bit or PSR.id. Specifically, POPF cannot modify the RF-bit and execution of IRET is always intercepted in the Intel® Itanium™ System Environment. See the *Intel Architecture Software Developer's Manual* for details on this bit.
EFLAG.vm	17	IA-32 Virtual Mode 86. When 1, IA-32 instructions execute in the VM86 environment. This bit can only be modified by IA-32 or Intel® Itanium™ instructions executing at privilege ring 0, otherwise modifications of this bit by Intel® Itanium™ or IA-32 instructions is silently ignored. Itanium™-based software is responsible for initializing the processor with the required VM86 register state before transferring to IA-32 VM86 environment. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments. See the *Intel Architecture Software Developer's Manual* for complete details of the VM86 environment. Software must ensure the processor is in IA-32 Protected Mode when setting the VM bit.
EFLAG.ac	18	IA-32 Alignment Check. In the Intel® Itanium™ System Environment, IA-32 instructions raise an IA-32_Exception(AlignmentCheck) fault if an unaligned reference is performed and PSR.ac is 1 or (CFLG.am is 1 and EFLAG.ac is 1 and memory is accessed at an effective privilege level of 3). Neither EFLAG.ac, CR0.am nor privilege level affect alignment check faults for Intel® Itanium™ instructions. See "Memory Alignment" for details on alignment conditions. This bit can be modified by IA-32 and Intel® Itanium™ instructions at any privilege level. Modification of this bit by the POPF instructions results in an IA-32_Intercept(SystemFlag) trap. See the *Intel Architecture Software Developer's Manual* for details on this bit.
EFLAG.vif	19	IA-32 Virtual Interrupt Flag. See VME extensions in the *Intel Architecture Software Developer's Manual* for details. Affects execution of POPF, PUSHF, CLI and STI. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments. A IA-32 Code Fetch fault (GPFault(0)) is generated on every IA-32 instruction (including the target of rfi and br.ia), if the following condition is true: EFLAG.vip & EFLAG.vif & CFLG.pe & PSR.cpl==3 & (CFLG.pvi | (EFLAG.vm & CFLG.vme))
EFLAG.vip	20	IA-32 Virtual Interrupt Pending. See VME extensions in the *Intel Architecture Software Developer's Manual* for programming details. Affects execution of POPF, PUSHF, CLI and STI. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments.
EFLAG.id	21	IA-32 Identifier bit, can be written and read by IA-32 instructions, indicates IA-32 CPUID instruction is supported. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments.
	63:22	reserved must be set to zero.

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

10.3.3 IA-32 System Registers

IA-32 system registers such as CR3, CR2, debug registers, performance counters. IA-32 control registers do not affect execution of Itanium instructions. All IA-32 privileged instructions that access prior IA-32 system registers are intercepted.

10.3.3.1 IA-32 Control Registers

IA-32 control registers CR0 and CR4 are mapped into the Itanium application register CFLG (AR27). IA-32 control bits, shown in Figure 10-3, only control execution of the IA-32 instruction set. Additional CR0 bits are defined in CFLG to control virtualization of IA-32 code (namely the IO and IF bits) as shown in Figure 10-3. CFLG is readable by Itanium-based code at all privilege levels but can only be written at privilege level 0, otherwise a Privileged Register fault is generated. When Itanium-based software loads this application register (AR24), a Reserved Register/Field fault will be raised if a non-zero value is written into bits listed as reserved.

Figure 10-3. Control Flag Register (CFLG, AR27)

31	30	29	28 27 26 25 24 23 22 21 20 19	18	17	16	15 14 13 12 11 10 9	8	7	6	5	4	3	2	1	0
PG	CD	NW	ignored (set to 0)	AM	ig	WP	ignored (set to 0)	II	IF	IO	NE	ET	TS	EM	MP	PE

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43	42	41	40	39	38	37	36	35	34	33	32
reserved (set to 0)	MMXEX	FXSR	PCE	PGE	MCE	PAE	PSE	DE	TSD	PVI	VME

- State in italics is virtualized. This state has no impact on a IA-32 or Itanium instruction set execution**.**
- State in bold only affects IA-32 instruction set execution, Itanium instruction execution is not affected.

Table 10-4 defines all IA-32 control register state and the behavior of each bit in the Itanium System Environment.

Table 10-4. IA-32 Control Register Field Definition

Bit	Intel® Itanium™ State	Bit	Description
CR0	CFLG{31:0}		CR0: IA-32 Mov to CR0 result in a instruction interception fault. Mov from CR0 returns the value contained in CFLG{31:0}. Modification of CFLG{31:0} by Intel® Itanium™ instructions only alters the CR0 state, no side effects (such as TLB flushes) occur.
CR0.PE	CFLG.pe	0	Protected Mode Enable: This bit determines whether the processor operates in IA-32 Protected Mode or Real Mode. This bit affects only IA-32 instruction set execution, Intel® Itanium™ operations are not affected by this bit. Modification of this bit by Itanium™-based code does have NOT any side effects such as flushing the TLBs. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments. See *Intel Architecture Software Developer's Manual* for details on this bit and the Protected Mode environment.
CR0.MP	CFLG.mp	1	Monitor co-Processor: When CFLG.ts is 1 and CFLG.mp is 1, execution of IA-32 FWAIT/WAIT instructions results in an Device Not Available fault. This bit is ignored by Intel® Itanium™ floating-point instructions. This bit is supported in both IA-32 and Intel® Itanium™ System Environments. See the *Intel Architecture Software Developer's Manual* for details on this bit.

Table 10-4. IA-32 Control Register Field Definition (Continued)

Bit	Intel® Itanium™ State	Bit	Description
CR0.EM	CFLG.em	2	Emulation: When CFLG.em is set, execution of IA-32 ESC and floating-point instructions generates an IA-32_exception(DNA) fault. When CFLG.em is 1, execution of IA-32 MMX technology or Streaming SIMD Extension instructions results in an IA-32_Intercept (Instruction) fault. This bit does not affect Intel® Itanium™ floating-point instructions. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments. See *Intel Architecture Software Developer's Manual* for details on this bit.
CR0.TS	CFLG.ts	3	Task Switched: When CFLG.ts is 1, execution of an IA-32 ESC, floating-point instruction, MMX technology or Streaming SIMD Extension instruction results in a IA-32_Exception(DNA) fault. When CFLG.ts is 1 and CFLG.mp is 1, execution of IA-32 FWAIT/WAIT instructions results in an IA-32_Exception(DNA) fault. This bit is ignored by Intel® Itanium™ instructions. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments. See *Intel Architecture Software Developer's Manual* for details on this bit.
CR0.ET	CFLG.et	4	Extension Type: ET is ignored since i387 co-processor instructions are supported. This bit is reserved on all Pentium processors. Reads always return 1. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments.
CR0.NE	CFLG.ne	5	Numeric Error: Numeric errors are always enabled in the Intel® Itanium™ System Environment. The NE bit and the IGNNE# pin are ignored by the processor and the FERR# pin is not asserted for any numeric errors on IA-32 or Intel® Itanium™ floating-point instructions. In the IA-32 System Environment, this bit is supported as defined in the *Intel Architecture Software Developer's Manual*.
--	CFLG.io	6	I/O Enable: If CFLG.io is 1 and CPL>IOPL, IA-32 IN, INS, OUT, OUTS instructions consulted the TSS for I/O permission. If CFLG.io is 0 or CPL<=IOPL, permission is granted regardless of the state of the TSS I/O permission bitmap (the bitmap is not referenced). This bit always returns zero when read by the IA-32 Mov from CR0 instruction. This bit is not defined in the IA-32 System Environment.
--	CFLG.if	7	IF Enable: When CFLG.if is 1, EFLAG.if can be used to enabled or disable external interrupts for IA-32 instructions. If CFLG.if is 0, EFLAG.if does not control external interrupt enabling. External interrupts are enabled for the IA-32 instruction set by if PSR.i and (~CLFG.if or EFLAG.if). This bit always returns zero when read by the IA-32 Mov from CR0 instruction. This bit is not defined in the IA-32 System Environment.
--	CFLG.ii	8	IF Intercept: When CFLG.ii is 1, successful modification of the EFLAG.if bit by IA-32 CLI, STI or POPF instructions result in a IA-32_Intercept(SystemFlag) trap. This bit always returns zero when read by the IA-32 Mov from CR0 instruction. This bit is not defined in the IA-32 System Environment.
ignored		9:15, 17, 19:28	Ignored - May have a possible future use. Software should set these fields to zero.
CR0.WP	CFLG.wp	16	Write Protect: This bit is ignored in the Itanium System Environment. In the IA-32 System Environment, WP controls supervisor write-protection for IA-32 paging. See *Intel Architecture Software Developer's Manual* for details on this bit.
CR0.AM	CFLG.am	18	Alignment Mask: For IA-32 instructions an IA-32_Exception(AlignmentCheck) fault is generated on a reference to an unaligned data memory operand if PSR.ac is 1 or (CFLG.am is 1 and EFLAG.ac is 1 and memory is accessed at an effective privilege level of 3). Neither EFLAG.ac, CR0.am nor privilege level affect alignment check faults for Itanium instructions. This bit is supported in both the IA-32 and Itanium System Environments. See the *Intel Architecture Software Developer's Manual* for details on this bit.
CR0.NW	CFLG.nw	29	Not Write-through and Cache Disable: These bits are ignored in the Itanium System Environment. Cacheability is controlled virtual memory attributes. These bits are provided as storage for compatibility purposes.
CR0.CD	CFLG.cd	30	

Table 10-4. IA-32 Control Register Field Definition (Continued)

Bit	Intel® Itanium™ State	Bit	Description
CR0.PG	CFLG.pg	31	Paging Enable: In the Itanium System Environment, this bit is ignored for IA-32 and Itanium memory references. Virtual translations are enabled via PSR.it and PSR.dt. This bit is provided as storage for compatibility purposes. Modification of this bit by Itanium-based code does NOT have any side effects such as flushing the TLBs. This bit is supported as defined in the *Intel Architecture Software Developer's Manual* for the IA-32 System Environment.
CR2	KR2{63:32}		IA-32 Page Fault Virtual Address: IA-32 Mov to CR2 result in an interception fault. Mov from CR2 returns the value contained in KR2{63:32}. CR2 is replaced by IFA in the Itanium System Environment.
CR3	KR2{31:0}		IA-32 Page Table Address: IA-32 Mov to CR3 result in an interception fault. Mov from CR3 return the value contained in KR2{31:0}. CR3 is replaced by PTA in the Itanium System Environment. Modification of KR2{31:0} by Itanium-based code does NOT have the side effect of flushing the TLBs.
CR3.PWT	KR4.pwt		Page Write-Through and Cache Disabled: In the Itanium System Environment, these bits are ignored. This bit are provided as storage for compatibility purposes. These bits are supported as defined in the *Intel Architecture Software Developer's Manual* for the IA-32 System Environment.
CR3.PCD	KR4.pcd		
CR4	CFLG{63:32}		CR4: A-32 Mov to CR4 result in an instruction interception fault. Mov from CR4 returns the value contained in CFLG{63:32}. Modification of CFLG{63:32} by Itanium instructions only alters the register state, no side effects (such as TLB flushes) occur.
CR4.VME	CFLG.vme	32	IA-32 Virtual Machine Extension and Protected Mode Virtual Interrupt Enable: These bits control the VM86 VME extensions and Protected Mode Virtual Interrupt extensions defined in the *Intel Architecture Software Developer's Manual* for STI, CLI and PUSHF. These bits have no effect on Intel® Itanium™ instructions. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments.
CR4.PVI	CFLG.pvi	33	
CR4.TSD	CFLG.tsd	34	Time Stamp Disable: IA-32 RDTSC user level reads of the Time Stamp Counter are enabled when CR4.tsd when zero. Otherwise execution of the RDTSC instruction results in a GPFault. CFLG.tsd is ignored by Intel® Itanium™ instructions. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments. See the *Intel Architecture Software Developer's Manual* for details on these bits.
CR4.DE	CFLG.de	25	Debug Extensions: In the Intel® Itanium™ System Environment, this bit is ignored by IA-32 or Intel® Itanium™ references to the I/O port space. This bit is provided as storage for compatibility purposes. This bit is supported as defined in the *Intel Architecture Software Developer's Manual* for the IA-32 System Environment.
CR4.PSE	CFLG.pse	36	Page Size Extensions: In the Intel® Itanium™ System Environment, this bit is ignored by IA-32 or Intel® Itanium™ references. In the IA-32 System Environment, this bit enables 4M-byte page extensions for IA-32 paging. Modification of this bit by Itanium™-based code does have any side effects such as flushing the TLBs.
CR4.PAE	CFLG.pae	37	Physical Address Extensions: In the IA-32 System Environment, this bit enables IA-32 Physical Address Extensions for IA-32 paging This bit is ignored in the Intel® Itanium™ System Environment. Modification of this bit by Itanium™-based code does have any side effects such as flushing the TLBs.
CR4.MCE	CFLG.mce	38	Machine Check Enable: This bit is ignored in the Intel® Itanium™ System Environment. This bit is provided as storage for compatibility purposes. This bit is supported as defined in the *Intel Architecture Software Developer's Manual* for the IA-32 System Environment.
CR4.PGE	CFLG.pge	39	Paging Global Enable: This bit is ignored in the Intel® Itanium™ System Environment. This bit is provided as storage for compatibility purposes. This bit is supported as defined in the *Intel Architecture Software Developer's Manual* for the IA-32 System Environment, where this bit enables global pages for the IA-32 paging. Modification of this bit by Itanium™-based code does have any side effects such as flushing the TLBs.

Table 10-4. IA-32 Control Register Field Definition (Continued)

Bit	Intel® Itanium™ State	Bit	Description
CR4.PCE	CFLG.pce	40	Performance Counter Enable: IA-32 RDPMC user level reads of the performance counters are enabled when CR4.pce is 1. Otherwise execution of the RDPMC instruction results in a GPFault. CFLG.pce is ignored by Intel® Itanium™ instructions. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments. See the *Intel Architecture Software Developer's Manual* for details on these bits.
CR4. FXSR	CFLG. FXSR	41	Streaming SIMD Extension FXSR Enable. When 1, enables the Streaming SIMD Extension register context. When 0, execution of all Streaming SIMD Extension instructions results in an IA-32_Intercept(Instruction) fault. This bit does not control the behavior of Intel® Itanium™ instructions. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments. See the *Intel Architecture Software Developer's Manual* for details on these bits.
CR4. MMXEX	CFLG. MMXEX	42	Streaming SIMD Extension Exception Enable: When 1, enables Streaming SIMD Extension unmasked exceptions. When 0, all Streaming SIMD Extension Exceptions are masked. This bit does not control the behavior of Intel® Itanium™ instructions. This bit is supported in both the IA-32 and Intel® Itanium™ System Environments. See the *Intel Architecture Software Developer's Manual* for details on these bits.
reserved		43:63	Reserved

10.3.3.2 IA-32 Debug Registers

Within the Itanium System Environment, the IA-32 debug registers (DR0 - DR7) are superseded by the Itanium debug registers DBR0-7 and IBR0-7, see "Data Breakpoint Register Matching" for details. Accesses to the IA-32 debug registers result in an interception fault.

The Itanium debug registers are designed to facilitate debugging of both IA-32 and Itanium-based code. Specifically, instruction and data breakpoints can be programmed by loading 64-bit virtual addresses into IBR and DBR along with an address mask. Itanium defined single stepping mechanisms, and taken branch traps are also defined to trap on IA-32 instructions. See "Data Breakpoint Register Matching" for details on IA-32 instruction set behavior with respect to the debug facilities defined by the Itanium architecture.

10.3.3.3 IA-32 Memory Type Range Registers (MTRRs)

Within the Itanium System Environment, IA-32 MTRR registers are superseded by physical memory attributes supplied by the TLB, as defined in "Cacheability and Coherency Attribute". IA-32 instruction references to the MTRRs in the MSR register space results in an instruction intercept fault.

10.3.3.4 IA-32 Model Specific and Test Registers

Within the Itanium System Environment, the IA-32 Model Specific Register space (MSRs) are superseded by the PAL firmware interface. Cache testing, initialization, processor configuration should be performed through the PAL interface. See "PAL Procedures" for a complete definition of the PAL functions and interfaces. Accesses to the IA-32 Model Specific Register space result in an instruction interception fault.

10.3.3.5 IA-32 Performance Monitor Registers

Within the Itanium System Environment, the Itanium performance monitors are designed to measure IA-32 and Itanium instructions, and system performance through a unified performance monitoring facility. Itanium-based code can program the performance monitors for IA-32 and/or Itanium events by configuring the PMC registers. Count values are accumulated in the PMD registers for both IA-32 and Itanium events. See implementation specific documentation for the list of supported events and encodings.

IA-32 code can sample the performance counters by issuing the RDPMC instruction. RDPMC returns count values from the specified Itanium performance monitor. Operating systems can secure the monitors from being read by IA-32 code by setting PSR.sp to 1, or setting CR4.pce to 0, or setting the performance monitor’s pm-bit. Reads of a secured counter by RDPMC return a IA-32_Exception(GPFault(0)). IA-32 code cannot write or configure the performance monitors, all writes to the MSR register space are intercepted.

10.3.3.6 IA-32 Machine Check Registers

Within the Itanium System Environment, IA-32 machine check registers are superseded by the Itanium machine check architecture. See “Machine Checks” for details. IA-32 accesses to the Pentium III Processor machine check registers results in an instruction intercept.

10.4 Register Context Switch Guidelines for IA-32 Code

The following section gives operating system performance guidelines to minimize the amount of register context that must be saved and restored for IA-32 processes during a context switch.

10.4.1 Entering IA-32 Processes

High FP registers (FR32-127) - The processor requires access to all high FP registers during the execution of IA-32 instructions. It is recommended on entering an IA-32 process, that the OS save the high FP registers belonging to a prior context and then **enable** the high FP registers (PSR.dfh is 0). Otherwise, the processor will immediately raise a Disabled FP Register fault on the first IA-32 instruction executed in the IA-32 process. Performing the state save of the prior high FP register context during the context switch avoids the unnecessary generation of the Disabled FP Register fault.

Low FP registers (FR2-31) - The processor does not require access to the low FP registers unless executing IA-32 FP, MMX technology or Streaming SIMD Extension instructions. It is recommended on entry to an IA-32 process, that the OS **disable** the low FP registers by setting PSR.dfl to 1. PSR.dfl set to 0 indicates that there was a possibility that IA-32 FP, MMX technology or Streaming SIMD Extension instruction could execute and write FR8-31. If the low FP registers are enabled on entry to an IA-32 process (PSR.dfl is 0), all low FP registers will appear to be dirty on IA-32 process exit.

High Integer Registers (GR32-127) - Since the processor leaves all high registers in the register stack in an undefined state, these registers must be saved by the RSE before entering an IA-32 process.

Low Integer registers (GR1-31) - These registers must be explicitly saved before entering an IA-32 process.

10.4.2 Exiting IA-32 Processes

High FP registers (FR32-127) - PSR.mfh is unmodified when leaving the IA-32 instruction set. IA-32 instruction set execution leaves FR32-127 in an undefined state. Software can not rely on register values being preserved across an instruction set transition. These registers do NOT need to be preserved across a context switch.

Low FP registers (FR2-31) - PSR.mfl indicates there is a possibility that FR8-31 were modified by IA-32 FP, MMX technology, or Streaming SIMD Extension instruction. The modify bit is set by the processor when leaving the IA-32 instruction set, if PSR.dfl is 0, otherwise PSR.mfl is unmodified. During the state save of the outbound IA-32 process, it is recommended that the OS save FR2-31 if and only if the lower FP registers are marked as modified.

High Integer Registers (GR32-127) - Since the processor leaves all high registers undefined across an instruction set transition, these registers do NOT need to be preserved across an IA-32 context switch.

Low Integer registers (GR1-31) - These registers must be explicitly preserved across a context switch.

10.5 IA-32 Instruction Set Behavior Summary

Table 10-5 summarizes IA-32 instruction behavior within the Itanium System Environment. All IA-32 instructions are unchanged from the *Intel Architecture Software Developer's Manual* except where noted. IA-32 instructions can also generate additional Itanium register and memory faults as defined in Table 5-5. Please refer to the *Intel Architecture Software Developer's Manual* for the behavior of all IA-32 instructions in the IA-32 System Environment.

For all listed and unlisted IA-32 instructions in Table 10-5 the following relationships hold:

- Writes of any IA-32 general purpose, floating-point or MMX technology or Streaming SIMD Extension registers by IA-32 instructions are reflected in the Itanium registers defined to hold that IA-32 state when the IA-32 instruction set completes execution.
- Reads of any IA-32 general purpose, floating-point or MMX technology or Streaming SIMD Extension registers by IA-32 instructions see the state of the Itanium registers defined to hold the IA-32 state after entering the IA-32 instruction set.
- IA-32 numeric instructions are controlled by and reflect their status in FCW, FSW, FTW, FCS, FIP, FOP, FDS and FEA. On exit from the IA-32 instruction set, Itanium registers defined to hold IA-32 state reflect the results of all IA-32 prior numeric instructions (FSR, FCR, FIR, FDR). Itanium numeric status and control resources defined to hold IA-32 state are honored by IA-32 numeric instructions when entering the IA-32 instruction set.

In Table 10-5 *unchanged* indicates there is no change in behavior with respect to the IA-32 System Environment.

Table 10-5. IA-32 Instruction Summary

IA-32 Instruction	Intel® Itanium™ System Environment	Comments
AAA, AAD. AAM, AAS	unchanged	
ADC, ADD, AND,		
ADDPS, ADDSS, ANDNPS, ANDPS		
ARPL		
BOUND		
BSF, BSR		
BSWAP		
BT, BTC, BTS, BTR		
CALL	near: no change far: no change gate more privilege: Gate Intercept gate same privilege: Gate Intercept task gate: Gate Intercept + additional taken branch trap	Intercept if through a call or task gate If PSR.tb is 1, raise a taken branch trap.
CBW, CWDE, CDQ	unchanged	
CLC, CLD		
CLI	Optional System Flag Intercept	Intercept if EFLAG.if changes state and CFLG.ii is 1
CLTS	Instruction Intercept	IA-32 privileged instruction
CMC	unchanged	
CMOV		
CMP		
CMPPS, CMPSS, COMISS		
CMPS		
CMPXCHG, 8B	Optional Lock Intercept	If Locks are disabled (DCR.lc is 1) and a processor external lock transaction is required
CPUID	unchanged	
CWD, CDQ		
CVTPI2PS, CVTPS2PI, CVTSI2SS, CVTSS2SI, CVTTPS2PI, CVTTSS2SI		
DAA, DAS		
DEC		
DIV		
DIVPS, DIVSS		
ENTER		
EMMS		

Table 10-5. IA-32 Instruction Summary (Continued)

IA-32 Instruction	Intel® Itanium™ System Environment	Comments
F2XM1	unchanged	IA-32 numeric instructions manipulate the IA-32 numeric register stack contained in f8-f15, status is reflected in FSR. Modification of the IA-32 numeric environment changes FIR, FDR FCR and FSR.
FABS		
FADD, FADDP, FIADD		
FBLD		
FBSTP		
FCHS		
FCLEX, FNCLEX		
FCMOV		
FCOM, FCOMPP		
FCOMI, FCOMIP		
FUCOMI, FUCOMIP		
FCOS		
FDECSTP		
FDIV, FDIVP, FIDIV		
FDIVR, FDIVRP, FDIVR		
FFREE		
FICOM, FICOMP		
FILD		
FINCSTP		
FINIT, FNINIT		
FIST, FISTP		
FLD		
FLD constant		
FLDCW		
FLDENV		
FMUL, FMULP, FIMUL		
FNOP		
FPATAN, FPTAN		
FPREM, FPREM1		
FRNDINT		
FRSTOR		
FSAVE, FNSAVE		
FSCALE		
FSIN, FSINCOS		
FSQRT		
FST, FSTP		
FSTCW, FNSTCW		
FSTENV, FNSTENV		
FSTSW, FNSTSW		
FSUB, FSUBP, FISUB		
FSUBR, FSUBRP, FISUBR		
FTST		
FUCOM, FUCOMP		
FWAIT		
FXAM		
FXCH		
FXTRACT		
FXRSTOR, FXSAVE		
FYL2X, FYL2XP1		

Table 10-5. IA-32 Instruction Summary (Continued)

IA-32 Instruction	Intel® Itanium™ System Environment	Comments
HLT	Instruction Intercept	IA-32 privileged instruction
IDIV	unchanged	
IMUL		
IN, INS	unchanged + I/O ports are mapped virtually	If CFLG.io is 0, the TSS I/O permission bitmap is not consulted. Intel® Itanium™ TLB faults control accessibility to I/O ports.
INC	unchanged	
INT 3, INTO	Mandatory Exception vector #	Delivered as an IA-32_Interrupt
INT n	Mandatory Interruption vector #	Delivered as an IA-32_Exception
INVD	Instruction Intercept	IA-32 privilege instruction
INVLPG		
IRET, IRETD	Real Mode: Instruction Intercept to VM86: Instruction Intercept from VM86: Instruction Intercept same privilege: Instruction Intercept less privilege: Instruction Intercept different task: Instruction Intercept	All forms of IRET result in an instruction intercept
Jcc	additional taken branch trap	If PSR.tb is 1, raise a taken branch trap.
JMP	near: no change far: no change gate task: Gate Intercept call gate: Gate Intercept additional taken branch trap	Intercept fault if through a call or task gate If PSR.tb is 1, raise a taken branch trap.
JMPE		Jumps to the Intel® Itanium™ instruction set
LAHF	unchanged	
LAR		
LDMXCSR		
LDS, LES, LFS, LGS, LSS		
LEA		
LEAVE		
LGDT, LLDT	Instruction Intercept	IA-32 privileged register resource
LIDT		
LMSW		
Lock prefix	Optional Lock Intercept	If Locks are disabled (DCR.lc is 1) and a processor external lock transaction is required
LODS	unchanged	
LOOP, LOOPcc	additional taken branch trap	If PSR.tb is 1, raise a taken branch trap.
LSL	unchanged	User level instruction
LTR	Instruction Intercept	IA-32 privileged register

Table 10-5. IA-32 Instruction Summary (Continued)

IA-32 Instruction	Intel® Itanium™ System Environment	Comments
MASKMOVQ	unchanged	
MAXPS, MAXSS, MINPS, MINSS		
MOV		
MOVNTPS, MOVNTQ		
MOV from CR	unchanged	
MOV to CR	Instruction Intercept	IA-32 privileged system registers
MOV to/from DR		
Mov SS	System Flag Intercept Trap	System Flag Intercept Trap after instruction completes
MOVAPS, MOVHPS, MOVLPS. MOVMSKPS, MOVSS, MOVUPS	unchanged	
MOVD, MOVQ		
MOVS		
MOVSX, MOVZX		
MUL		
MULPS, MULSS		
NEG		
NOP		
NOT		
OR		
ORPS		
OUT, OUTS	unchanged + I/O ports are mapped virtually	If CFLG.io is 0, the TSS I/O permission bitmap is not consulted. Intel® Itanium™ TLB faults control accessibility to I/O ports.
PACKSS, PACKUS	unchanged	
PADD, PADDS, PADDUS		
PAND, PANDN		
PCMPEQ, PCMPGT		
PEXTRW, PINSRW		
PMADD		
PMULHW, PMULLW, PMULHUW		
PMOVMSKB		
POP, POPA		
POP SS	System Flag Intercept	System Flag Intercept Trap after instruction completes
POPF, POPFD	Optional System Flag Intercept	Intercept if EFLAG.if changes state and CFLG.ii is 1 Intercept if EFLAG.ac, or tf change state.
POR	unchanged	
PREFETCH		
PSHUFW		
PSLL, PSRA, PSRL		
PSUB, PSUBS, PSUBUS		
PUNPCKH, PUNPCKL		
PXOR		
PUSH, PUSA	unchanged	
PUSHF, PUSHFD		Pushes value in EFLAG, no intercept
RCL, RCR, ROL, ROR		
RCPPS, RSQRTPS		

Table 10-5. IA-32 Instruction Summary (Continued)

IA-32 Instruction	Intel® Itanium™ System Environment	Comments
RDMSR	Instruction Intercept	IA-32 privileged system register space
RDTSC	Optional GPFault	No longer faults in VM86, GPFault if secured by PSR.si or CFLG.tsd.
RDPMC		
REP, REPcc prefix	unchanged	
RET	near: no change far: no change less privilege: no change same privilege: no change + additional taken branch trap	If PSR.tb is 1, raise a taken branch trap.
RSM	Instruction Intercept	IA-32 privileged instruction
SAHF	unchanged	
SAL, SAR, SHL, SHR		
SBB		
SCAS		
SFENCE		
SETcc		
SGDT, SLDT	Instruction Intercept	IA-32 privileged instruction
SHLD, SHRD	unchanged	
SHUFPS, SQRTPS, SQRTSS		
SIDT	Instruction Intercept	IA-32 privileged instructions
SMSW		
STC, STD	unchanged	
STI	Optional System Flag Intercept	Intercept if EFLAG.if changes state and CFLG.ii is 1
STMXCSR	unchanged	
STOS		
STR	Instruction Intercept	IA-32 privileged instruction
SUB	unchanged	
SUBPS, SUBSS		
TEST	unchanged	
UCOMISS		
UNPCKHPS, UNPCKLPS		
UD2	Instruction Intercept	Reserved undefined opcodes
VERR, VERW	unchanged	User level instruction
WAIT		
WBINVD	Instruction Intercept	IA-32 privileged instructions
WRMSR		
XADD	Optional Lock Intercept	If Locks are disabled (DCR.lc is 1) and a processor external lock transaction is required than a Lock Intercept.
XCHG		
XLAT, XLATB	unchanged	
XOR		
XORPS		

10.6 System Memory Model

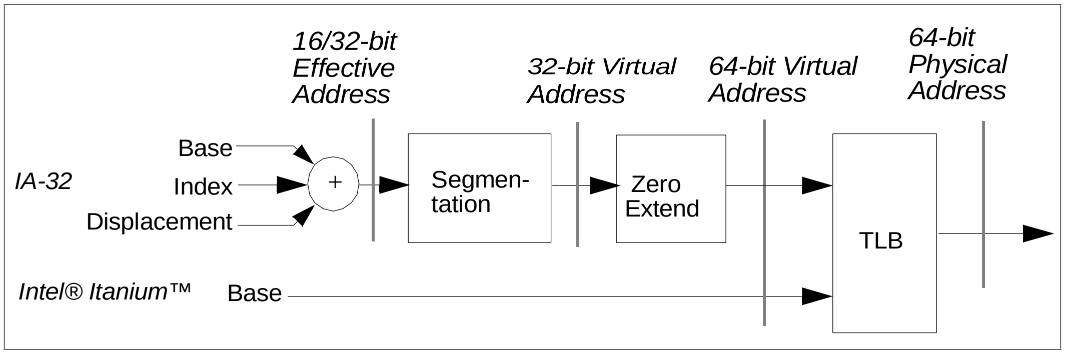
Within the Itanium System Environment, a unified memory model is presented to the programmer. Applications and the operating system see the same 64-bit virtual memory space and virtual addressing mechanisms regardless of the referencing instruction set. A virtual address points to the same physical storage location from both IA-32 and Itanium instruction sets.

Itanium-based operating systems must not use IA-32 segmentation as a protected system resource. An Itanium-based operating system must use virtual memory management defined by the Itanium architecture to secure IA-32 and Itanium-based applications, memory and I/O devices. The Itanium architecture is defined to be *unsegmented architecture and all Itanium memory references bypass IA-32 segmentation and protection checks*. In addition, Itanium-based user level code can directly modify IA-32 segment selector and descriptor values for all segments (including GDT and LDT). If operating systems do not rely on segmentation for protection, there are no security concerns for exposing IA-32 segment registers and descriptors to Itanium-based user level applications

IA-32 instruction and data reference addresses are generated as 16/32-bit effective addresses as shown in Figure 10-7. IA-32 segmentation is then applied to map 32-bit effective addresses into 32-bit virtual addresses, the processor then converts the address into a 64-bit virtual address by zero extension from 32 to 64-bits. Itanium instructions bypass all of these steps and directly generate addresses within the 64-bit virtual address space.

For both IA-32 and Itanium instruction set memory references, virtual memory management defined by the Itanium architecture is used to map a given virtual address into a physical address. Itanium-based virtual memory management hardware does not distinguish between Itanium and IA-32 instruction set generated memory references during the conversion from a virtual to physical address.

Figure 10-4. Virtual Memory Addressing



10.6.1 Virtual Memory References

In the Itanium System Environment the following virtual memory options are available for supporting IA-32 and Itanium memory references.

- Software TLB fills (TLBs are enabled, but the VHPT is disabled).
- 8-byte short format VHPT, see “Virtual Hash Page Table (VHPT)” for details.
- 32-byte long format VHPT.

Itanium virtual memory resources can be used by the operating system for all IA-32 memory references. These resources include virtual Region Registers (RR), Protection Key Registers (PKR), the Virtual Hash Page Table (VHPT), all supported range of page sizes, Translation Registers (ITR, DTR), the Translation Cache (ITC, DTC) and the complete set of Itanium virtual memory management faults defined in Chapter 5.

10.6.2 IA-32 Virtual Memory References

By definition, IA-32 instruction and data memory references are confined to 32-bits of virtual addressing, the first 4 G-bytes of virtual region 0. However, IA-32 memory references can be mapped anywhere within the implemented physical address space by operating system code.

Virtual addresses are converted into physical addresses through the process defined in “Virtual Addressing”. IA-32 references use the Itanium TLB resources as follows.

- **Region Identifiers**- Operating systems can place IA-32 processes within virtual region 0, and use the entire 2^{24} region identifier name space. By using region identifiers there is no requirement to flush IA-32 mappings on a context switch.
- **Protection Keys** - Operating systems can place mappings used by IA-32 processes within any number of protection domains. If PSR.pk is 1, all IA-32 references search the Protection Key Registers (PKR) for matching keys. If a key is not found, a Key Miss fault is generated. Otherwise, key read, write, execute permissions are verified.
- **TLB Access Bit** - If this bit is zero, an Access Bit fault is generated during Itanium or IA-32 instruction set memory references. Note: the processor does not automatically set the Access bit in the VHPT on every reference to the page. Access bit updates are controlled by the operating system.
- **TLB Dirty Bit** - If this bit is zero, a Dirty bit fault is generated during any Itanium or IA-32 instruction that stores to a dirty page. Note: the processor does not automatically set the Dirty bit in the VHPT on every write. Dirty bit updates are managed by the operating system.

10.6.3 IA-32 TLB Forward Progress Requirements

To ensure forward progress while executing IA-32 instructions, additional TLB resources and replacement policies must be defined over and above the definition given in “Translation Cache (TC)”. IA-32 instructions and data accesses may not be aligned resulting in a worst case scenario for two possible pages being referenced for every memory datum referenced during the execution of an IA-32 instruction. Furthermore, the worst case non-intercepted IA-32 opcode can reference up to 4 independent data pages.

The Translation Cache's (TC) are required to have the following minimum set of resources to ensure forward progress. Given that software TLB fills can be used to insert entries into the TLB and a hardware page table walker is not necessarily used, the following requirements must be satisfied by the processor:

- Instruction Translation Cache - at least 1 way set associative with 2 sets, or 2 entries in a fully associative design. Replacement algorithms must not consistently displace the last 2 entries installed by software.
- Data Translation Cache - at least 4 way set associative with 2 sets, or 8 entries in a fully associative design. Replacement algorithms must not consistently displace the last 8 entries installed by software or the last 8 translations referenced by an IA-32 instruction.
- Unified Translation Cache - at least 5 way set associative with 2 sets, or 10 entries in a fully associative design. The processor must not consistently displace the last 10 entries installed or the last 10 translations referenced by an IA-32 instruction.

The processor must ensure that the minimum number of entries can co-exist in the TLB, and TC replacement algorithms allow software insertion of the required entries such that the required number of translations can be co-resident in the TLB.

The processor cannot ensure forward progress unless translations mapping the Itanium-based TLB Miss handlers are statically mapped by the Instruction Translation Registers.

10.6.4 Multiprocessor TLB Coherency

Global TLB purges can not occur on another processor unless that processor is at an interruptible point. For IA-32 instruction set execution, interruptible points are defined as; 1) when the processor is between instructions (regardless of the state of PSR.i and EFLAG.if), and 2) each iteration of an IA-32 string instruction, regardless of the state of PSR.i and EFLAG.if

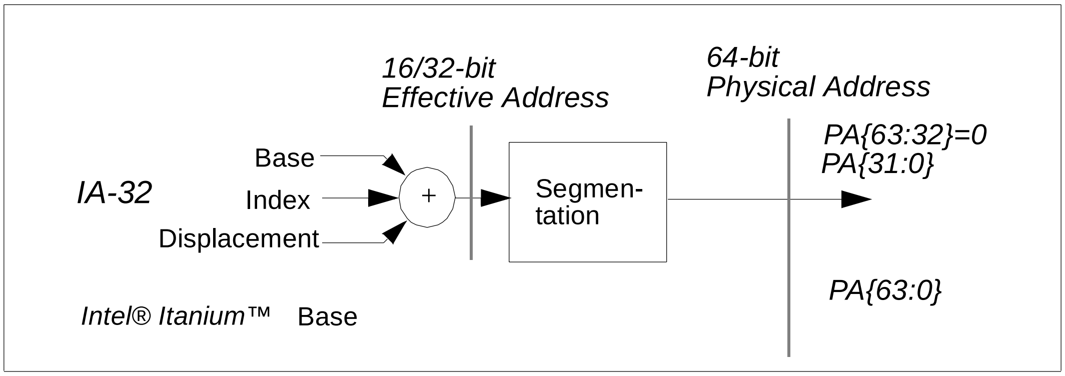
The processor may delay in its response and acknowledgment to a broadcast purge TC transaction until the processor executing an IA-32 instruction has reached a point (e.g. an IA-32 instruction boundary) where it is safe to process the purge TC request. The amount of the delay is implementation specific and can vary depending on the receiving processor and what instructions or operations are executing when it receives the purge request.

10.6.5 IA-32 Physical Memory References

When running IA-32 code, virtual addressing must be utilized by setting PSR.dt to 1 and PSR.it to 1, otherwise processor operation is undefined. Undefined behavior can include, but is not limited to: machine check abort on entry to IA-32 code, and unpredictable behavior for IA-32 self modifying code.

Operating systems must ensure PSR.dt and PSR.it are 1 before invoking IA-32 code. From a practical standpoint, the TLBs must be enabled so IA-32 code can access the virtual address space, and access memory areas other than WB (e.g. UC or the I/O port space).

Figure 10-5. Physical Memory Addressing



10.6.6 Supervisor Accesses

If the processor is operating in the Itanium System Environment, supervisor override is disabled, and LDT, GDT, TSS references are performed at the privilege level specified by PSR.cpl. Unaligned processor references to LDT, GDT, and TSS segments will never generate an EFLAG.ac enabled IA-32 Exception (AlignmentCheck) fault, even if PSR.cpl equals 3 and supervisor override is disabled.

Operating systems must ensure that the GDT/LDT are mapped to pages with user level read/write access.

Write permission is required if GDT, or LDT memory descriptor Access-bits are zero regardless of supervisor override conditions. If all GDT/LDT descriptor Access-bits are one, write permission can be removed. Otherwise, Access Rights, Key Miss or Key Miss faults can be generated during all segment descriptor referencing instructions.

If a fault is generated during a supervisory access, the ISR.so bit indicates that CPL is zero or a supervisor override condition was in effect (reference as made to GDT, LDT or TSS).

10.6.7 Memory Alignment

Depending on software conventions, memory structures may have different alignment or padding restrictions for the IA-32 and Itanium instruction sets. IA-32 and Itanium-based software should use aligned memory operands as much as possible to avoid possible severe performance degradation associated with un-aligned values and extra over-head for unaligned data memory fault handlers.

The processor provides full functional support for all cases of un-aligned IA-32 data memory references. If PSR.ac is 1 or EFLAG.ac is 1 and CR0.am is 1and the effective privilege level is 3, unaligned IA-32 memory references result in an IA-32 Exception (AlignmentCheck) fault. Unaligned processor references to LDT, GDT, and TSS segments will never generate an EFLAG.ac enabled IA-32 Exception (AlignmentCheck) fault, even if the effective privilege level is 3 and supervisor override is disabled.

Alignment conditions for Itanium memory references are not affected by the EFLAG.ac, CFLG.am bits.

If EFLAG.ac and CFLG.am are 1 and the reference is done at privilege level 3, IA-32 instruction set unaligned conditions are; 2-byte references not a 2-byte boundary, 4-byte references not on a 4-byte boundary, 8-byte references not on a 8-byte boundary, and 10-byte references not on a 8-byte boundary.

If PSR.ac is 1, IA-32 instruction set unaligned conditions are; 2-byte references not a 2-byte boundary, 4-byte references not on a 4-byte boundary, 8-byte references not on a 8-byte boundary, and 10-byte references not on a **16**-byte boundary.

The processor exhibits the following behavior when accesses are made to un-aligned data operands that span virtual page boundaries:

- IA-32 instruction set - If either page contains a fault, no memory location is modified. For reads, the destination register is not modified.
- Itanium instruction set - All page crossers result in an unaligned reference fault. Memory contents and register contents are not modified.

10.6.8 Atomic Operations

All Itanium load/stores and IA-32 non-locked memory references up to 64-bits that are aligned to their natural data boundaries are atomic.

Both IA-32 and Itanium atomic semaphore operations can be performed on the same shared memory location. The processor ensures IA-32 locked read-modify-write opcodes and Itanium semaphore operations are performed atomically even if the operations are initiated from the other instruction set by the same processors, or between multiple processors in an multiprocessing system.

There are some restrictions placed on Itanium atomic operations that may prevent Itanium-based code from manipulating IA-32 semaphores in some rare cases:

- Unaligned Itanium semaphore operations result in an Unaligned Data Reference fault. Itanium-based code manipulation of an IA-32 semaphore can only be performed if the IA-32 semaphore is aligned.
- Itanium semaphore operations to memory which is neither write-back cacheable nor a NaTPage result in an Unsupported Data Reference fault (regardless of the state of the DCR.lc). Itanium-based code manipulation of an IA-32 semaphore can only be performed if the IA-32 semaphore is allocated in aligned write-back cacheable memory.

If an IA-32 locked atomic operation is defined as requiring a read-modify-write operation external to the processor under external bus lock and if DCR.lc is set to 1, an IA-32_Intercept(Lock) fault is generated. (IA-32 atomic memory references are defined to require an external bus lock for atomicity when the memory transaction is made to non-write-back memory or are unaligned across an implementation-specific non-supported alignment boundary.) If DCR.lc is set to 0, the processor may either execute the transaction as a series of non-atomic transactions or perform the transaction with an external bus lock, depending on the processor implementation. For processor implementations that do support external bus locks, software must ensure that the Bus Lock Mask bit is set to one, in order to ensure atomicity of these IA-32 operations when DCR.lc=0. The Bus Lock Mask bit is a feature controllable by the PAL_BUS_SET_FEATURES procedure. (See Table 11-24 on page 2:296 for more information).

If the processor supports external bus locks, unaligned IA-32 atomic references are supported, but their usage is strongly discouraged since they are typically performed outside the processor's cache which can severely degrade performance of the system. IA-32 external bus locks are not supported on all processor implementations.

For IA-32 semaphores, atomicity to uncached memory areas (UC) is platform specific, atomicity can only be ensured by the platform design and can not be enforced by the processor.

10.6.9 Multiprocessor Instruction Cache Coherency

The processor and platform ensure the processor's instruction cache is coherent for the following conditions:

- For all processors in the coherency domain, local and remote instruction cache coherency on all processors is enforced for any store generated by any processor running the IA-32 instruction set.
- For all processors in the coherency domain, instruction cache coherency on all processors is enforced for all coherent I/O traffic. (For non-coherent I/O, a processor may or may not see the results of an I/O operation.)
- For all processors in the coherency domain, instruction cache coherency is not enforced for stores generated by any processor running the Itanium instruction set. To ensure instruction cache coherency, Itanium-based code must use the code sequence defined in "Memory Consistency" on page 2:65.

Table 10-6. Instruction Cache Coherency Rules

Originating Instruction Set	Local processor	External Processor	Coherent, I/O	Non-Coherent I/O
IA-32	Coherent	Coherent	Coherent	Maybe Non-Coherent
Intel® Itanium™	May be Non-coherent	May be Non-coherent		

10.6.10 IA-32 Memory Ordering

IA-32 memory ordering follows the Pentium III defined ***processor ordered*** model for cacheable and uncacheable memory. IA-32 *processor ordered* memory references are mapped onto the Itanium memory ordering model as follows:

- All IA-32 stores have ***release*** semantics. Except for IA-32 stores to write-coalescing memory that are unordered. Subsequent loads are allowed to bypass buffered local store data before it is globally visible. The amount of store buffering is model specific and can vary across processor generations.
- All IA-32 loads have ***acquire*** semantics. Some high performance processor implementations may speculatively issue *acquire* loads into the memory system for speculative memory types, if and only if the loads do not *appear* to pass other loads as observed by the program. If there is a coherency action that would result in the appearance to the program of a load bypassing other load, the processor will reissue the load operation(s) in program order.

- All IA-32 read-modify-write or locked instructions have memory ***fence*** semantics. All buffered stores are flushed.
- IA-32 IN, OUT and serializing operations (as defined in the *Intel Architecture Software Developer's Manual*) have memory ***fence*** semantics. In addition, the processor will wait for completion (acceptance by the platform) of the IN or OUT before executing the next instruction. All buffered stores are flushed before the IN or OUT operation.
- IA-32 SFENCE has ***release*** semantics and will flush all buffered stores.

Table 10-7. IA-32 Load/Store Sequentiality and Ordering

IA-32 Memory Reference	Uncacheable	Write Coalescing	Cacheable
store	sequential release[a]	non-sequential unordered	non-sequential release[b]
load	sequential acquire[a]	non-sequential unordered	non-sequential acquire[b]
locked or read-modify-write operation	sequential fence flush prior stores	non-sequential fence flush prior stores	non-sequential fence flush prior stores
IN, INS, OUT, OUTS	sequential fence flush prior stores	undefined	undefined
IA-32 Serialization	fence, flush prior stores		
SFENCE	release, flush prior stores		

a. However, IA-32 loads/stores to uncacheable memory flush the write coalescing buffers.
b. However, IA-32 load/stores to cacheable memory do not flush the write coalescing buffers.

Per Table 10-7, IA-32 memory references can be expressed in terms of acquire, release, fence and sequential ordering rules defined by the Itanium architecture. IA-32 data memory references follow the same ordering relationships as defined for Itanium-based code as defined in "Sequentiality Attribute and Ordering" on page 2:69. The following additional clarifications need to be made for IA-32 instruction set execution:

- IA-32 loads and instruction fetches to speculative memory can occur randomly. Read accesses to speculative memory can occur at arbitrary times even if the in-order execution of the program does not require a read or instruction fetch from a given memory location.
- IA-32 instruction fetches and loads to non-speculative memory occur in program order. IA-32 instruction cache line fetch accesses to uncached memory occur in the order specified by an in-order execution of the program. Note however that the same cache line may be fetched multiple times by the processor as multiple instructions within the cache line are executed. The size of a cache line and number of instruction fetches is model specific.
- IA-32 instruction fetches are not perceived as passing prior IA-32 stores. IA-32 stores into the IA-32 instruction stream are observed by the processor's self modifying code logic. Speculative instruction fetches may be emitted by the processor before a store is seen to the instruction stream and then discarded. Self modifying code due to Itanium stores is not detected by the processor.
- IA-32 instruction fetches can pass prior loads or memory fence operations from the same processor. Data memory accesses, and memory fences are not ordered with respect to IA-32 instruction fetches.

- IA-32 instruction fetches can not pass any serializing instructions, including Itanium `srlz.i` and IA-32 CPUID. For speculative memory types the processor may prefetch ahead of a serialization operation and then discard the prefetched instructions.
- IA-32 serializing operations wait for all previous stores and loads to complete, and for all prior stores buffered by the processor to become visible. IA-32 serializing instructions include CPUID.
- IA-32 OUT instructions may be buffered, however the processor will not start execution of the next IA-32 instruction until the OUT has completed (been accepted by the platform).
- The processor does not begin execution of the next IA-32 instruction until the IN or OUT has been completed (accepted) by the platform. This statement does not apply for Itanium memory references to the I/O port space. The processor may issue instruction fetches and VHPT walks ahead of an IN or OUT.
- VHPT Walks are speculative and can occur at any time. VHPT walks can pass all prior IA-32 loads, stores, instruction fetches, I/O operations and serializing instructions.

10.6.10.1 Instruction Set Transitions

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

10.6.10.1.1Transitions from Intel® Itanium™ Instruction Set to IA-32 Instruction Set

- All data dependencies are honored, IA-32 loads see the results of all prior Itanium and IA-32 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-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.

10.6.10.1.2Transitions from IA-32 Instruction Set to Intel® Itanium™ Instruction Set

- All data dependencies are honored, Itanium loads see the results of all prior Itanium and 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-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.

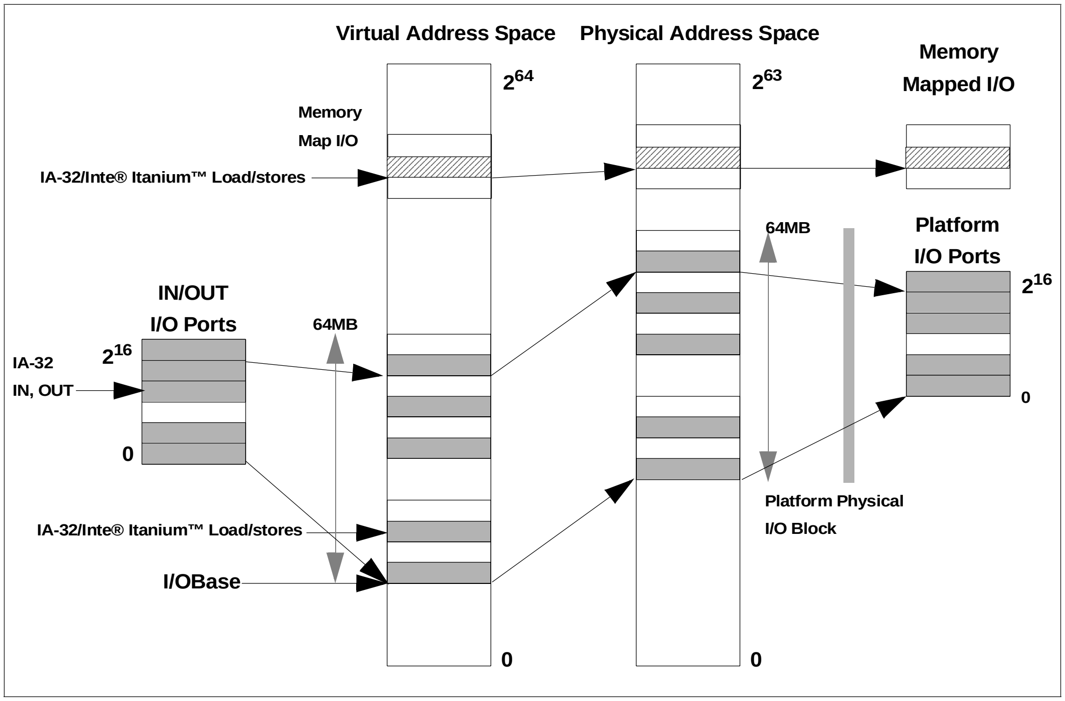
10.7 I/O Port Space Model

A consistent unified addressing model is used for both IA-32 and Itanium references to the I/O port space. On prior IA-32 processors two I/O models exist; memory mapped I/O and the 64KB I/O port space. On processors based on the Itanium instruction set, the 64KB I/O port space defined by

IA-32 processors is effectively mapped into the 64-bit virtual address space of the processor, producing a single memory mapped I/O model as shown in Figure 10-7. This model allows Itanium normal load and store instructions to also access the I/O port space.

Itanium-based operating system code can directly control IA-32 IN, OUT instruction and accessibility by IA-32 or Itanium load/store instructions to blocks of 4 virtual I/O ports using the TLBs. The entire range of virtual memory mechanisms defined by the Itanium architecture: access rights, dirty, access bits, protection keys, region identifiers can be used to control permission and addressability.

Figure 10-6. I/O Port Space Model



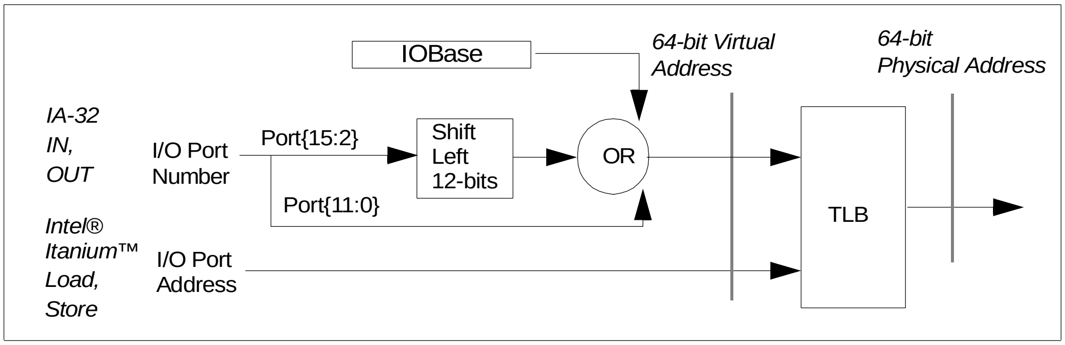
In the Itanium System Environment, the virtual location of the 64MB I/O port space is determined by operating system. For IA-32 IN and OUT instructions, the operating system can specify the virtual base location via the I/O base register.

Any IA-32 or Itanium load or store within the virtual region mapped by the operating system to the platform's physical 64MB I/O port block, directly accesses physical I/O devices within the I/O port space. The location of the 64MB I/O port block within the 2^{63} byte physical address space is determined by platform conventions, see "Physical I/O Port Addressing" for details.

10.7.1 Virtual I/O Port Addressing

The IA-32 defined 64-KB I/O port space is expanded into 64MB. This effectively places 4 I/O ports per each 4KB virtual and physical page. Since there are 4 ports per virtual page, the TLBs can be used port address translation, and permission checks as shown in Figure 10-7.

Figure 10-7. I/O Port Space Addressing



For IA-32 IN and OUT instructions a port's virtual address is computed as:

```
port_virtual_address = IOBase | (port{15:2}<<12) | port{11:0}
```

This address computation places 4 ports on each 4K page and expands the space to 64MB, with the ports being at a relative offset specified by port{11:0} within each 4K-byte virtual page. IOBase is a kernel register *(KR)* maintained by the operating system that points to the base of the 64MB Virtual I/O port space. *The value in IOBase must be aligned on a 64MB boundary otherwise port address aliasing will occur and processor operation is undefined.*

For Itanium load and stores accesses to the I/O port space, a port's virtual address can be computed in the same manner, specifically.

```
port_virtual_address = IOBase | (port{15:2}<<12) | port{11:0}
```

In practice this address is a constant for any given physical I/O device.

Software Warning: In the generation of the I/O port virtual address, software MUST ensure that port_virtual_address{11:2} are equal to port{11:2} bits. Otherwise, some processors implementations may place the port data on the wrong bytes of the processor's bus and the port will not be correctly accessed.

IA-32 IN and OUT instructions and Itanium or IA-32 load/store instructions can reference I/O ports in 1, 2, or 4-byte transactions. References to the legacy I/O port space cannot be performed with greater than 4 byte transactions due to bus limitations in most systems. Since an IA-32 IN/OUT instruction can access up to 4 bytes at port address 0xFFFF, the I/O port space effectively extends 3 bytes beyond the 64KB boundary. Operating systems can; 1) not map the excess 3 bytes, resulting in denial of permission for the excess 3 bytes, or 2) map via the TLB the excess 3 bytes back to port address 0 effectively wrapping the I/O port space at 64KB.

Operating system code can map each virtual I/O port space page anywhere within the physical address space using the Data Translation Registers or the Data Translation Cache. Large page translations can be used to reduce the number of mappings required in the TLB to map the I/O port space. For example, one 64MB translation is sufficient to map the entire expanded 64MB I/O port space. The **UC memory attribute** must be used for all I/O port space mappings to avoid speculative processor references to I/O devices, otherwise processor and platform operation is undefined.

Operating System Warning: Operating system code can not remap a given port to another port address within the I/O port space, such that port_physical_address{21:12} != port_physical_address{11:2}. Otherwise, based on the processor model, I/O port data may be placed on the wrong bytes of the processor's bus and the port will not be correctly accessed.

I/O port space breakpoints can be configured by loading the address and mask fields with the virtual address defined by the operating system to correspond to the I/O port space.

The processor (as defined in the next section) ensures that load, store references will not result in references to I/O devices for which permission was not granted.

All memory related faults defined in Chapter 5, "Interruptions" can be generated by IA-32 IN and OUT references to the I/O port space, including IA-32_Exception(Debug) traps for data address breakpoints and IA-32_Exception(AlignmentCheck) for unaligned references. (EFLAG.ac enabled unaligned port references are not detected by the processor). Itanium Data Breakpoint registers (DBRs) can be configured to generate debug traps for references into the I/O port space by either IA-32 IN/OUT instructions or by IA-32 or Itanium load/store instructions.

10.7.2 Physical I/O Port Addressing

Some processors implementations will provide an M/IO pin or bus indication by decoding physical addresses if references are within the 64MB physical I/O block. If so the 64MB I/O port space is compressed back to 64KB. Subsequent processor implementations may drop the M/IO pin (or bus indication) and rely on platform or chip-set decoding of a range of the 64MB physical address space.

Through the PAL firmware interface, the 64MB physical I/O block can be programmed to any arbitrary physical location. It is suggested that to be compatible with IA-32 based platforms, the platform physical location of the physical I/O block be programmed above 4G-bytes and above all useful DRAM, ROM and existing memory mapped I/O areas. See PAL_PLATFORM_ADDR on page 2:352 for details.

Based on the platform design, some platforms can accept addresses for the expanded 64MB I/O block, while other platforms will require that the I/O port space be compressed back to 64KB by the processor. If the I/O port space needs to be compressed either the processor or platform (based on the implementation) will perform the following operation for all memory references within the physical I/O block.

```
IO_address{1:0} = PA{1:0}
IO_address{15:2} = PA{25:12}  // byte strobes are generated from the lower I/O_address bits
```

The processor ensures that the bus byte strobes and bus port address are derived from PA{25:12,1:0}. Thus allowing an operating system to control security of each 4 ports via TLB management of PA{25:12}.

10.7.2.1 I/O Port Addressing Restrictions

For the 64MB physical I/O port block the following operations are undefined and may result in unpredictable processor operation; references larger than 4-bytes, instruction fetch references, references to any memory attribute other than UC, or semaphore references which require an atomic lock. To ensure I/O ports accesses are not granted for which the TLB has not been consulted, the processor ensures:

- All byte addresses within the same 4KB page alias to the 4 ports defined by the mapped physical I/O port page.
- All IA-32 and Itanium unaligned loads and stores that cross a 4-byte boundary to the processor's physical I/O port block are truncated. That is the bus transaction to the area before the 4-byte boundary is performed (the number of bytes emitted is model specific). No bus transaction is performed for the bytes that are beyond the 4-byte boundary. 4-byte crosser loads while return some undefined data. 4-byte crosser stores will not write all intended bytes.
- For IA-32 IN/OUT accesses that cross a 4-port boundary the processor will break the operation into smaller 1, 2, or 3 byte I/O port transactions within each 4KB virtual page.

10.7.3 IA-32 IN/OUT instructions

IA-32 I/O instructions (IN, OUT, INS, OUTS) defined in the *Intel Architecture Software Developer's Manual* are augmented as follows:

- I/O instructions first check for IOPL permission. If PSR.cpl<=EFLAG.iopl, access permission is granted. Otherwise the TSS I/O permission bitmap may be consulted as defined below. If the Bitmap denies permission or is not consulted an IA-32_Exception(GPFault) is generated.
- If IOPL permission is denied and CFLG.io is 1, the TSS I/O permission bitmap is consulted for access permission. If the corresponding bit(s) for the I/O port(s) is 1, indicating permission is denied, a GPFault is generated. Otherwise access permission is granted. The TSS I/O permission bitmap provides 1 port permission control at the expense of additional processor data memory references. This mechanism can be used in the Itanium System Environment, but is not recommended since TLB access controls defined by the Itanium architecture are faster and provide a consistent control mechanism for both IA-32 and Itanium-based code. Whereas, the TLB mechanism provides a control mechanism for both IA-32 and Itanium memory references.
- If CFLG.io is 0, the TSS I/O permission bitmap is not consulted and if the IOPL check failed an IA-32_Exception(GPFault) is generated. By setting CFLG.io to 0, operating system code can disable all processor references to the TSS. By setting IOPL<PSR.cpl and setting CFLG.io to 0, operating system code can block all user level execution of IA-32 I/O instructions, no TSS needs to be allocated or defined by the operating system.
- I/O port references generate a virtual port address relative to the IOBase register as defined in"Virtual I/O Port Addressing"
- If data translations are enabled, the TLB is consulted for the required virtual to physical mapping. If the required mapping is not present a VHPT Translation, Data TLB Miss or Alternative Data TLB Miss fault is generated.
- If data translations are enabled, Access Rights, Permission Keys, Access, Dirty and Present bits are checked and faults generated.
- If data translations are disabled (PSR.dt is 0) or the referenced I/O port is mapped to an unimplemented virtual address (via the IOBase register), a GPFault is raised on the referencing IA-32 IN, OUT, INS, or OUTS instruction.

- Alignment and Data Address breakpoints are also checked and may result in an IA-32_Exception(AlignmentCheck) fault (if PSR.ac is 1) or IA-32_Exception(Debug) trap.
- If an IA-32 IN/OUT I/O port Accesses cross a 4-port boundary the processor will break the operation into smaller 1, 2, or 3 byte transactions.
- Assuming no faults, a physical transaction is emitted to the mapped or specified physical address.

The processor ensures that IA-32 IN, INS, OUT, OUTS references are fully ordered and will not allow prior or future data memory references to pass the I/O operation as defined in "IA-32 Memory Ordering". The processor will wait for acceptance for IN and OUT operations before proceeding with subsequent externally visible bus transactions.

10.7.4 I/O Port Accesses by Loads and Stores

If an access is made to the I/O port block using IA-32 or Itanium loads and stores the following differences in behavior should be noted; EFLAG.iopl permission is not checked, TSS permission bitmap is not checked, and stores and loads do not honor IN and OUT memory ordering and acceptance semantics (the processor will not automatically wait for a store to be accepted by the platform).

Virtual addresses for the I/O port space should be computed as defined in "Virtual I/O Port Addressing" If data translations are enabled, the TLB is consulted for mappings and permission, and the resulting mapped physical address used to address the physical I/O device.

If IA-32 ordering semantics are required to a particular I/O port device (or memory mapped I/O device), IA-32 or Itanium-based software must enforce ordering to the I/O device. Software can either perform a memory ordering fence before and after the transaction, or use an load acquire or store release

To ensure the processor does not speculatively access an I/O device, all I/O devices must be mapped by the UC memory attribute.

If IA-32 acceptance semantics are required (i.e. additional data memory transactions are not initiated until the I/O transaction is completed), Itanium-based code can issue a memory acceptance fence. Conversely, if certain I/O devices do not require IA-32 IN/OUT ordering or acceptance semantics, Itanium-based code can relax ordering and acceptance requirements as shown below.

OUT

```
[mf]/  /Fence prior memory references, if required

add port_addr = IO_Port_Base, Expanded_Port_Number
st.rel (port_addr), data
[mf.a] //Wait for platform acceptance, if required
[mf]   //Fence future memory operations, if required
```

IN

```
[mf]   //Fence prior memory references, if required
add port_addr = IO_Port_Base, Expanded_Port_Number
ld.acq data, (port_addr)
[mf.a] //Wait for platform acceptance, if required
[mf]   //Fence future memory references, if required
```

10.8 Debug Model

The debug facilitates defined by the Itanium architecture are designed to support debugging of both the Itanium and IA-32 instruction set. The following debug events can be triggered during IA-32 instruction set execution by Itanium debug resources.

- **Single Step trap** - When PSR.ss is 1 (or EFLAG.tf is 1), successful execution of each IA-32 instruction, results in an IA-32_Exception(Debug) trap. After the single step trap, IIP points to the next IA-32 instruction to be executed.
- **Breakpoint Instruction trap** - execution of INT 3 (breakpoint) instruction results in a IA-32_Exception(Debug) trap.
- **Instruction Debug fault** - When PSR.db is 1 and PSR.id is 0 and EFLAG.rf is 0, any IA-32 instruction fetch that matches the parameters specified by the IBR registers results in an IA-32_Exception(Debug) fault. After servicing a Debug fault, debuggers can set PSR.id (or EFLAG.rf for IA-32 instructions) before restarting the faulting instruction. If PSR.id is 1, Instruction Debug faults are temporarily disabled for one Itanium instruction. If PSR.id is 1 or EFLAG.rf is 1, Instruction Debug faults are temporarily disabled for one IA-32 instruction. The successful execution of an IA-32 instruction clears both PSR.id and EFLAG.rf bits. The successful execution of an Itanium instruction only clears PSR.id.
- **Data Debug traps** - When PSR.db is 1, any IA-32 data memory reference that matches the parameters specified by the DBR registers results in a IA-32_Exception(Debug) trap. IA-32 data debug events are traps, not faults as defined for Itanium instruction set data debug events. Trap behavior is required since any given IA-32 instruction can access several memory locations during its execution. The reported trap code returns the match status of the first four DBR registers that matched during the execution of the IA-32 instruction. Zero, one or DBR registers may be reported as matching.
- **Taken Branch trap** - When PSR.tb is 1, a IA-32_Exception(Debug) trap occurs on every IA-32 taken branch instruction (CALL, Jcc, JMP, RET, LOOP). After the trap, IIP points to the branch target.
- **Lower Privilege Transfer trap** - Does not occur during IA-32 instruction set execution.

For virtual memory accesses, breakpoint address registers contain the virtual addresses of the debug breakpoint. For physical accesses, the addresses in these registers are treated as a physical address. Software should be aware that debug registers configured to fault on virtual references, may also fault on a physical reference if translations are disabled. Likewise a debug register configured for physical references can fault on virtual references that match the debug breakpoint registers.

10.8.1 Data Breakpoint Register Matching

Each Itanium data breakpoint register has the following matching behavior for IA-32 instruction set data memory references:

- **DBR.addr** - IA-32 single or multi-byte data memory references that access ANY memory byte specified by the DBR address and mask fields results in a debug breakpoint trap regardless of datum size and alignment. The upper 32-bits of DBR.addr must be zero to detect IA-32 data memory references. Since IA-32 data breakpoints are traps, all processor implementations ensure data breakpoint traps are precise. Traps are only reported if any byte in the data memory reference ANDed with the DBR mask bitwise matches the DBR address field ANDed with the DBR mask. No spurious data breakpoint faults are generated for IA-32 data memory operands that are unaligned, nor are matches reported if no bytes of the operand lie within the address range specified by the DBR address and mask fields. Note, Itanium instruction set generated unaligned data memory references may result in spurious imprecise breakpoint faults.

- **DBR.mask** - by programming the mask a breakpoint range of 1, 2, 4, 8, or any power of 2 combination can be supported. Mask bits above bit 31 are checked by the processor during IA-32 data memory references
- **trap code B bits** - are set indicating a match with the corresponding data breakpoint register DBR0-3. For IA-32 data debug traps, any number of B-bits can be set indicating a match.

The B-bits are only set and a data breakpoint trap generated if 1) the breakpoint register precisely matches the specified DBR address and mask, 2) it is enabled by the DBR read or write bits for the type of the memory transaction, 3) the DBR privilege field matches PSR.cpl, 4) PSR.db is 1, and 5) no other higher priority faults are taken.

I/O port space breakpoints can be configured by loading the address and mask fields with the virtual address defined by the operating system to correspond to the I/O port space.

10.8.2 Instruction Breakpoint Register Matching

Each Itanium instruction breakpoint register has the following matching behavior for IA-32 instruction set memory fetches:

- **IBR.addr** - an IBR register matches an IA-32 instruction fetch address, if the first byte of an IA-32 instruction address ANDed with the IBR mask bitwise matches the IBR address field ANDed with the IBR mask. Note that only the first byte is analyzed. The upper 32-bits of IBR.addr must be zero to detect IA-32 instruction fetches.
- **IBR.mask** - by programming the mask a breakpoint range of 1, 2, 4, 8, or any power of 2 combination can be supported. Mask bits above bit 31 are ignored during IA-32 instruction fetches.

The instruction breakpoint fault is generated if 1) the breakpoint register precisely matches the specified IBR address and mask, 2) it is enabled by the IBR execute bit, 3) the IBR privilege field matches PSR.cpl, 4) PSR.db is 1, 5) PSR.id is 0, and 6) no other higher priority faults are taken.

10.9 Interruption Model

Within the Itanium System Environment, all interruptions originating out of the IA-32 or Itanium instruction sets are delivered to Itanium-based Interruption Handlers within the Itanium-based operating system. Virtual memory management faults, machine checks, and external interrupts are always delivered to Itanium-based interruption handlers regardless of the instruction set running at the time of the interruption. IA-32 exceptions, control transfers through gates, task switches, and accesses to sensitive IA-32 system resources are intercepted into Itanium-based interruption handlers. Using these intercepts, Itanium-based software can implement specific policies with regard to that resource. Policies may include virtualization, emulation of an IA-32 opcode or memory access, or various permission policies.

In general, if an interruption is independent of the executing instruction set (such as an external interruption or TLB fault) common Itanium-based handlers are invoked. For classes of exceptions and intercept conditions that are specific to the IA-32 instruction set, three special Itanium-based software handlers are invoked to deal with IA-32 instruction set interruptions. Table 10-8 shows the 3 interruption handlers defined to support IA-32 events. See “IA-32 Interruption Vector Definitions” for details on these interruption handlers.

Table 10-8. IA-32 Interruption Vector Summary

Handler	Description
IA_32_Intercept	Intercepted IA-32 instructions, I/O, system flag manipulation and gate transfers.
IA-32_Exception	IA-32 instruction set generated exceptions.
IA_32_Interrupt	IA-32 instruction set generated software interrupts

This grouping of interruption handlers simplifies software handlers such that they do not need to be concerned with behavior of both IA-32 and Itanium instruction sets.

Interruption registers (defined in Chapter 3) record the state of IA-32 execution at the point of interruption. For IA-32 exceptions, ISR contains IA-32 defined error codes and vector numbers as defined by the *Intel Architecture Software Developer’s Manual*. IA-32 instruction set related exceptions and software interruptions vector directly through the interruption mechanism defined by the Itanium architecture without consulting the IA-32 IDT or performing any memory stack pushes.

10.9.1 Interruption Summary

Table 10-9 summarizes the set of all IA-32 interruptions and how they are mapped to Itanium-based interruption handlers within the Itanium System Environment. See Chapter 9 and Chapter 8 for a detailed definition of each interruption.

Table 10-9. IA-32 Interruption Summary

IA-32 Vector	Itanium™-based Interruption Handler	ISR Vect	ISR Code	Description
IA-32 Defined Interruptions				
0	IA-32_Exception (Divide)	0	0	IA-32 divide by zero fault.
1	IA-32_Exception (Debug)	1	0	IA-32 instruction breakpoint fault.
1	IA-32_Exception (Debug)	1	TrapCode	IA-32 debug events. The Trap Code indicates concurrent taken branch, data breakpoint and single step trap conditions.
2	External Interrupt	0	0	NMI is delivered through the Intel® Itanium™ External Interrupt vector.
3	IA-32_Exception(Break)	3	TrapCode	IA-32 INT 3 instruction.
4	IA-32_Exception(INTO)	4	TrapCode	IA-32 INTO detected overflow trap.
5	IA-32_Exception (Bound)	5	0	IA-32 BOUND range exceeded fault.
6	IA-32_Intercept(Inst)	0	InterceptCode	All IA-32 unimplemented and illegal opcodes.
7	IA-32_Exception(DNA)	7	0	IA-32 Device not available fault.
8	--	na		IA-32 Double fault can not be generated in the Intel® Itanium™ System Environment, Intel reserved.
9	--	na		Intel reserved

Table 10-9. IA-32 Interruption Summary (Continued)

IA-32 Vector	Itanium™-based Interruption Handler	ISR Vect	ISR Code	Description
10	--	na		IA-32 Invalid TSS fault can not generated in the Intel® Itanium™ System Environment, Intel reserved,
11	IA-32_Exception(NotPresent)	11	ErrorCode[a]	IA-32 Segment Not present fault.
12	IA-32_Exception (Stack)	12	ErrorCode	IA-32 Stack Exception fault.
13	IA-32_Exception (GPFault)	13	ErrorCode	IA-32 General Protection fault.
14	Intel® Itanium™ TLB faults	see Data TLB faults below		IA-32 Page fault can not be generated in the Intel® Itanium™ System Environment, replaced by Intel® Itanium™ TLB faults, Intel reserved,
15	--	na		Intel reserved.
16	IA-32_Exception (FPError)	16	0	IA-32 floating-point fault.
17	IA-32_Exception(AlignCheck)	17	0	IA-32 un-aligned data references.
18	Corrected MCHK	na		IA-32 Machine Check can not be generated in the Intel® Itanium™ System Environment, replaced by the PAL Machine Check Architecture, Intel reserved.
19	IA-32_Exception (StreamSIMD)	19	0	IA-32 Streaming SIMD Extension Numeric Error fault.
20-31	--	na		Intel reserved.
0-255	External Interrupt	0	0	External interrupts are delivered through the Intel® Itanium™ External Interrupt vector. Software must read the IVR register to determine the vector number.
0-255	IA-32_Interrupt (vector #)	Vect#	TrapCode	IA-32 Software Interrupt trap. ISR contains the vector number.

IA-32 Interceptions				
	IA-32_Intercept(Inst)	0	InterceptCode	Intercept for unimplemented, illegal or privileged IA-32 opcodes.
	IA-32_Intercept(Gate)	1	TrapCode	Intercept for control transfers through a Call Gate, Task gate or Task Segment.
	IA-32_Intercept(SystemFlag)	2	TrapCode	Intercept for modification of system flag values.
	IA-32_Intercept(Lock)	4	0	IA-32 semaphore operation requires an external bus lock when DCR.lc is 1.
		3,5-255	--	Intel reserved

[a] The IA-32 Error Code is defined as a Selector Index, and TI, IDT and EXT bits are set based on the exception. See *Intel Architecture Software Developer's Manual* for the complete definition.

10.9.2 IA-32 Numeric Exception Model

IA-32 numeric instructions follow the IA-32 delayed floating-point exception model. Specifically IA-32 numeric exceptions are held pending until the next IA-32 numeric instruction or MMX technology instruction as defined in the *Intel Architecture Software Developer's Manual*. Numeric faults generated on Streaming SIMD Extension instructions are reported precisely on the faulting Streaming SIMD Extension instruction. Streaming SIMD Extension instructions do NOT trigger the report of pending IA-32 numeric exceptions.

For voluntary transitions out of the IA-32 instruction, an implicit FWAIT operation is performed by the `jmpe` instruction to ensure all pending numeric exceptions are reported. For involuntary transitions out of the IA-32 instruction set (external interruptions, TLB faults, exceptions, etc.) the processor does not perform a FWAIT operation. However, every IA-32 numeric instruction that generates a pending numeric exception loads the application registers FSR, FIR, and FDR with the IA-32 floating-point state on the instruction that generating the exception. This state contains information defined by the IA-32 FSTENV and FLDENV instructions. During a process context switch, the operating system must save and restore FSR, FIR, and FDR (effectively performing an FSTENV and FLDENV) to ensure numeric exceptions are correctly reported across a process switch.

10.10 Processor Bus Considerations for IA-32 Application Support

The section briefly discusses bus and platform considerations when supporting IA-32 applications in the Itanium System Environment.

Itanium-based code does not assert the SPLCK and LOCK pins. The LOCK pin is used by IA-32 code to signal an external atomic bus transaction for which atomicity cannot be enforced within the processor's caches, whereas, SPLCK indicates if an unaligned external bus lock requires a split lock operation and hence several bus transactions. For IA-32 code, if the platform does not support LOCK or SPLCK, the operating system must disable external bus lock transactions by setting DCR.lc to 1. When DCR.lc is 1, any IA-32 atomic reference not serviced internally in the processor's caches results in an IA-32_Intercept(Lock) fault. See "Default Control Register (DCR – CR0)" for details. When DCR.lc is 0, operating system code is responsible for emulation of the IA-32 instruction and ensuring atomicity (if required).

The A20M and IGNE pins are ignored in the Itanium System Environment. FERR is not asserted in the Itanium System Environment.

In both IA-32 and Itanium System Environments, the M/IO pin (or an external bus indication) is asserted by any memory reference to the 64MB I/O port block range of the physical address space. See Section 10.7 "I/O Port Space Model" for details.

SMI and the SMM environment are not supported on processors based on the Itanium architecture. The PMI interrupt and PAL firmware environment replace them. See Section 11.5 "Platform Management Interrupt (PMI)" for details.

10.10.1 IA-32 Compatible Bus Transactions

Within the Itanium System Environment, the following bus transactions are initiated:

- INTA - Interrupt Acknowledge - emitted by the operating system (via a read to the INTA byte in the processor's Interrupt Block) to acquire the interrupt vector number from an external interrupt controller.
- HALT - Emitted when the processor has entered the halt state due to the operating system/ platform firmware calling PAL_HALT or PAL_HALT_LIGHT.
- SHUTDOWN - Emitted when the processor has entered the shutdown state. This can only be generated when the processor has entered into the IA-32 System Environment by calling PAL_ENTER_IA_32_ENV procedure call.
- STPACK - Stop Acknowledge. Emitted by calling an implementation specific PAL firmware procedure. See the processor specific firmware guide for more information.
- FLUSH - Emitted when the WBINVD or INVD instruction is executed when running in the IA-32 System Environment entered by calling PAL_ENTER_IA_32_ENV procedure call. Indicates that external caches (if any) should be invalidated.
- SYNC - Emitted when the WBINVD instruction is executed when running in the IA-32 System Environment entered by calling PAL_ENTER_IA_32_ENV procedure call. Indicates that external caches (if any) should copy all modified cache lines back to main memory.

Processor Abstraction Layer 11

This chapter defines the architectural requirements for the **Processor Abstraction Layer (PAL)** for all processors based on the Itanium architecture. It is intended for processor designers, firmware/BIOS designers, system designers, and writers of diagnostic and low level operating system software.

PAL is part of the Itanium processor architecture and its goal is to provide a consistent firmware interface to abstract processor implementation-specific features.

The objectives of this chapter are to define:

- The architectural behavior and interface requirements for processor testing, configuration and error recovery. This includes the hardware entrypoints into PAL and the PAL interfaces to platform firmware and system software.
- A set of boot and runtime PAL procedures to access processor implementation-specific hardware and to return information about processor implementation-dependent configuration.
- A computing environment for both PAL entrypoints and procedures such that:
  - Memory used by PAL procedures is allocated by the caller of PAL procedures.
  - PAL code runs little endian.
  - PAL interface is as endian neutral as possible.
  - PAL is Itanium-based code.
  - PAL code runs at privilege level 0.
  - PAL procedures can be called without backing store, except where memory based parameters are returned.
- The processor and platform hardware requirements for PAL. This includes minimizing PAL dependencies on platform hardware and clearly stating where those dependencies exist.
- A PAL interface and requirements to support firmware update and recovery.

11.1 Firmware Model

As shown in Figure 11-1, Itanium-based firmware consists of three major components: Processor Abstraction Layer (PAL), System Abstraction Layer (SAL), and Extensible Firmware Interface (EFI) layer. PAL, SAL, and EFI together provide processor and system initialization for an operating system boot. PAL and SAL provide machine check abort handling and other processor and system functions which would vary from implementation to implementation. The interactions of the various services that PAL, SAL, and EFI provide are shown in Figure 11-2.

In the context of this model and throughout the rest of this chapter, the System Abstraction Layer (SAL) is a firmware layer which isolates operating system and other higher level software from implementation differences in the platform, while PAL is the firmware layer that abstracts the processor implementation.

Figure 11-1. Firmware Model

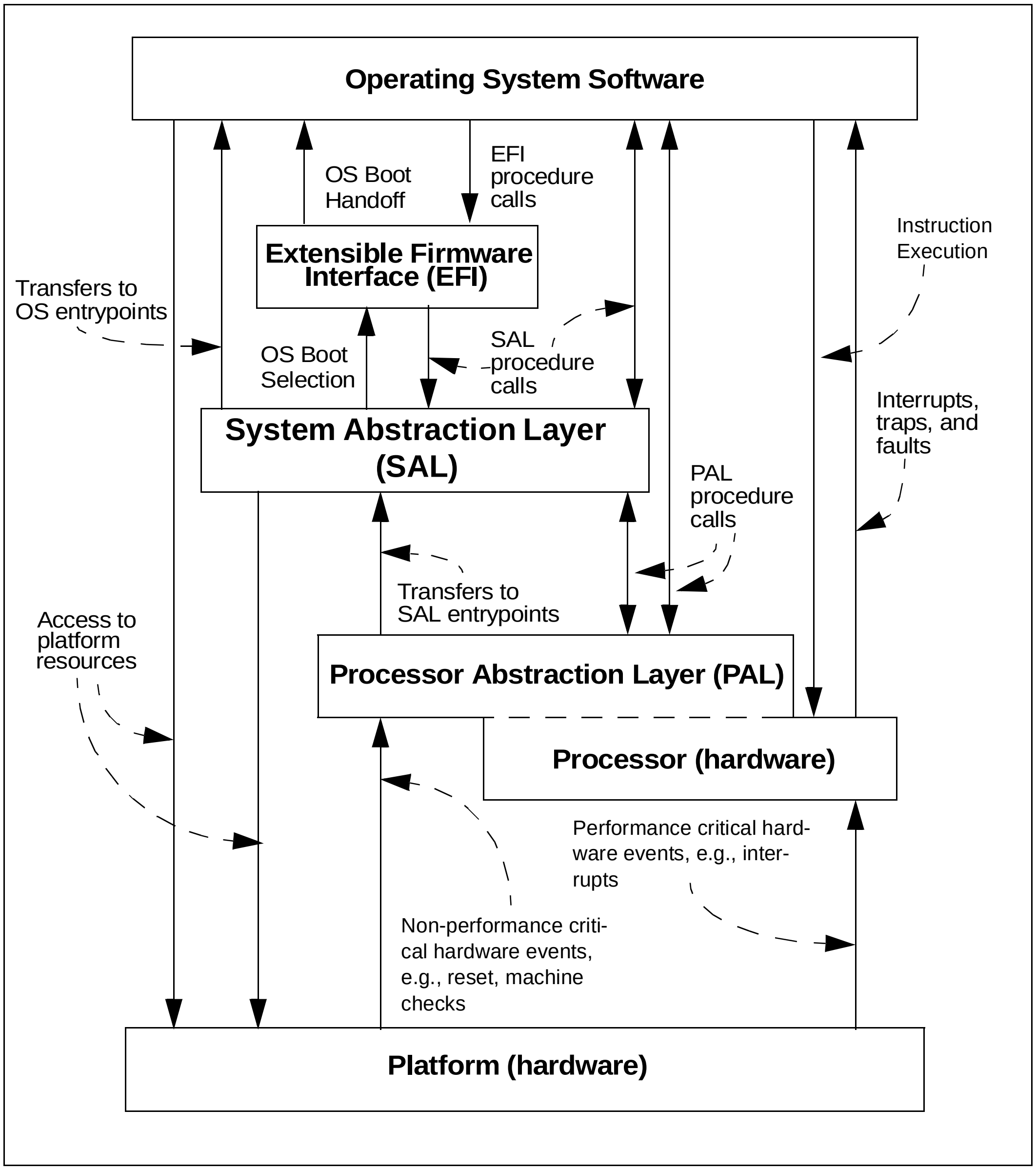
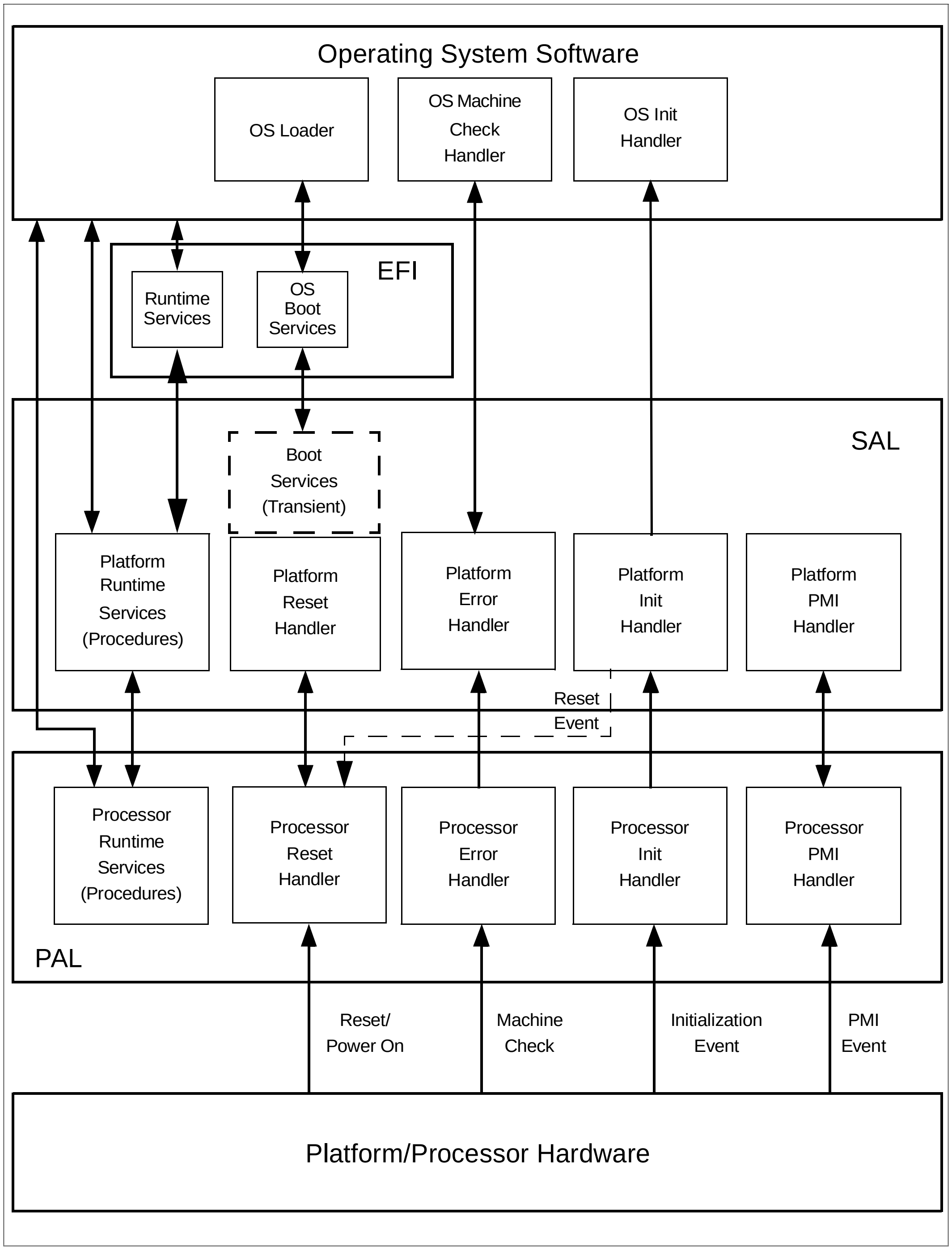

Operating System Software
OS Boot Handoff
EFI procedure calls
Instruction Execution
Extensible Firmware Interface (EFI)
Transfers to OS entrypoints
SAL procedure calls
OS Boot Selection
System Abstraction Layer (SAL)
Interrupts, traps, and faults
PAL procedure calls
Transfers to SAL entrypoints
Access to platform resources
Processor Abstraction Layer (PAL)
Processor (hardware)
Performance critical hardware events, e.g., interrupts
Non-performance critical hardware events, e.g., reset, machine checks
Platform (hardware)

Figure 11-2. Firmware Services Model



11.1.1 Processor Abstraction Layer (PAL) Overview

The purpose of the Processor Abstraction Layer, is to provide a firmware abstraction between the processor hardware implementation and system software and platform firmware, so as to maintain a single software interface for multiple implementations of the processor hardware. PAL is defined to be independent of the number of processors on a platform.

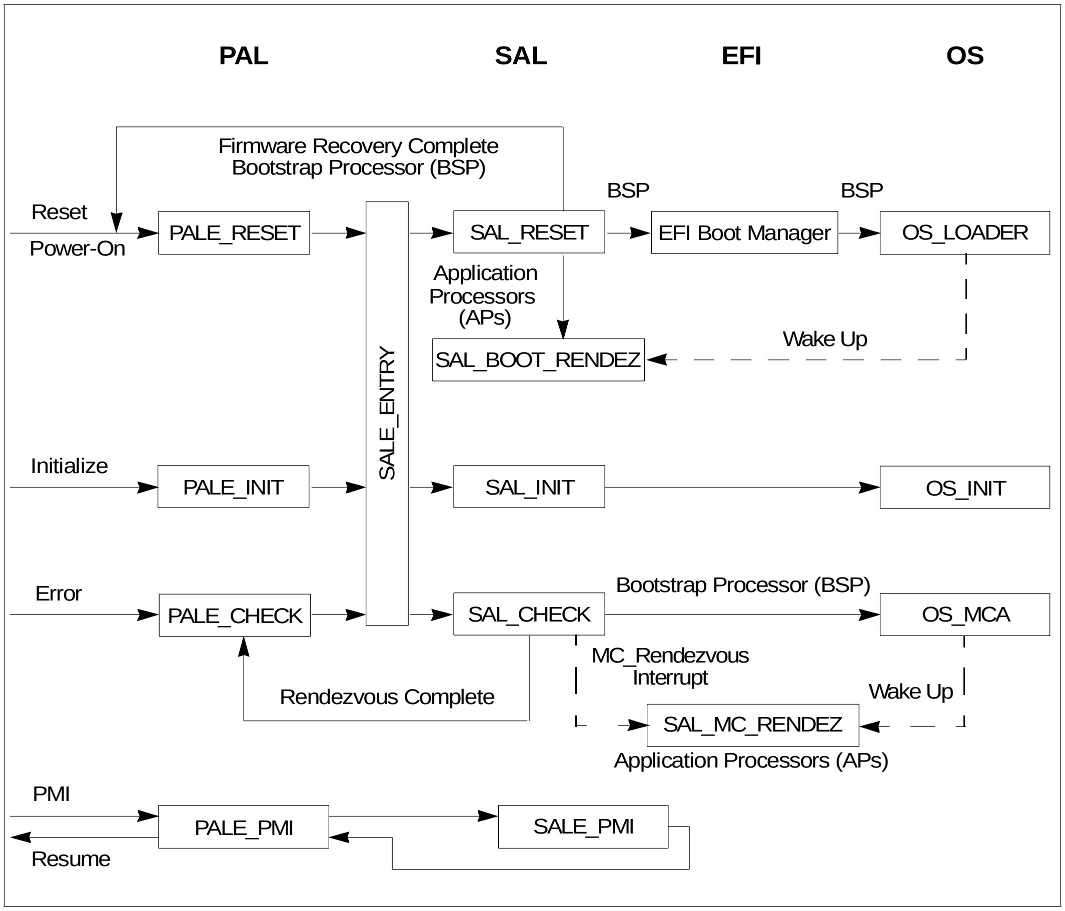
PAL encapsulates those processor functions that are likely to change on an implementation to implementation basis so that SAL firmware and operating system software can maintain a consistent view of the processor. These include non-performance critical functions dealing such as processor initialization, configuration and error handling.

PAL consists of two main components:

- Entrypoints, which are invoked directly by hardware events such as reset, init and machine checks. These interruption entrypoints perform functions such as processor initialization and error recovery.
- Procedures, which may be called by higher level firmware and software to obtain information about the identification, configuration, and capabilities of the processor implementation; to perform implementation-dependent functions such as cache initialization; or to allow software to interact with the hardware through such functions as power management or enabling/disabling processor features.

11.1.2 Firmware Entrypoints

Figure 11-3. Firmware Entrypoints Logical Model



11.1.3 PAL Entrypoints

The following hardware events can trigger the execution of a PAL entrypoint:

- Power-on/reset
- Hardware errors (both correctable and uncorrectable)
- Initialization event (via external interrupt bus message or processor pin)
- Platform management interrupt (via external interrupt bus message or processor pin)

These hardware events trigger the execution of one of the following PAL entrypoints: (as shown in Figure 11-3)

- PALE_RESET - Initializes and tests the processor following power-on or reset and then branches to SALE_ENTRY to determine whether to perform firmware recovery update, or to boot the machine for OS use. See Section 11.1.4.

- PALE_CHECK - Determines if errors are processor related, saves processor related error information and corrects errors where possible (for example, by flushing a corrupted instruction cache line and marking the cache line as unusable). In all cases, PALE_CHECK branches to SALE_ENTRY to complete the error logging, correction, and reporting.
- PALE_INIT - Saves the processor state, places the processor in a known state, and branches to SALE_ENTRY. PALE_INIT is entered as a response to an initialization event.
- PALE_PMI - Saves the processor state and branches to SALE_PMI. PALE_PMI is entered as a response to a platform management interrupt.

11.1.4 SAL Entrypoints

There are two entrypoints from PAL into SAL:

- SALE_ENTRY - PAL branches to this SAL entrypoint after a power-on, reset, machine check, or initialization event. If SALE_ENTRY was invoked by a machine check or initialization event, SALE_ENTRY branches to the appropriate routine:
    - SAL_CHECK is invoked after a machine check.
    - SAL_INIT is invoked after an initialization event.

  If SALE_ENTRY was invoked by a reset or power on, it checks to determine if a firmware recovery condition exists. If it does, SALE_ENTRY performs the firmware update, then performs a RESET operation to invoke PAL_RESET. If a recovery condition does not exist, SAL_ENTRY returns to PAL_RESET to complete processor self-test. PAL_RESET then branches back to SALE_ENTRY, which, in turn, branches to SAL_RESET.
- SALE_PMI - platform management interrupt. PALE_PMI branches to this SAL entrypoint after saving processor state in response to the platform management interrupt.

11.1.5 OS Entrypoints

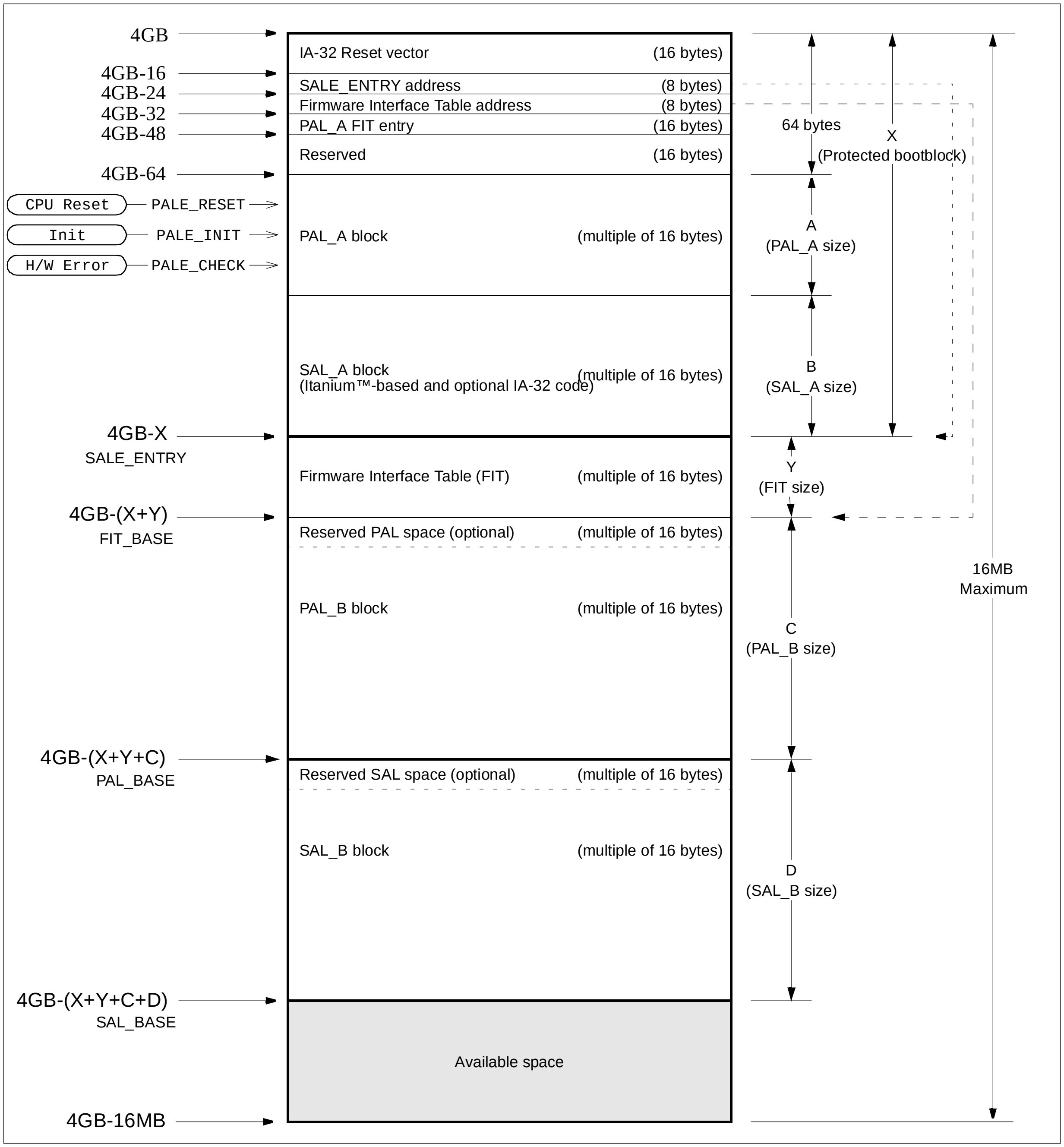
There are several entrypoints from SAL into an operating system (or equivalent software). Entrypoints from SAL into the operating system are expected to meet the following model:

- OS_BOOT - Operating System Boot interface.
- OS_MCA - Operating System Machine Check Abort Handler.
- OS_INIT - Operating System Initialization Handler.
- OS_RENDEZ - Operating System Multiprocessor Rendezvous interface.

11.1.6 Firmware Address Space

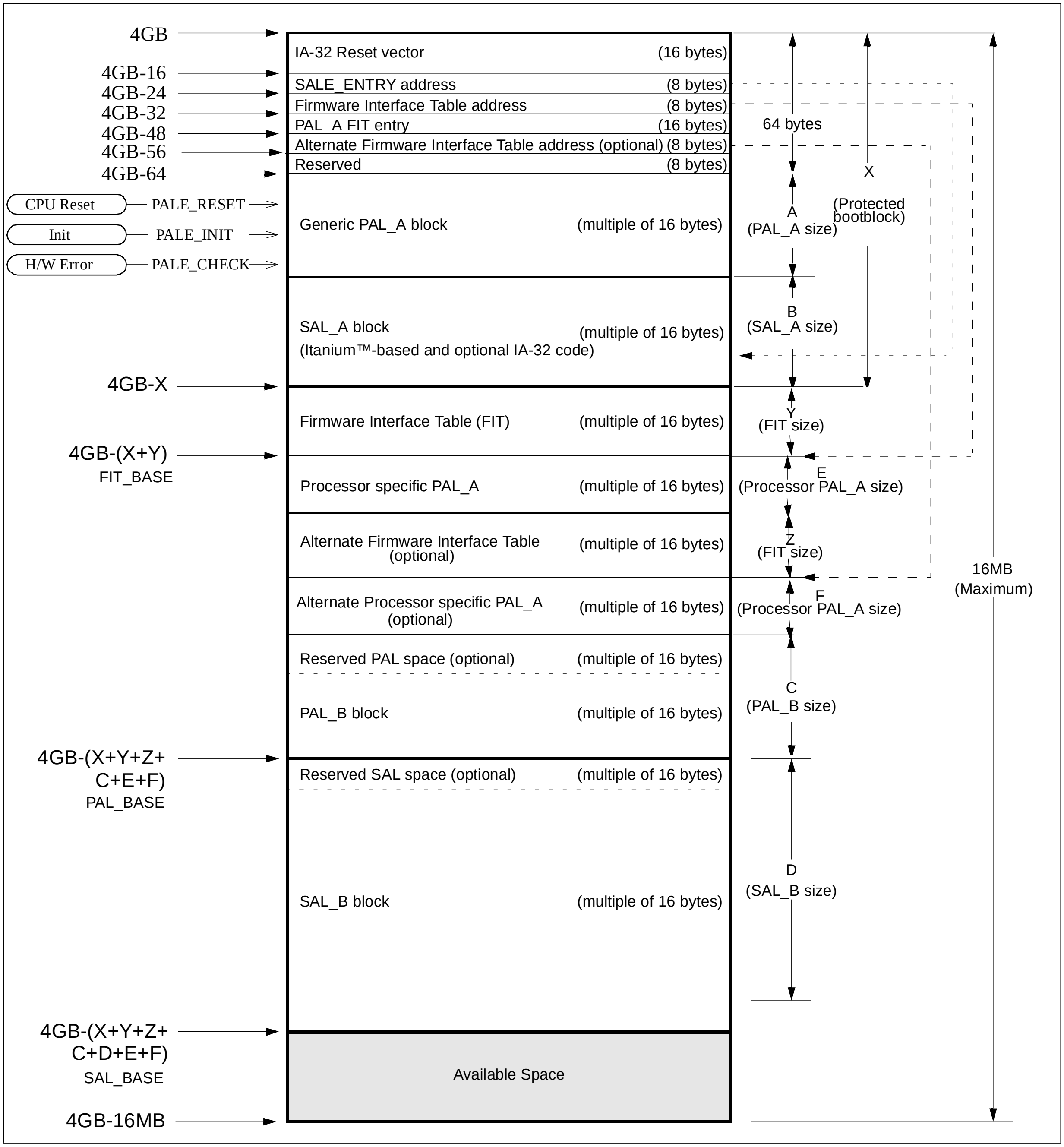
The firmware address space occupies the 16 MB region between 4 GB - 16 MB and 4 GB (addresses 0xFF00_0000 through 0xFFFF_FFFF). There are two primary layouts of this address space. The first version is shown in Figure 11-4 and the second version is shown in Figure 11-5. The first version has one PAL_A component. This layout allows for robust recovery of PAL_B and SAL_B components. This layout is useful for cases where PAL_A will not need to be upgraded. The second version splits the PAL_A block into two components. The first component is referred to as the generic PAL_A and the second component is the processor specific PAL_A. Splitting the PAL_A up in this manner allows for a robust upgrade of the processor specific PAL_A firmware as well as the PAL_B and SAL_B components. This is very useful if a platform is designed to support multiple processor generations which would require a PAL_A upgrade when the new processor generation is released. The generic PAL_A which resides in the Protected Boot Block will work across processor generations for a given platform. The processor specific PAL_A resides outside the Protected Boot Block and works for a specific processor generation.

Figure 11-4. Firmware Address Space



4GB
4GB-16
4GB-24
4GB-32
4GB-48
4GB-64
CPU Reset
PALE_RESET
Init
PALE_INIT
H/W Error
PALE_CHECK
4GB-X
SALE_ENTRY
4GB-(X+Y)
FIT_BASE
4GB-(X+Y+C)
PAL_BASE
4GB-(X+Y+C+D)
SAL_BASE
4GB-16MB
IA-32 Reset vector (16 bytes)
SALE_ENTRY address (8 bytes)
Firmware Interface Table address (8 bytes)
PAL_A FIT entry (16 bytes)
Reserved (16 bytes)
PAL_A block (multiple of 16 bytes)
SAL_A block
(Itanium™-based and optional IA-32 code) (multiple of 16 bytes)
Firmware Interface Table (FIT) (multiple of 16 bytes)
Reserved PAL space (optional) (multiple of 16 bytes)
PAL_B block (multiple of 16 bytes)
Reserved SAL space (optional) (multiple of 16 bytes)
SAL_B block (multiple of 16 bytes)
Available space
64 bytes
X
(Protected bootblock)
A
(PAL_A size)
B
(SAL_A size)
Y
(FIT size)
C
(PAL_B size)
D
(SAL_B size)
16MB
Maximum

Figure 11-5. Firmware Address Space with Processor-specific PAL_A Components


4GB
4GB-16
4GB-24
4GB-32
4GB-48
4GB-56
4GB-64
IA-32 Reset vector (16 bytes)
SALE_ENTRY address (8 bytes)
Firmware Interface Table address (8 bytes)
PAL_A FIT entry (16 bytes)
Alternate Firmware Interface Table address (optional) (8 bytes)
Reserved (8 bytes)
64 bytes
X
(Protected bootblock)
CPU Reset
PALE_RESET
Init
PALE_INIT
H/W Error
PALE_CHECK
Generic PAL_A block (multiple of 16 bytes)
A
(PAL_A size)
SAL_A block
(Itanium™-based and optional IA-32 code)
(multiple of 16 bytes)
B
(SAL_A size)
4GB-X
Firmware Interface Table (FIT) (multiple of 16 bytes)
Y
(FIT size)
4GB-(X+Y)
FIT_BASE
Processor specific PAL_A (multiple of 16 bytes)
E
(Processor PAL_A size)
Alternate Firmware Interface Table (optional) (multiple of 16 bytes)
Z
(FIT size)
Alternate Processor specific PAL_A (optional) (multiple of 16 bytes)
F
(Processor PAL_A size)
16MB
(Maximum)
Reserved PAL space (optional) (multiple of 16 bytes)
PAL_B block (multiple of 16 bytes)
C
(PAL_B size)
4GB-(X+Y+Z+
C+E+F)
PAL_BASE
Reserved SAL space (optional) (multiple of 16 bytes)
SAL_B block (multiple of 16 bytes)
D
(SAL_B size)
4GB-(X+Y+Z+
C+D+E+F)
SAL_BASE
Available Space
4GB-16MB

The firmware address space is shared by SAL and PAL. Some of the SAL/PAL boundaries are implementation dependent. The address space contains the following regions and locations.

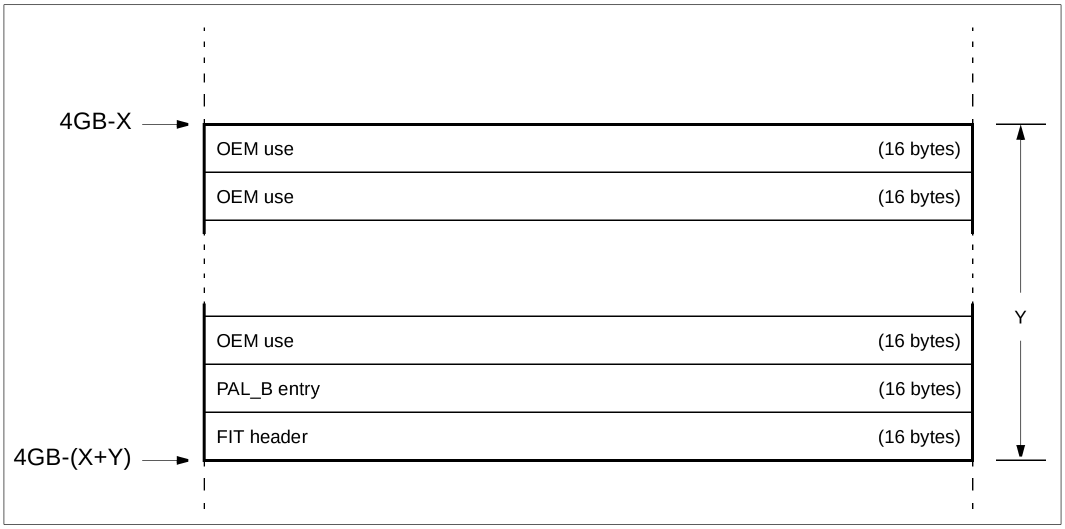
- The 16 bytes at 0xFFFF_FFF0 (4GB-16) contain IA-32 Reset Code.
- The 8 bytes at 0xFFFF_FFE8 (4GB-24) contain the physical address of the SALE_ENTRY entrypoint.
- The 8 bytes at 0xFFFF_FFE0 (4GB-32) contain the physical address of the Firmware Interface Table.

- The 16 bytes at 0xFFFF_FFD0 (4GB-48) contain the FIT entry for the PAL_A (or generic PAL_A in the split PAL_A model) code provided by the processor vendor. The format of this FIT entry is described in Figure 11-7.
- The 8 bytes at 0xFFFF_FFC8 (4GB-56) contains the physical address of the alternate Firmware Interface Table. This pointer is optional and is only needed if the firmware contains an alternate FIT table. If no alternate FIT table it provided a value of 0x0 should be encoded in this entry.
- The 8 bytes at 0xFFFF_FFC0 (4GB-64) are zero-filled and reserved for future use.
- PAL_A code (also known as generic PAL_A code in split PAL_A model) resides below 0xFFFF_FFC0. This area contains the hardware-triggered entrypoints PALE_RESET, PALE_INIT, and PALE_CHECK. In the model where PAL_A is not split, the PAL_A code will perform any processor-specific initialization needed in order for SAL to perform a firmware recovery. In the split PAL_A model, the generic PAL_A will search the FIT table(s) to find the processor-specific PAL_A code. It will then branch to this code to perform the processor-specific initialization needed in order for SAL to perform a firmware recovery. The PAL_A code area is a multiple of 16 bytes in length.
- SAL_A code occupies the region immediately below the PAL_A code. This area contains the SALE_ENTRY entrypoint as well as optional implementation-independent firmware update code. The SAL_A code area is a multiple of 16 bytes in length.
- The collection of regions above from the beginning of the SAL_A code to 4GB is called the Protected Bootblock. The size of the Protected Bootblock is SAL_A size + PAL_A size + 64.
- The Firmware Interface Table (FIT) comprises of 16-byte entries containing starting address and size information for the firmware components. The FIT is generated at build time, based on the size and location of the firmware components. Optionally, an alternate FIT may be included in the firmware. The alternate FIT will only be used if the primary FIT failed its checksum. In the split PAL_A model, this allows the generic PAL_A firmware to find the processor-specific PAL_A component(s), even if the primary FIT is corrupt. This feature allows hand-off to the SAL recovery code, even if there is a primary FIT checksum failure.
- The processor-specific PAL_A contains the code that is required to be run before handing off to SAL for a firmware recovery check. This component is only available on processors that support a split PAL_A firmware model. One processor specific PAL_A is architecturally required in this model. The firmware may optionally contain two or more processor specific PAL_A components.
- The PAL_B block is comprised of code that is not required to be executed for SAL to perform a firmware recovery update. The PAL_B code area is a multiple of 16 bytes in length. The PAL_B block must be aligned on a 32K byte boundary.
- The remainder of the firmware address space is occupied by SAL_B code. SAL_B may include IA-32 BIOS code. The location of the SAL_B and IA-32 BIOS code within the firmware address space is implementation dependent.

The firmware code and data within the firmware address range must be accessible from the processor without any special system fabric initialization sequence. This implies that the system fabric is implicitly initialized at power on for accessing the firmware address space or that the special hardware which contains the firmware code and data is implemented on the processor and not accessed across the system fabric.

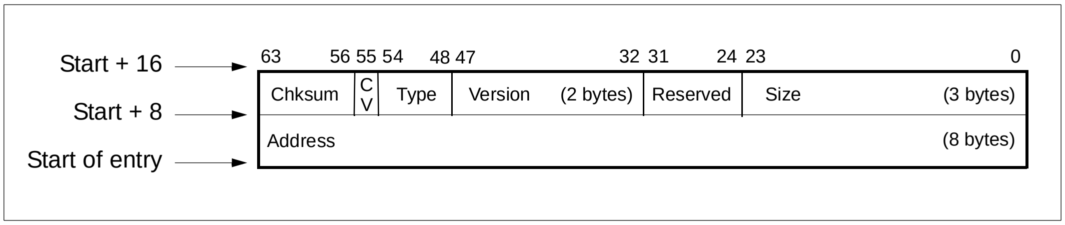
The Firmware Interface Table (FIT) contains starting addresses and sizes for the different firmware components. Because these code blocks may be compiled at different times and places, code in one block (such as PAL_A) cannot branch to code in another block (such as PAL_B) directly. The FIT allows code in one block to find entrypoints in another. Figure 11-6 below shows the FIT layout.

Figure 11-6. Firmware Interface Table



Each FIT entry contains information for the corresponding firmware component. The first entry contains size and checksum information for the FIT itself and the second entry is used for the PAL_B block. OEMs may use additional entries for other firmware components. FIT entries must be arranged in ascending order by the *type* field, otherwise execution of firmware code will be unpredictable. Multiple FIT entries of the same type are allowed.

Figure 11-7. Firmware Interface Table Entry



- *Size* - a 3-byte field containing the size of the component in bytes divided by 16.
- Reserved - All fields listed as reserved must be zero filled.
- *Version* - a 2-byte field containing the component's version number.
- *Type* - A 7-bit field containing the type code for the element. Types are defined in Table 11-1.

Table 11-1. FIT Entry Types

Type	Meaning
0x00	FIT Header
0x01	PAL_B (required)
0x02-0x0D	Reserved
0x0E	Processor Specific PAL_A
0x0F	PAL_A (also generic PAL_A)[a]
0x10-0x7E	OEM-defined
0x7F	Unused Entry

[a] The PAL_A FIT entry is located at 0xFFFF_FFDO (4GB-48) and is not part of the actual FIT table.

OEMs may define unique types for one or more blocks of SAL_B, IA-32 BIOS, etc., within the OEM-defined type range of 0x10 to 0x7E.

- *C_V* - a 1-bit flag indicating whether the component has a valid checksum. If this field is zero, the value in the *Chksum* field is not valid.
- *Chksum* - a 1-byte field containing the component's checksum. The modulo sum of all the bytes in the component and the value in this field (Chksum) must add up to zero. This field is only valid if the *C_V* flag is non-zero. If the checksum option is selected for the FIT, in the FIT Header entry (FIT type 0), the modulo sum of all the bytes in the FIT table must add up to zero.

  Note: The PAL_A FIT entry is not part of the FIT table checksum.
- *Address* - an 8-byte field containing the base address of the component. For the FIT header, this field contains the ASCII value of "_FIT_<sp><sp><sp>" (<sp> represents the space character).

The FIT allows simpler firmware updates. Different components may be updated independently. This address layout can also support firmware images spanning multiple storage devices. FIT entries must be arranged in ascending order by the *type* field, otherwise execution of firmware code will be unpredictable.

11.2 PAL Power On/Reset

11.2.1 PALE_RESET

The purpose of PALE_RESET is to initialize and test the processor. Upon receipt of a power-on reset event the processor begins executing code from the PALE_RESET entrypoint in the firmware address space. PALE_RESET initializes the processor and may perform a minimal processor self test. PAL may optionally perform authentication of the PAL firmware to ensure data integrity. If the authentication code runs cacheable by default, then a processor-specific mechanism will be provided to disable caching for diagnostic purposes.

PALE_RESET then branches to SALE_ENTRY to determine if a recovery condition exists, which would require an update of the firmware. If it does, SALE_ENTRY performs the update and resets the system. If no firmware recovery is needed, SAL returns to PALE_RESET to perform the processor self-tests and initialization. SAL can control the length and coverage of the PAL processor self-test by examining and modifying the self-test control word passed to SAL at the firmware recovery hand-off state. Please see Section 11.2.3 for more information on the self-test control word.

The PAL processor self-tests are split into two phases. The first phase is written to test processor features that do not require external memory to be present to execute correctly. These tests are automatically run when SAL returns to PAL after the branch to SALE_ENTRY for a firmware recovery check. This section is referred to as phase one of processor self-test and they are generally run early during the processor boot process. The second phase is written requiring that external memory is available to execute correctly. These tests are run when a call to the PAL procedure PAL_TEST_PROC is made with the correct parameters set up. These tests are referred to as phase two of processor self-test since they are usually run later in the processor boot process after external memory has been initialized on the platform.

PAL may execute IA-32 instructions to fully test and initialize the processor. This IA-32 code will not generate any special IA-32 bus transactions nor will it require any special platform features to correctly execute. PAL then branches to SALE_ENTRY to conduct platform initialization and testing before loading the operating system software.

11.2.2 PALE_RESET Exit State

- GRs: The contents of all general registers are undefined except the following:
  - GR20 (bank 1) contains the SALE_ENTRY State Parameter as defined in Figure 11-8. For the function field of the SALE_ENTRY State Parameter, only the values 3, RECOVERY CHECK, for the first call to SALE_ENTRY, and 0, RESET, for the second call to SALE_ENTRY are valid.
  - GR32 contains 0 indicating that SALE_ENTRY was entered from PALE_RESET.
  - GR33 contains the geographically significant unique processor ID. The value is the same as that returned by PAL_FIXED_ADDR.
  - GR34 contains the physical address for making a PAL procedure call. If the call is for RECOVERY CHECK, only the subset of PAL procedures needed for SALE_ENTRY to perform firmware recovery will be available. These procedures are:
    - PAL_PLATFORM_ADDR
    - an implementation-specific PAL procedure for PAL authentication.
  - GR35 contains the Self Test State Parameter as defined in Figure 11-9.
  - GR36 contains the PAL_RESET return address for SALE_ENTRY to return to if a recovery condition does not exist. When PAL_RESET calls SALE_ENTRY the second time to initialize the system for operating system use, this register will contain the physical address for making an implementation-specific PAL procedure call for PAL authentication.

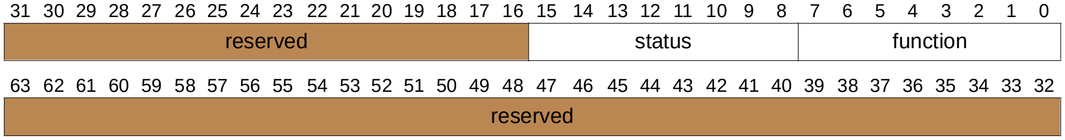
    Note: For all other PAL procedure calls, the physical address at GR34 should be used.
  - GR37 contains the self-test control word as defined in Figure 11-10. This control word is processor implementation-specific and informs SAL if self-test control is implemented and the number of controllable bits. If self-test control is implemented, PAL will read this value when SAL returns to PAL after firmware recovery check. If the self-test control is not supported, this register will be ignored when SAL returns to PAL after firmware recovery check.
  - Banked GRs: All bank 0 general registers are undefined.
- FRs: The contents of all floating-point registers are undefined. The floating-point registers are enabled unless the *state* field of the Self Test State Parameter is FUNCTIONALLY RESTRICTED and the floating-point unit failed self test. Then, the floating-point registers are disabled. Refer to Section 11.2.2.2 for the definition of FUNCTIONALLY RESTRICTED.
- Predicates: The contents of all predicate registers are undefined.
- BRs: The contents of all branch registers are undefined.
- ARs: The contents of all application registers are undefined except the following:
  - RSC: All fields in the register stack configuration register are 0, which places the RSE in enforced lazy mode.
- CFM: The CFM is set up so that all stacked registers are accessible, CFM.sof = 96 and all other CFM fields are 0.

- PSR: PSR.bn is 1; PSR.df1 and PSR.dfh are 1 if the floating-point unit failed self test. All other PSR bits are 0. PSR.ic and PSR.i are zero to ensure external interrupts, NMI and PMI interrupts are disabled.
- CRs: The contents of all control registers are undefined except the following:
  - DCR: contains the value 0.
  - IVA: contains the physical address of an interruption vector table previously set up by PAL. SAL may choose to change this value. The IVA will be 0 when the SALE_ENTRY State Parameter function is RECOVERY CHECK.
- RRs: The contents of all region registers are undefined.
- PKRs: The contents of all protection key registers are undefined.
- DBRs: The contents of all data breakpoint registers are undefined
- IBRs: The contents of all instruction breakpoint registers are undefined.
- PMCs: The contents of all performance monitor control registers are undefined.
- PMDs: The contents of all performance monitor data registers are undefined.
- Cache: The processor internal caches are enabled and invalidated. Unless directed otherwise by the self-test control word, phase one of the processor self-test verifies the caches themselves and the paths from the caches to the processor core. The path from external memory to the caches cannot be tested until phase two of the processor self-test.
- TLB: The TRs and TCs are initialized with all entries having been invalidated. The TLB is disabled because PSR.it=PSR.dt=PSR.rt=0. The TLBs cannot be fully tested until phase two of the processor self-test.

Prior to passing control to SALE_ENTRY, PALE_RESET must ensure that the processor Interrupt block pointer is set to point to address 0x0000_0000_FEE0_0000.

11.2.2.1 Definition of SALE_ENTRY State Parameter

Figure 11-8. SALE_ENTRY State Parameter



- *function* - an 8-bit field indicating the reason for branching to SALE_ENTRY.

Table 11-2. *function* Field Values

Function	Value	Description
RESET	0	System reset or power-on
MACHINE CHECK	1	Machine check event
INIT	2	Initialization event
RECOVERY CHECK	3	Check for recovery condition

All other values of *function* are reserved.

- *status* - a *function*-dependent 8-bit field indicating the firmware status on entry to SALE_ENTRY. If the function value is RESET or RECOVERY_CHECK, the *status* values are:

Table 11-3. *status* Field Values

Status	Value	Description
Normal	0	Normal reset.
FIT Header Failure	1	FIT header is incorrect
FIT Checksum Failure	2	FIT checksum is incorrect
PAL_B Checksum Failure	3	PAL_B checksum is incorrect
PAL_A Authentication Failure	4	PAL_A failed authentication
PAL_B Authentication Failure	5	PAL_B failed authentication
PAL_B Not Found	6	FIT Entry for PAL_B missing
Incompatible	7	PAL_B is incompatible with the processor's stepping
Unaligned	8	PAL_B IVT is not aligned on a 32KB boundary

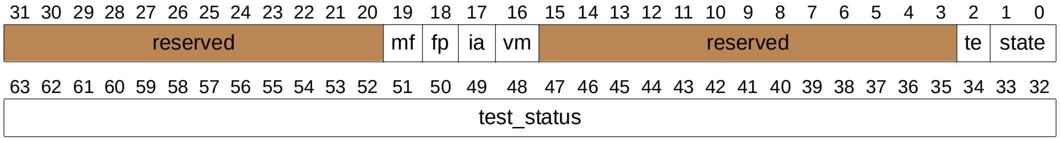
All other values of *status* are reserved.

Definitions of *status* values for other values of *function* are listed in the machine check and init sections.

For the case of RECOVERY CHECK, authentication of PAL_A and PAL_B should be completed before call to SALE_ENTRY.

11.2.2.2 Definition of Self Test State Parameter

Figure 11-9. Self Test State Parameter



- *state* - a 2-bit field indicating the state of the processor after self-test. If SAL directed PAL to skip some self-tests by modifying the self-test control word, failures related to these self-tests will not be reflected in this state.

Table 11-4. *state* Field Values

State	Value	Description
Catastrophic Failure	N/A	The processor is not capable of continuing. In this case it does not branch to SALE_ENTRY.
Healthy	00	No hardware failures have occurred in testing that would affect either the performance or functionality of the processor.
Performance Restricted	01	A hardware failure has occurred in testing that does not affect the functionality of the processor, but performance may be degraded.
Functionally Restricted	10	A hardware failure has occurred in testing that affects the functionality of the processor, but firmware code can still be run. The processor may also be performance restricted.

To further qualify FUNCTIONALLY RESTRICTED, the following requirements will be met:

- The processor has detected and isolated the failing component so that it will not be used.
- The processor must have at least one functioning memory unit, ALU, shifter, and branch unit.
- The floating-point unit may be disabled.

- The RSE is not required to work, but register renaming logic must work properly.
- The paths between the processor controlled caches and the register files have been shown to work. The path between the processor caches and memory cannot be validated until phase two of the processor self-test invoked by the PAL_TEST_PROC procedure.
- Loads and stores to firmware address space must work correctly.

Additional information about the failure can be obtained by examining the *test_status* field of the *Self Test State Parameter.*

For the case of FUNCTIONALLY RESTRICTED, it is required that higher level firmware or OS not use failing functional units during their execution. PAL will not prevent failing functional units from being used.

- *te* - a 1-bit field indicating whether testing has occurred. If this field is zero, the processor has not been tested, and no other fields in the *Self Test State Parameter* are valid. The processor can be tested prior to entering SALE_ENTRY for both RECOVERY CHECK and RESET functions.

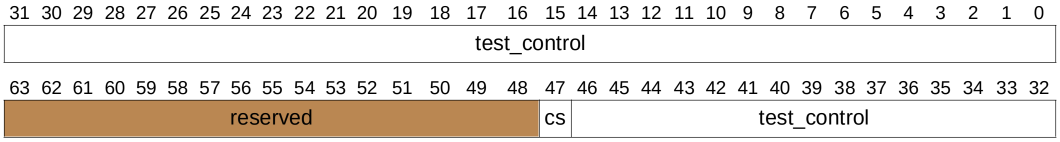
  If the *state* field indicates that the processor is functionally restricted, then the fields *vm, ia* & *fp* specify additional information about the functional failure.

  - *vm* - a 1-bit field, if set to 1, indicating that virtual memory features are not available
  - *ia* - a 1-bit field, if set to 1, indicating that IA-32 execution is not available
  - *fp* - a 1-bit field, if set to 1, indicating that floating-point unit is not available
  - *mf* - a 1-bit field, if set to 1, indicating miscellaneous functional failure other than *vm, ia,* or *fp*. The *test_status* field provides additional information about this failure on an implementation-specific basis.
- *test_status* - an unsigned 32-bit-field providing additional information on test failures when the *state* field returns a value of PERFORMANCE RESTRICTED or FUNCTIONALLY RESTRICTED. The value returned is implementation dependent.

11.2.3 PAL Self-test Control Word

The PAL self-test control word is a 48-bit value. This bit field is defined in Figure 11-10.

Figure 11-10. Self-test Control Word



- *test_control* - This is an ordered implementation-specific control word that allows the user control over the length and run-time of the processor self-tests. This control word is ordered from the longest running tests up to the shortest running tests with bit 0 controlling the longest running test.

  PAL may not implement all 47-bits of the *test_control* word. PAL communicates if a bit provides control by placing a zero in that bit. If a bit provides no control, PAL will place a one in it.

  PAL will have two sets of *test_control* bits for the two phases of the processor self-test.

  PAL provides information about implemented *test_control* bits at the hand-off from PAL to

SAL for the firmware recovery check. These *test_control* bits provide control for phase one of processor self-test. It also provides this information via the PAL procedure call PAL_TEST_INFO for both the phase one and phase two processor tests depending on which information the caller is requesting.

PAL interprets these bits as input parameters on two occasions. The first time is when SAL passes control back to PAL after the firmware recovery check. The second time is when a call to PAL_TEST_PROC is made. When PAL interprets these bits it will only interpret implemented *test_control* bits and will ignore the values located in the unimplemented *test_control* bits.

PAL interprets the implemented bits such that if a bit contains a zero, this indicates to run the test. If a bit contains a one, this indicates to PAL to skip the test.

If the *cs* bit indicates that control is not available, the *test_control* bits will be ignored or generate an illegal argument in procedure calls if the caller sets these bits.

- *cs* - Control Support: This bit defines if an implementation supports control of the PAL self-tests via the self-test control word. If this bit is 0, the implementation does not support control of the processor self-tests via the self-test control word. If this bit is 1, the implementation does support control of the processor self-tests via the self-test control word.

  If control is not supported, GR37 will be ignored at the hand-off between SAL and PAL after the firmware recovery check and the PAL procedures related to the processor self-tests may return illegal arguments if a user tries to use the self-test control features.

11.3 Machine Checks

11.3.1 PALE_CHECK

When a machine check abort (MCA) occurs, PALE_CHECK is responsible for saving minimal processor state to a uncacheable platform-specific memory location previously registered with PAL via the PAL_MC_REGISTER_MEM procedure. This platform location is called the Minimal State Save Area (min-state save area) and is described in Section 11.3.2.3. PALE_CHECK is also responsible for correcting processor related errors whenever possible. PALE_CHECK terminates by branching to SALE_ENTRY, passing the state of the processor at the time of the error. The level of recovery provided by PALE_CHECK is implementation dependent and is beyond the scope of this specification.

At the hand-off from PALE_CHECK to SALE_ENTRY, error information is passed in the Processor State Parameter described in Section 11.3.2.1. After exit from PALE_CHECK, more detailed error information is available by calling the PAL_MC_ERROR_INFO procedure. Information about implementation-dependent state is available by calling the PAL_MC_DYNAMIC_STATE procedure. The interrupted process may be resumed by calling the PAL_MC_RESUME procedure. See Section 11.3.3 for more information on returning to the interrupted context and Section 11.9 "PAL Procedures" on page 2:285 for detailed descriptions of all these procedure calls.

Code for handling machine checks must take into consideration the possibility that nested machine checks may occur. A nested machine check is a machine check that occurs while a previous machine check is being handled.

PALE_CHECK is entered in the following conditions:

- When PSR.mc = 0 and an error occurs which results in a machine check, or
- When PSR.mc changes from 1 to 0 and there is a pending machine check from an earlier error.

PSR.mc is set to 1 by the hardware when PALE_CHECK is entered. PSR.mc will remain set for the duration of PALE_CHECK, and PALE_CHECK will exit with PSR.mc set. SAL must not clear PSR.mc to 0 before all the information from the current machine check is logged. If SAL enables machine checks (by setting PSR.mc=0) during the SAL MCA handling, there is a potential for the error logs in the processor and the min-state save area to be overwritten by a subsequent MCA event. PALE_CHECK must attempt to branch to SALE_ENTRY unless code execution is not possible.

The error information logged will reflect the state at the time the error occurred. State information from a different point in time will NOT be logged. If complete information is not available a code is logged which indicates that the information is not available.

- The processor state information used to resume a process for which an error has been corrected will reflect the state at the time the machine check interruption occurred and will be sufficient to resume the interrupted process.
- When a single error is signalled multiple times (for example, multiple operations to a single bad cache line), hardware and firmware will be able to perform the same logging and recovery as if the error had been signalled once.

For testing and configuration purposes, it may be necessary for software to intentionally generate a machine check. In this case PALE_CHECK will log the error information, but not attempt recovery before branching to SALE_ENTRY. To allow for this, the PAL_MC_EXPECTED procedure call is defined to indicate that PALE_CHECK should not to attempt recovery.

11.3.1.1 Resources Required for Machine Check and Initialization Event Recovery

While the level of recovery from machine checks is implementation dependent, for each particular level of recovery there is a set of architecturally required resources. The following paragraphs define the required and optional resources needed to support firmware and software recovery of machine checks and initialization events.

- Minimal resources required to allow software recovery of machines checks when PSR.ic=1:
  - XR0 register: memory pointer to min-state save area previously registered with PAL via the PAL_MC_REGISTER_MEM procedure. The layout of this memory area is described in Section 11.3.2.3.
  - Bank zero registers GR 24 through GR 31. These registers are not preserved across interruptions and may be used as scratch registers by machine check recovery code. See Section 3.3.7 "Banked General Registers" for the definition of the bank 0 registers.
- Additional resources required to allow software recovery of machine checks when PSR.ic=0. The presence of these resources is processor implementation specific. The PAL_PROC_GET_FEATURES procedure described on page 2:351 returns information on the existence of these optional resources.
  - XIP, XPSR, XFS: interruption resources implemented to store information about the IP, PSR and IFS when the machine check occurred. A model-specific version of the `rfi` instruction must also be implemented to restore the machine context from these resources.

- XR1-XR3: scratch registers implemented to preserve bank 0 GR 24 through GR 31.

Each of the registers described above should be accessed only by PAL in order to support firmware and software recovery of machine checks.

11.3.2 PALE_CHECK Exit State

The state of the processor on exiting PALE_CHECK is:

- GRs: The contents of all non-banked static registers (GR1-GR15), bank zero static registers and bank one static registers (GR16-31) at the time of the MCA have been saved in the min-state save area and are available for use.
    - If recovery is not supported when PSR.ic=0 then GR24 - GR31 (bank 0) are undefined and their contents have been lost. In this case, recovery is not possible. See Section 11.3.1.1 for details.
    - GR16 through GR20 (bank 0) contain parameters which PALE_CHECK passes to SALE_ENTRY for diagnostic and recovery purposes:
        - GR16 contains the address to the first available location in the min-state save area for use by SAL. The address is 8-byte aligned.
        - GR17 contains the value of the min-state save area address stored in XR0.
        - GR18 contains the Processor State Parameter, as defined in Figure 11-11.
        - GR19 contains the PALE_CHECK return address for rendezvous, or 0 if no return is expected. (See Section 11.3.2.2)
        - GR20 contains the SALE_ENTRY State Parameter as defined in Figure 11-14.
- FRs: The contents of all floating-point registers are unchanged from the time of the MCA.
- Predicates: All predicate registers have been saved in the min-state save area and are available for use.
- BRs: The contents of all branch registers are unchanged from the time of the MCA, except the following.
    - BR0 has been saved to the min-state save area and is available for use.
- ARs: The contents of all application registers are unchanged from the time of the MCA, except the RSE control register (RSC) and the RSE backing store pointer (BSP). The RSC register is unchanged, except that the RSC.mode field will be set to 0 (enforced lazy mode) and the RSC register at the time of the MCA has been saved in the min-state save area. A `cover` instruction is executed in the PALE_CHECK handler which allocates a new stack frame of zero size. BSP will be modified to point to a new location, since all the registers from the current frame at the time of interruption were added to the RSE dirty partition by the allocation of a new stack frame.
- CFM: The CFM register points to a zero-size current frame and all the rotating register bases are set to zero. The CFM register at the time of the MCA has been saved to the min-state save area in either the IFS or XFS slot depending on the implementation.
- RSE: Is in enforced lazy mode, and stacked registers are unchanged from the time of the MCA.
- PSR: PSR.mc is 1; all other bits are 0. The PSR at the time of the MCA is saved in the min-state save area.
- CRs: The contents of all control registers are unchanged from the time of the MCA with the exception of interruption resources, which are described below.
- RRs: The contents of all region registers are unchanged from the time of the MCA.

- PKRs: The contents of all protection key registers are unchanged from the time of the MCA.
- DBR/IBRs: The contents of all breakpoint registers are unchanged from the time of the MCA.
- Cache: The processor internal cache is enabled and is unchanged from the time of the MCA except for any lines that were invalidated to correct the error.
- TLB: The TCs may be initialized and the TRs are unchanged from the time of the MCA.
- Interruption Resources:
  - IRR: PALE_CHECK may not change the IRR, but interrupts may have arrived asynchronously, changing the contents of the IRRs.
  - The contents of IIP, IPSR and IFS at the time of the MCA are saved to the min-state save area and are available for use.

11.3.2.1 Processor State Parameter (GR 18)

Figure 11-11. Processor State Parameter

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
gr	b0	b1	fp	pr	br	ar	rr	tr	dr	pc	cr	ex	cm	rs	in	dy	pm	pi	mi	tl	hd	us	ci	co	sy	mn	me	ra	rz	rsvd	

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
uc	rc	bc	tc	cc	reserved											dsize															

The term “valid” in Table 11-5 indicates that the registers are either unchanged from the time of interruption or that the values have been preserved in the min-state save area.

Table 11-5. Processor State Parameter Fields

Field name	Bit	Description
rsvd	0-1	Reserved
rz	2	The attempted processor rendezvous was successful if set to 1.
ra	3	A processor rendezvous was attempted if set to 1.
me	4	Distinct multiple errors have occurred, not multiple occurrences of a single error. Software recovery may be possible if error information has not been lost.
mn	5	Min-state save area has been registered with PAL if set to 1.
sy	6	Storage integrity synchronized. A value of 1 indicates that all loads and stores prior to the instruction on which the machine check occurred completed successfully, and that no loads or stores beyond that point occurred. See Table 11-6.
co	7	Continuable. A value of 1 indicates that all in-flight operations from the processor where the machine check occurred were either completed successfully (such as a load), were tagged with an error indication (such as a poisoned store), or were suppressed and will be re-issued if the current instruction stream is restarted. This bit can only be set if the architectural state saved on a machine check is all valid. If this bit is set, then *us* must be cleared to 0, and *ci* must be set to 1. See Table 11-6.
ci	8	Machine check is isolated. A value of 1 indicates that the error has been isolated by the system, it may or may not be recoverable. If 0, the hardware was unable to isolate the error within the CPU and memory hierarchy. The error may have propagated off the system (to persistent storage or the network). If *ci* = 0 then *us* will be set to 1, and *co* and *sy* are cleared to 0. See Table 11-6.
us	9	Uncontained storage damage. A value of 1 indicates the error is contained within the CPU and memory hierarchy, but that some memory locations may be corrupt. If *us* is set to 1, then *co* and *sy* will always be cleared to 0. See Table 11-6.
hd	10	Hardware damage. A value of 1 indicates that as a result of the machine check some non essential hardware is no longer available causing this processor to execute with degraded performance (no functionality has been lost).

Table 11-5. Processor State Parameter Fields (Continued)

Field name	Bit	Description
tl	11	Trap lost. A value of 1 indicates the machine check occurred after an instruction was executed but before a trap that resulted from the instruction execution could be generated.
mi	12	More information. A value of 1 indicates that more error information about the machine check event is available by making the PAL_MC_ERROR_INFO procedure call.
pi	13	Precise instruction pointer. A value of 1 indicates that the machine logged the instruction pointer to the bundle responsible for generating the machine check.
pm	14	Precise min-state save area. A value of 1 indicates that the min-state save area contains the state of the machine for the instruction responsible for generating the machine check. When this bit is set, the *pi* bit will always be set as well.
dy	15	Processor Dynamic State is valid. (1=valid, 0=not valid) See the PAL_MC_DYNAMIC_STATE procedure call for more information.
in	16	Interruption caused by INIT. (0=machine check, 1=INIT)
rs	17	The RSE is valid. (1=valid, 0=not valid)
cm	18	The machine check has been corrected. (1=corrected, 0=not corrected)
ex	19	A machine check was expected. (1=expected, 0=not expected)
cr	20	Control registers are valid. (1=valid, 0=not valid)
pc	21	Performance counters are valid. (1=valid, 0=not valid)
dr	22	Debug registers are valid. (1=valid, 0=not valid)
tr	23	Translation registers are valid. (1=valid, 0=not valid)
rr	24	Region registers are valid. (1=valid, 0=not valid)
ar	25	Application registers are valid. (1=valid, 0=not valid)
br	26	Branch registers are valid. (1=valid, 0=not valid)
pr	27	Predicate registers are valid. (1=valid, 0=not valid)
fp	28	Floating-point registers are valid. (1=valid, 0=not valid)
b1	29	Preserved bank one general registers are valid. (1=valid, 0=not valid)
b0	30	Preserved bank zero general registers are valid. (1=valid, 0=not valid)
gr	31	General registers are valid. (1=valid, 0=not valid) (does not include banked registers)
dsize	47:32	Size in bytes of Processor Dynamic State returned by PAL_MC_DYNAMIC_STATE.
reserved	58:48	Reserved
cc	59	Cache check. A value of 1 indicates that a cache related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
tc	60	TLB check. A value of 1 indicates that a TLB related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
bc	61	Bus check. A value of 1 indicates that a bus related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
rc	62	Register file check. A value of 1 indicates that a register file related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
uc	63	Uarch check. A value of 1 indicates that a micro-architectural related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.

11.3.2.1.1 Using Processor State Parameter to Determine if Software Recovery of a Machine Check is Possible

The *us, ci, co, and sy* bits in the Processor State Parameter are valid only if the error has not been previously corrected in hardware or firmware (*cm* bit is 0). Even then, only the bit combinations shown in Table 11-6 are valid. If the multiple error bit is set (*me*=1) both the *co* and *sy* bits must be 0. The *us* and *ci* bits will be set according to the worst case of the errors that occurred.

Table 11-6. Software Recovery Bits in Processor State Parameter

cm	us	ci	co	sy	Description
1	x	x	x	x	The machine check is corrected. The *us, ci, co,* and *sy* bits are not valid.
0	1	0	0	0	The error was not isolated. Software must reset system. Data on disk may be corrupt.
0	1	1	0	0	The error was isolated but not contained. Corrupt data was not written to I/O, but may remain in the CPU or memory untagged. Software must reset system.
0	0	1	0	0	The error was isolated and contained, but is not continuable. The current instruction stream cannot be restarted without loss of information. Partial recovery may be possible.
0	0	1	1	0	The error was isolated, contained, and is continuable. If software can correct the error the current instruction stream can be continued with no loss of information.
0	0	1	1	1	The error was isolated, contained, and is continuable. The instruction pointer points to the instruction where the error occurred. If software can correct the error the current instruction stream can be continued with no loss of information.

11.3.2.2 Multiprocessor Rendezvous Requirements for Handling Machine Checks

When PALE_CHECK has determined that an error has occurred which could cause a multiprocessor system to lose error containment, it must rendezvous the other processors in the system before proceeding with further processing of the machine check. This is accomplished by branching to SALE_ENTRY with a non-zero return vector address in GR19. It is then the responsibility of SAL to rendezvous the other processors and return to PALE_CHECK through the address in GR19. If the rendezvous was successful GR19 must be set to 0 before return.

At the time PALE_CHECK makes the rendezvous call to SALE_ENTRY, the processor state is exactly the same as defined in Section 11.3.2 "PALE_CHECK Exit State" with the following requirement on the use of registers by SAL:

Any processor state not listed below must be either unchanged or restored by SAL before returning to PALE_CHECK.

- SAL will preserve the values in GR4-GR7 and GR17-GR18.
- SAL will return to PALE_CHECK via the address in GR19.
- SAL will set up GR19 to indicate the success of the rendezvous before returning to PAL.
  - GR19 is zero to indicate the rendezvous was successful.
  - GR19 is non zero to indicate that the rendezvous was unsuccessful.
- All other non-banked (GR1-3, GR8-15), bank 0 GRs (GR20-GR31) and BR0 are undefined and available for use by SAL.

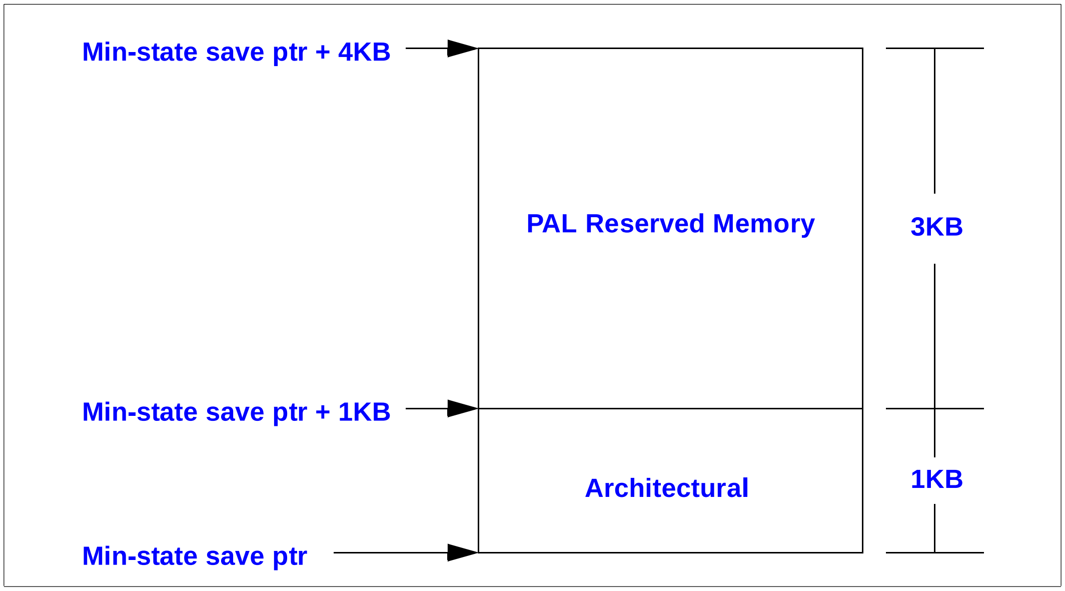
After return from the SAL rendezvous call, PALE_CHECK will complete processing the machine check if the rendezvous was successful and then branch to SALE_ENTRY with GR19 set to zero. The processor state when transferring to SAL is as defined in Section 11.3.2 "PALE_CHECK Exit State". If the rendezvous failed PALE_CHECK will simply construct the Processor State Parameter and branch to SALE_ENTRY.

Any further discussion of multiprocessor rendezvous, including platform requirements and implications, is beyond the scope of this specification. See the relevant SAL/Error handling documents for further information.

11.3.2.3 Processor Min-state Save Area Layout

The processor min-state save area is 4KB in size and must be in an uncacheable region. The first 1KB of this area is architectural state needed by the PAL code to resume during MCA and INIT events (architected min-state save area + reserved). The remaining 3KB is a scratch buffer reserved exclusively for PAL use, therefore SAL and OS must not use this area. The layout of the processor min-state save area is shown in Figure 11-12.

Figure 11-12. Processor Min-state Save Area Layout



The layout for the processors portion of the architectural 1KB processor min-state save area is shown in Figure 11-13. When SAL registers the area with PAL, it passes in a pointer to offset zero of the area. When PALE_CHECK is entered as a result of a machine check, it fills in processor state, processes the machine check, and branches to SALE_ENTRY with a pointer to the first available memory location that SAL can use in GR16. SAL may allocate a variable sized area above the address passed in GR16 up to the 1KB architectural limit, but this is internal to SAL and not known to PAL.

The base address of the min-state save area must be aligned on a 512-byte boundary. All saves and restores to and from the min-state save area are made using 8-byte wide load and store instructions. If the processor min-state save area is not registered via the PAL_MC_REGISTER_MEM procedure prior to the machine check, software recovery is not possible.

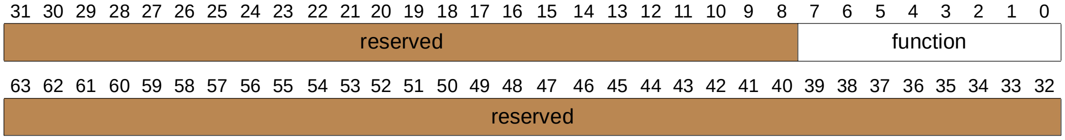
Figure 11-13. Processor State Saved in Min-state Save Area

0xf8	Bank 0 GR31			
0xf0	Bank 0 GR30			
0xe8	Bank 0 GR29			
0xe0	Bank 0 GR28			
0xd8	Bank 0 GR27			GR16
0xd0	Bank 0 GR26		≈	
0xc8	Bank 0 GR25			
0xc0	Bank 0 GR24	0x1c0	XFS or undefined	
0xb8	Bank 0 GR23	0x1b8	XPSR or undefined	
0xb0	Bank 0 GR22	0x1b0	XIP or undefined	
0xa8	Bank 0 GR21	0x1a8	IFS	
0xa0	Bank 0 GR20	0x1a0	IPSR	
0x98	Bank 0 GR19	0x198	IIP	
0x90	Bank 0 GR18	0x190	RSC	
0x88	Bank 0 GR17	0x188	BR0	
0x80	Bank 0 GR16	0x180	Predicate Registers	
0x78	GR15	0x178	Bank 1 GR31	
0x70	GR14	0x170	Bank 1 GR30	
0x68	GR13	0x168	Bank 1 GR29	
0x60	GR12	0x160	Bank 1 GR28	
0x58	GR11	0x158	Bank 1 GR27	
0x50	GR10	0x150	Bank 1 GR26	
0x48	GR9	0x148	Bank 1 GR25	
0x40	GR8	0x140	Bank 1 GR24	
0x38	GR7	0x138	Bank 1 GR23	
0x30	GR6	0x130	Bank 1 GR22	
0x28	GR5	0x128	Bank 1 GR21	
0x20	GR4	0x120	Bank 1 GR20	
0x18	GR3	0x118	Bank 1 GR19	
0x10	GR2	0x110	Bank 1 GR18	
0x8	GR1	0x108	Bank 1 GR17	
0x0	NaT bits for saved GRs	0x100	Bank 1 GR16	

The value passed in GR16 to SAL may point beyond the defined processor state shown in Figure 11-13. PAL may use this area for implementation-dependent processor state that needs to be saved and restored.

11.3.2.4 Definition of SALE_ENTRY State Parameter

Figure 11-14. SALE_ENTRY State Parameter



- *function* - an 8-bit field indicating the reason for branching to SALE_ENTRY.

Table 11-7. *function* Field Values

Function	Value	Description
RESET	0	System reset or power-on
MACHINE CHECK	1	Machine check event
INIT	2	Initialization event
RECOVERY CHECK	3	Check for recovery condition in SAL

All other values of *function* are reserved.

11.3.3 Returning to the Interrupted Process

The PAL_MC_RESUME procedure is defined to return to the interrupted context after handling a machine check or initialization event. See page 2:349 for a description of the PAL_MC_RESUME procedure. If software attempts to return to the interrupted context without using this procedure, processor behavior is undefined.

There are certain error cases that may require returning to a new context in order to handle the machine check. If this occurs a new context can be returned to via the PAL_MC_RESUME procedure with the *new_context* flag set. The caller needs to set up the new processor min-state save area as shown in Figure 11-13 for all the listed register states. The IIP, IPSR, IFS and the XIP, XPSR, and XFS should both contain the new instruction pointer, PSR value, and CFM values. The IPSR and XPSR must have the PSR.ic bit set to one, since return to an interruption handler is not supported.

When returning to a new context, the memory area from XFS up to the 1KB architectural limit is ignored by the PAL_MC_RESUME procedure. When a new context is returned to, the state originally saved in the min-state save area (old context) shall be discarded and never used again.

11.4 PAL Initialization Events

11.4.1 PALE_INIT

PALE_INIT is entered when an initialization event (INIT) occurs, as a result of the assertion on an INIT signal to the processor or an INIT interruption occurring. If PSR.mc = 1, the initialization event is held pending until PSR.mc becomes 0. The purpose of PALE_INIT is to save the architecturally defined processor state to the Minimal State Save Area (min-state save area) and to branch to SALE_ENTRY. The code sequence interrupted by the initialization event can be restarted via PAL_MC_RESUME if PSR.ic = 1. The code sequence interrupted by the initialization event can be restarted if PSR.ic = 0 and the processor has implemented the optional recovery resources described in Section 11.3.1.1 "Resources Required for Machine Check and Initialization Event Recovery". If PSR.ic = 0 and the optional recovery resources have not been implemented, then the initialization event is not recoverable.

11.4.2 PALE_INIT Exit State

The state of the processor on exiting PALE_INIT is:

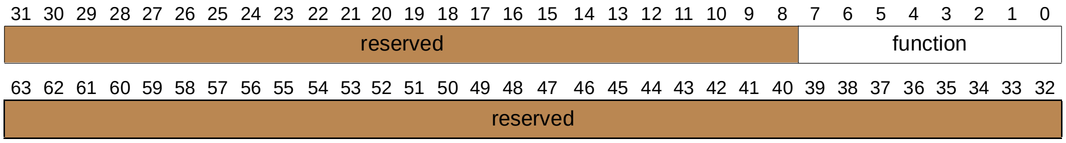
- GRs: The contents of all non-banked static registers (GR1-GR15), bank zero static registers and bank one static registers (GR16-31) at the time of the INIT have been saved in the min-state save area and are available for use.
  - If recovery is not supported when PSR.ic=0 then GR24 - GR31 (bank 0) are undefined and their contents have been lost. In this case, recovery is not possible. See Section 11.3.1.1 for details.
  - GR16 through GR20 (bank 0) contain parameters which PALE_INIT passes to SALE_ENTRY for diagnostic and recovery purposes:
    - GR16 contains the address to the first available location in the min-state save area for use by SAL. The address is 8-byte aligned.
    - GR17 contains the value of the min-state save area address stored in XR0.
    - GR18 contains the Processor State Parameter, as defined in Figure 11-11 on page 2:271, with the exception of the following fields. *cm, op, tl, hd, us, sy, me, uc, bc, tc*, and *cc* are all 0. *ci*, *in* and *co* are 1. (See Table 11-5 on page 2:271)
    - GR19 contains the PALE_INIT return address for rendezvous, or 0 if no return is expected. (See Section 11.3.2.2)
    - GR20 contains the SALE_ENTRY state as defined in Figure 11-14.
- FRs: The contents of all floating-point registers are unchanged from the time of the INIT.
- Predicates: All predicate registers have been saved in the min-state save area and are available for use.
- BRs: The contents of all branch registers are unchanged from the time of the INIT except the following:
  - BR0 is has been saved to the min-state save area and is available for use.
- ARs: The contents of all application registers are unchanged from the time of the INIT, except the RSE control register (RSC) and the RSE backing store pointer (BSP). The RSC register is unchanged, except that the RSC.mode field will be set to 0 (enforced lazy mode) and the RSC register at the time of the INIT has been saved in the min-state save area. A `cover` instruction

is executed in the PALE_INIT handler which allocates a new stack frame of zero size. BSP will be modified to point to a new location, since all the registers from the current frame at the time of interruption were added to the RSE dirty partition by the allocation of a new stack frame.

- CFM: The CFM register points to a zero-size current frame and all the rotating register bases are set to zero. The CFM register at the time of the INIT has been saved to the min-state save area in either the IFS or XFS slot depending on the implementation.
- RSE: The RSE is in enforced lazy mode, and all stacked registers are unchanged from the time of the INIT.
- PSR: PSR.mc is 1; all other bits are 0. The PSR at the time of the INIT is saved in the min-state save area.
- CRs: The contents of all control registers are unchanged from the time of the INIT with the exception of the interruption resources, which are described below.
- RRs: The contents of all region registers are unchanged from the time of the INIT.
- PKRs: The contents of all protection key registers are unchanged from the time of the INIT.
- DBR/IBRs: The contents of all breakpoint registers are unchanged from the time of the INIT.
- Cache: The contents of the caches are unchanged from the time of the INIT.
- TLB: The TCs may be initialized and the TRs are unchanged from the time of the INIT.
- Interruption Resources:
  - IRR: PALE_INIT may not change the IRR, but interrupts may have arrived asynchronously, changing the contents of the IRRs.
  - The contents of IIP, IPSR and IFS at the time of INIT are saved to the min-state save area and are available for use.

11.4.2.1 Definition of SALE_ENTRY State Parameter

Figure 11-15. SALE_ENTRY State Parameter



- *function* - an 8-bit field indicating the reason for branching to SALE_ENTRY.

Table 11-8. *function* Field Values

Function	Value	Description
RESET	0	System reset or power-on
MACHINE CHECK	1	Machine check event
INIT	2	Initialization event
RECOVERY CHECK	3	Check for recovery condition in SAL

All other values of *function* are reserved.

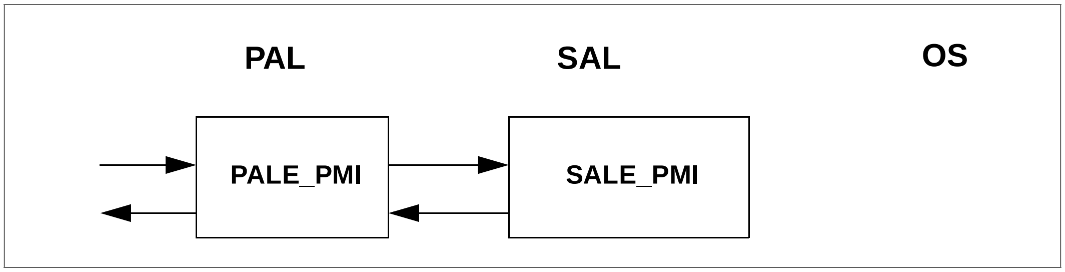
11.5 Platform Management Interrupt (PMI)

11.5.1 PMI Overview

PMI is an asynchronous highest-priority external interrupt that encapsulates a collection of platform-specific interrupts. Platform Management Interrupts occur during instruction processing, causing the flow of control to be passed to the PAL PMI handler. In the process, certain processor state is saved away automatically by the processor hardware and the processor starts executing instructions at the PALE_PMI entrypoint which then transitions to SAL PMI code. Upon completion of processing, the SAL PMI code returns to PAL PMI code to restore the interrupted processor state and to resume execution at the interrupted instruction.

As shown in Figure 11-16, PMI code consists of two major components, namely the PAL PMI handler which handles all processor-specific processing, and the SAL PMI handler which handles all platform-related processing.

Figure 11-16. PMI Entrypoints



The hardware events that can cause the PMI request are referred to as PMI events. PMI events are the highest priority external interrupts and are only maskable when the system software is processing very critical tasks with PSR.ic=0. When PSR.ic is 1, PMI events are unmasked. PSR.i has no effect on PMI events. All PMI events are internally latched into an array of implementation-specific latches in the processor. The PAL PMI handler reads the latches to determine what PMI vector requests are pending and dispatches them in priority order. Table 11-9 lists the PMI events and their priority.

Table 11-9. PMI Events and Priorities

PMI Events	Priority
PMI message for PAL (vectors 4-15)	High
PMI message for SAL (vectors 1-3)	
PMI pin (vector 0)	Low

PMI messages can be delivered by an external interrupt controller, or as an inter-processor interrupt using delivery mode 010. Table 11-10 shows the PMI message vector assignments. Vectors 4-15 are reserved for PAL, and within these PAL vectors, a higher vector number has higher priority. Vectors 1-3 are available for SAL to use, and within these SAL vectors, a higher vector number has higher priority. Vector 0 is used to indicate the PMI pin event. The PMI vector number is passed to the SAL PMI handler in GR 24. Vectors described as Intel reserved will be ignored by the processor.

Table 11-10. PMI Message Vector Assignments

Priority		Vector	Description
Low ↓ High	SAL Vectors	0	PMI pin
		1	Available for SAL firmware
		2	
		3	
Low ↓ High	Intel Reserved PAL Vectors	4	Intel Reserved
		5	
		6	
		7	
		8	
		9	
		10	
		11	
		12	
		13	IA-32 Machine Check Rendezvous
		14	Intel Reserved
		15	

11.5.2 PALE_PMI Exit State

The state of the processor on exiting PALE_PMI is:

- GRs: The contents of non-banked general registers are unchanged from the time of the interruption.
  - Bank 1 GRs: The contents of all bank one general registers are unchanged from the time of the interruption.
  - Bank 0:GR16-23: The contents of these bank zero general registers are unchanged from the time of the interruption.
  - Bank 0:GR24-31: contain parameters which PALE_PMI passes to SALE_PMI:
    - GR24 contains the value decoded as follows
      - Bits 7-0: PMI Vector Number
      - Bit 63-8: Reserved
    - GR25 contains the value of the min-state save area address stored in XR0.
    - GR26 contains the value of saved RSC. The contents of this register shall be preserved by SAL PMI handler.
    - GR27 contains the value of saved B0. The contents of this register shall be preserved by SAL PMI handler.
    - GR28 contains the value of saved B1. The contents of this register shall be preserved by SAL PMI handler.
    - GR29 contains the value of the saved predicate registers. The contents of this register shall be preserved by SAL PMI handler
    - GR30-31 are scratch registers available for use.

- FRs: The contents of all floating-point registers are unchanged from the time of the interruption.
- Predicates: The contents of all predicate registers are undefined and available for use.
- BRs: The contents of all branch registers are unchanged, except the following which contain the defined state.
    - BR1 is undefined and available for use.
    - BR0 PAL PMI return address.
- ARs: The contents of all application registers are unchanged from the time of the interruption, except the RSE control register (RSC). The RSC.mode field will be set to 0 (enforced lazy mode) while the other fields in the RSC are unchanged.
- CFM: The contents of the CFM register is unchanged from the time of the interruption.
- RSE: Is in enforced lazy mode, and stacked registers are unchanged from the time of the interruption.
- PSR: All PSR bits are equal to 0.
- CRs: The contents of all control registers are unchanged from the time of the interruption with the exception of interruption resources, which are described below.
- RRs: The contents of all region registers are unchanged from the time of the interruption.
- PKRs: The contents of all protection key registers are unchanged from the time of the interruption.
- DBR/IBRs: The contents of all breakpoint registers are unchanged from the time of the interruption.
- Cache: The processor internal cache is not specifically modified by the PMI handler but may be modified due to normal cache activity of running the handler code.
- TLB: The TCs are not modified by the PALE_PMI handler and the TRs are unchanged from the time of the interruption.
- Interruption Resources:
    - IRRs: The contents of IRRs are unchanged from the time of the interruption.
    - IIP and IPSR contain the value of IP and PSR. The IFS.v bit is reset to 0.

11.5.3 Resume from the PMI Handler

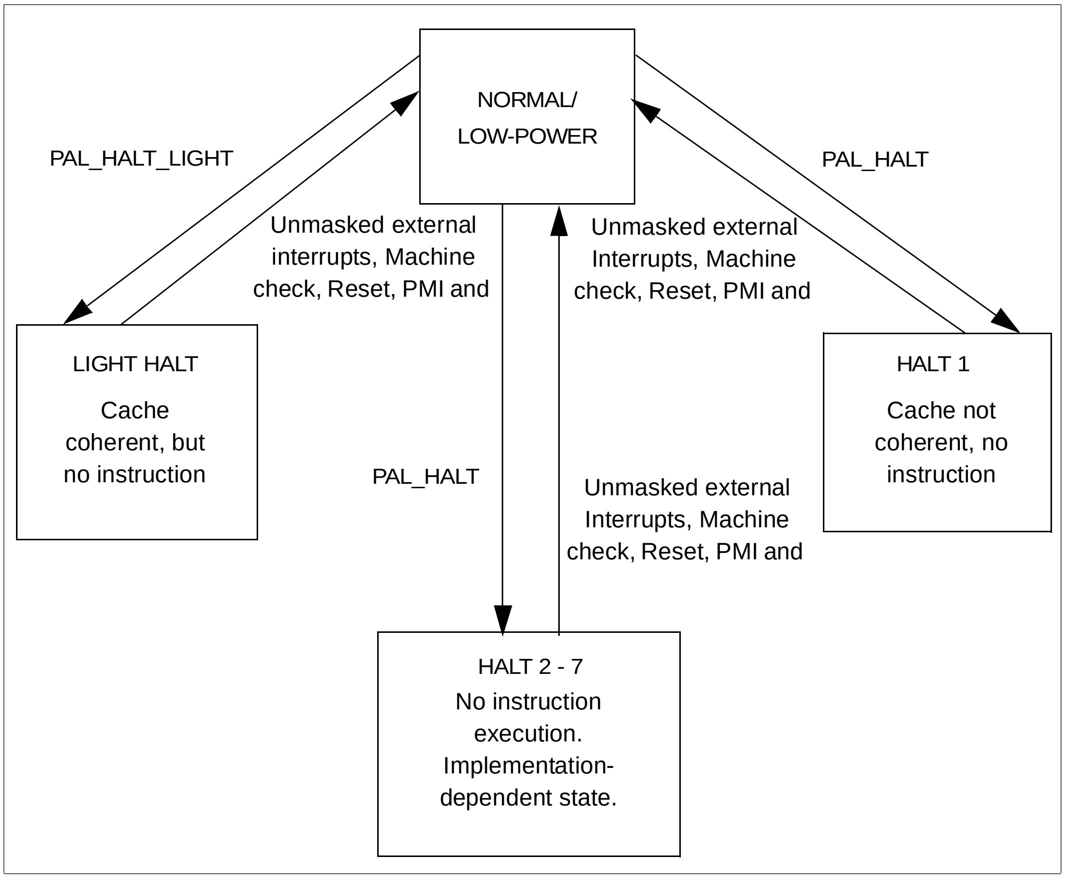
To return to the instruction that was interrupted by the PMI event, SAL PMI must branch to the PAL PMI target address in BR0. All register contents must be preserved as specified in Section 11.5.2.

11.6 Power Management

This section describes the architecturally supported set of required and optional power states that may be implemented to reduce power consumption in implementations where this is a design goal. In addition, the PAL interfaces required to manage these states are described.

Figure 11-17 shows state transitions for the various power states and the software interfaces required for the transitions.

Figure 11-17. Power States


NORMAL/
LOW-POWER
PAL_HALT_LIGHT
PAL_HALT
Unmasked external interrupts, Machine check, Reset, PMI and
Unmasked external Interrupts, Machine check, Reset, PMI and
LIGHT HALT
Cache coherent, but no instruction
HALT 1
Cache not coherent, no instruction
PAL_HALT
Unmasked external Interrupts, Machine check, Reset, PMI and
HALT 2 - 7
No instruction execution. Implementation-dependent state.

- NORMAL - the normal, fully functional, highest power state.
- LOW-POWER - An implementation may choose to dynamically reduce power via microarchitectural low power techniques. The operation of interrupts, snoops, etc., in low-power mode will be identical to those in normal-power mode. This dynamic power reduction is optional for an implementation to support. The PAL procedures PAL_PROC_GET_FEATURES and PAL_PROC_SET_FEATURES returns whether an implementation supports dynamic power reduction. If an implementation supports dynamic power reduction then this procedure will allow the caller to enable or disable this feature.

The following software controllable low power states may be provided. They are described below.

- LIGHT_HALT - entered by calling PAL_HALT_LIGHT. This state reduces power by stopping instruction execution, but maintains cache and TLB coherence in response to external requests. The processor transitions from this state to the NORMAL state in response to any unmasked external interrupt (including NMI), machine check, reset, PMI or INIT. An unmasked external interrupt is defined to be an interrupt that is permitted to interrupt the processor based on the current setting of the TPR.mic and TPR.mmi fields. This state is a required state.
- HALT 1 - entered by calling PAL_HALT with a power state argument equal to one. This implementation-dependent low-power state will maintain the processor caches but will ignore any coherency bus traffic. This state is optional for a processor to implement. It is the responsibility of the caller to ensure cache coherency in this state.

- HALT 2 - 7 - these are optional implementation-dependent states entered by calling PAL_HALT with a power state argument in the range of 2-7. Before making this procedure call, the operating system software should first ascertain that the states are implemented by calling PAL_HALT_INFO. The information returned from the PAL_HALT_INFO procedure will also specify the coherency of caches and TLBs for each of these low-power states.

The interval timer within the processor will function at a constant frequency in all the power states as long as the input clock to the processor is maintained.

The PAL procedure PAL_HALT_INFO returns information about the power states implemented in a particular processor. This information allows the caller to decide which low power states are implemented and which ones to call based on the callers requirements.

11.7 PAL Glossary

Corrected Machine Check (CMC)

A corrected machine check is a machine check that as been successfully corrected by hardware and/or firmware. Information about the cause of the error is recorded, and an interrupt is set to allow the Operating System software to examine and diagnose the error. Return is controlled to the program executing at the time of the error.

Entrypoint

A firmware entrypoint is a piece of code which is triggered by a hardware event, usually the assertion of a processor pin, or the receipt of an interruption. If return to the caller is done, it is though the RFI instruction. The currently defined PAL entrypoints are PALE_RESET, PALE_INIT, PALE_PMI, and PALE_CHECK.

Machine Check (MC)

A machine check is a hardware event that indicates that a hardware error or architectural violation has occurred that threatens to damage the architectural state of the machine, possibly causing data corruption. The occurrence of the error triggers the execution of firmware code which records information about the error, and attempts to recover when possible.

OLR

On line replacement. The replacement of a computer component while the system is up and running without requiring a reboot.

Preserved

When applied to an entrypoint, preserved means that the value contained in a register at exit from the entrypoint code is the same as the value at the time of the hardware event that caused the entrypoint to be invoked. When applied to a procedure, preserved means that the value contained in a register at exit from the procedure is the same as the value at entry to the procedure. The value may have been changed and restored before exit.

Processor Abstraction Layer (PAL)

PAL is firmware that abstracts processor implementation differences and provides a consistent interface to higher level firmware and software. PAL has no knowledge of platform implementation details.

Procedure

A firmware procedure is a piece of code which is called from other firmware or software, and which uses the return mechanism of the Itanium Runtime Calling Conventions to return to its caller.

Reserved

When applied to a data variable, it means that the variable must not be used to convey information. All software passing the variable must place a value of zero in the variable. The occurrence of a non-zero value may cause undefined results.

When applied to a value or range of values, any values not defined in the range and specified as reserved must not be used. The occurrence of a reserved value may cause undefined results.

Scratch

When applied to either an entrypoint or procedure, scratch means that the contents of the register has no meaning and need not be preserved. Further the register is available for the storage of local variables. Unless otherwise noted, the register should not be relied upon to contain any particular value after exit.

Stacked Calling Convention

The firmware calling convention which adheres fully to the Itanium Runtime Calling Conventions. To use this calling convention, the RSE must be working and usable.

Static Calling Convention

The firmware calling convention which adheres to the Itanium Runtime Calling Conventions for the static general registers, branch registers, predicate registers, but for which all other registers are unused except for the RSE control registers. The RSE is placed in enforced lazy mode, and the stacked general registers or memory are not referenced.

System Abstraction Layer (SAL)

SAL is firmware that abstracts platform implementation differences for higher level software. SAL has no knowledge of processor implementation details.

Unchanged

When applied to an entrypoint, unchanged means that the register referenced has not been changed from the time of the hardware event that caused the entrypoint to be invoked until it exited to higher level firmware or software. When applied to a procedure, unchanged means that the register referenced has not been changed from procedure entry until procedure exit. In all cases, the value at exit is the same as the value at entry or the occurrence of the hardware event.

11.8 PAL Code Memory Accesses and Restrictions

PAL issues load and store operations to memory in the following cases with the following memory attributes:

- during machine check/INIT handling to the min-state save area memory region registered with PAL using the UC memory attribute
- during the execution of PAL procedures to the memory buffer allocated by the caller of the procedure using the memory attribute of the address passed by the caller.
- PAL may also issue loads from the architected firmware address space and loads/stores from the registered min-state save area whenever it is executing a PAL procedure or handling PAL based interruptions (reset, MCA, INIT and PMI). PAL code may use either the UC or WBL memory attribute when accessing these areas.

PAL code will not send IPIs that require any special support from the platform.

11.9 PAL Procedures

PAL procedures may be called by higher-level firmware and software to obtain information about the identification, configuration, and capabilities of the processor implementation, or to perform implementation-dependent functions such as cache initialization. These procedures access processor implementation-dependent hardware to return information that characterizes and identifies the processor or implements a defined function on that particular processor.

PAL procedures are implemented by a combination of firmware code and hardware. The PAL procedures are defined to be relocatable from the firmware address space. Higher level firmware and software must perform this relocation during the reset flow. The PAL procedures may be called both before and after this relocation occurs, but performance will usually be better after the relocation. In order to ensure no problems occur due to the relocation of the PAL procedures, these procedures are written to be position independent. All references to constant data done by the procedures is done in an IP relative way.

PAL procedures are provided to return information or allow configuration of the following processor features:

- Cache and memory features supported by the processor
- Processor identification, features, and configuration
- Machine Check Abort handling
- Power state information and management
- Processor self test
- Firmware utilities

PAL procedures are implemented as a single high level procedure, named PAL_PROC, whose first argument is an index which specifies which PAL procedure is being called. Indices are assigned depending on the nature of the PAL procedure being referenced, according to Table 11-11.

Table 11-11. PAL Procedure Index Assignment

Index	Description
0	Reserved
1 - 255	Architected procedures; static register calling conventions
256 - 511	Architected procedures; stacked register calling conventions
512 - 767	Implementation-specific procedures; static registers calling conventions
768 - 1023	Implementation-specific procedures; stacked register calling conventions
1024 +	Reserved

The assignment of indices for all architected procedures is controlled by this document. The assignment of indices for implementation-specific procedures is controlled by the specific processor for which the procedures are implemented. No implementation-specific procedure calls are required for the correct operation of a processor. No SAL or operating system code should ever have to call an implementation-specific procedure call for normal activity. They are reserved for diagnostic and bring-up software and the results of such calls may be unpredictable.

Architected procedures may be designated as required or optional. If a procedure is designated as optional, a unique return code will be returned to indicate the procedure is not present in this PAL implementation. It is the caller’s responsibility to check for this return code after calling any optional PAL procedure

In addition to the calling conventions described below, PAL procedure calls may be made in physical mode (PSR.it=0, PSR.rt=0, and PSR.dt=0) or virtual mode (PSR.it=1, PSR.rt=1, and PSR.dt=1). All PAL procedures may be called in physical mode. Only those procedures specified later in this chapter may be called in virtual mode. PAL procedures written to support virtual mode, and the caller of PAL procedures written in virtual mode must obey the restrictions documented in this chapter, otherwise the results of such procedure calls may be unpredictable.

11.9.1 PAL Procedure Summary

The following tables summarize the PAL procedures by application area. Included are the name of the procedure, the index of the procedure, the class of the procedure (whether required or optional), and the calling convention used for the procedure (static or stacked), and whether the procedure can be called in physical mode only or both physical and virtual modes.

Table 11-12. PAL Cache and Memory Procedures

Procedure	Idx	Class	Conv.	Mode	Description
PAL_CACHE_FLUSH	1	Req.	Static	Both	Flush the instruction or data caches.
PAL_CACHE_INFO	2	Req.	Static	Both	Return detailed instruction or data cache information.
PAL_CACHE_INIT	3	Req.	Static	Phys.	Initialize the instruction or data caches.
PAL_CACHE_PROT_INFO	38	Req.	Static	Both	Return instruction or data cache protection information.
PAL_CACHE_SUMMARY	4	Req.	Static	Both	Return a summary of the cache hierarchy.
PAL_MEM_ATTRIB	5	Req.	Static	Both	Return a list of supported memory attributes.
PAL_PREFETCH_VISIBILITY	41	Req.	Static	Both	Used in architected sequence to transition pages from a cacheable, speculative attribute to an uncacheable attribute. See Section 4.4.11.2 "Physical Addressing Attribute Transition - Disabling Prefetch/Speculation and Removing Cacheability".
PAL_PTCE_INFO	6	Req.	Static	Both	Return information needed for `ptc.e` instruction to purge entire TC.
PAL_VM_INFO	7	Req.	Static	Both	Return detailed information about virtual memory features supported in the processor.
PAL_VM_PAGE_SIZE	34	Req.	Static	Both	Return virtual memory TC and hardware walker page sizes supported in the processor.
PAL_VM_SUMMARY	8	Req.	Static	Both	Return summary information about virtual memory features supported in the processor.
PAL_VM_TR_READ	261	Req.	Stacked	Phys.	Read contents of a translation register.

Table 11-13. PAL Processor Identification, Features, and Configuration Procedures

Procedure	Idx	Class	Conv.	Mode	Description
PAL_BUS_GET_FEATURES	9	Req.	Static	Phys.	Return configurable processor bus interface features and their current settings.
PAL_BUS_SET_FEATURES	10	Req.	Static	Phys.	Enable or disable configurable features in processor bus interface.
PAL_DEBUG_INFO	11	Req.	Static	Both	Return the number of instruction and data breakpoint registers.
PAL_FIXED_ADDR	12	Req.	Static	Both	Return the fixed component of a processor’s directed address.
PAL_FREQ_BASE	13	Opt.	Static	Both.	Return the frequency of the output clock for use by the platform, if generated by the processor.
PAL_FREQ_RATIOS	14	Req.	Static	Both.	Return ratio of processor, bus, and interval time counter to processor input clock or output clock for platform use, if generated by the processor.
PAL_PERF_MON_INFO	15	Req.	Static	Both	Return the number and type of performance monitors.
PAL_PLATFORM_ADDR	16	Req.	Static	Both	Specify processor interrupt block address and I/O port space address.
PAL_PROC_GET_FEATURES	17	Req.	Static	Phys.	Return configurable processor features and their current setting.
PAL_PROC_SET_FEATURES	18	Req.	Static	Phys.	Enable or disable configurable processor features.
PAL_REGISTER_INFO	39	Req.	Static	Both	Return AR and CR register information.
PAL_RSE_INFO	19	Req.	Static	Both	Return RSE information.
PAL_VERSION	20	Req.	Static	Both	Return version of PAL code.

Table 11-14. PAL Machine Check Handling Procedures

Procedure	Idx	Class	Conv.	Mode	Description
PAL_MC_CLEAR_LOG	21	Req.	Static	Both	Clear all error information from processor error logging registers.
PAL_MC_DRAIN	22	Req.	Static	Both	Ensure that all operations that could cause an MCA have completed.
PAL_MC_DYNAMIC_STATE	24	Opt.	Static	Phys.	Return Processor Dynamic State for logging by SAL.
PAL_MC_ERROR_INFO	25	Req.	Static	Both	Return Processor Machine Check Information and Processor Static State for logging by SAL.
PAL_MC_EXPECTED	23	Req.	Static	Phys.	Set/Reset Expected Machine Check Indicator.
PAL_MC_REGISTER_MEM	27	Req.	Static	Phys.	Register min-state save area with PAL for machine checks and inits.
PAL_MC_RESUME	26	Req.	Static	Phys.	Restore minimal architected state and return to interrupted process.

Table 11-15. PAL Power Information and Management Procedures

Procedure	Idx	Class	Conv.	Mode	Description
PAL_HALT	28	Opt.	Static	Phys	Enter the low-power HALT state or an implementation-dependent low-power state.
PAL_HALT_INFO	257	Req.	Stacked	Both	Return the low power capabilities of the processor.
PAL_HALT_LIGHT	29	Req.	Static	Both	Enter the low power LIGHT HALT state

Table 11-16. PAL Processor Self Test Procedures

Procedure	Idx	Class	Conv.	Mode	Description
PAL_CACHE_LINE_INIT	31	Req.	Static	Phys.	Initialize tags and data of a cache line for processor testing.
PAL_CACHE_READ	259	Opt.	Stacked	Phys.	Read tag and data of a cache line for diagnostic testing.
PAL_CACHE_WRITE	260	Opt.	Stacked	Phys.	Write tag and data of a cache for diagnostic testing.
PAL_TEST_INFO	37	Req.	Static	Phys.	Returns alignment and size requirements needed for the memory buffer passed to the PAL_TEST_PROC procedure as well as information on self-test control words for the processor self tests.
PAL_TEST_PROC	258	Req.	Stacked	Phys.	Perform late processor self test.

Table 11-17. PAL Support Procedures

Procedure	Idx	Class	Conv.	Mode	Description
PAL_COPY_INFO	30	Req.	Static	Phys.	Return information needed to relocate PAL procedures and PAL PMI code to memory.
PAL_COPY_PAL	256	Req.	Stacked	Phys.	Relocate PAL procedures and PAL PMI code to memory.
PAL_ENTER_IA_32_ENV	33	Opt.	Static	Phys.	Enter IA-32 System environment.
PAL_PMI_ENTRYPOINT	32	Req.	Static	Phys.	Register PMI memory entrypoints with processor.

11.9.2 PAL Calling Conventions

The following general rules govern the definition of the PAL procedure calling conventions.

11.9.2.1 Overview of Calling Conventions

There are two calling conventions supported for PAL procedures: static registers only and stacked registers. Any single PAL procedure will support only one of the two calling conventions. In addition, PAL procedure may be called in either physical mode (PSR.it=0, PSR.rt=0, and PSR.dt=0) or virtual mode (PSR.it=1, PSR.rt=1, and PSR.dt=1).

11.9.2.1.1 Static Registers Only

This calling convention is intended for boot time usage before main memory may be available or error recovery situations, where memory or the RSE may not be reliable. All parameters are passed in the lower 32 static general registers. The stacked registers will not be used within the procedure. No memory arguments may be passed as parameters to or from PAL procedures written using the static register calling convention. To avoid RSE activity, static register PAL procedures must be called with the br.cond instruction, not the br.call instruction. Please refer to Table 11-21 for a detailed list of the general register usage for static registers only calling convention.

11.9.2.1.2 Stacked Registers

This calling convention is intended for usage after memory has been made available, and for procedures which require memory pointers as arguments. The stacked registers are also used for parameter passing and local variable allocation. This convention conforms to the *Itanium Software Conventions and Runtime Architecture Guide*. Thus, procedures using the stacked register calling convention can be written in the C language. There is one exception to the runtime conventions. The first argument to the procedure must also be copied to GR28 prior to making the procedure call. This allows procedures written using both static and stacked register calling conventions to call a single PAL_PROC entrypoint. This should be accomplished by having the stacked register procedures call a stub module which copies GR32 to GR28, then call PAL_PROC. It is the responsibility of the caller to provide this stub. Please refer to Table 11-22 for a detailed list of the general register usage for the stacked register calling convention.

11.9.2.1.3 Making PAL Procedure Calls in Physical or Virtual Mode

PAL procedure calls which are made in physical mode must obey the calling conventions described in this chapter, but there are no additional restrictions beyond those noted above. PAL procedure calls made in virtual mode must have the region occupied by PAL_PROC virtually mapped with an ITR. It needs to map this same area with either a DTR or DTC using the same translation as the ITR. In addition, it must also provide a DTR or DTC mapping for any memory buffer pointers passed as arguments to a procedure. It is the responsibility of the caller to provide these mappings.

If the caller chooses to map the PAL_PROC area or any memory pointers with a DTC it must call the procedure with PSR.ic = 1 to handle any TLB faults that could occur. The PAL_PROC code needs to be mapped with an ITR. Unpredictable results may occur if it is mapped with an ITC register.

11.9.2.2 Processor State

The PAL procedures are only available to the code running at privilege level 0. They must run in physical mode (unless specified as callable in virtual mode). PAL procedures are not interruptible by external interrupt or NMI, since PSR.i must be 0 when the PAL procedure is called. PAL procedures are not interruptible by PMI events, if PSR.ic is 0. If PSR.ic is 1, PAL procedures can be interrupted by PMI events. PAL procedures can be interrupted by machine checks and initialization events.

Generally PAL procedures will not enable interruptions not already enabled by the caller. Any PAL call that might cause interruptions (besides data TLB faults, see Section 11.9.2.1.3), must install an IVA handler to handle them. PAL_TEST_PROC may generate any interruptions it needs to test the processor.

Table 11-18 defines the requirements for the PSR at entry to and at exit from a PAL procedure call. The operating system must follow the state requirements for PSR shown below. PAL procedure calls made by SAL may impose additional requirements. PAL_TEST_PROC may change PSR bits shown as unchanged in order to test the processor. These bits will be preserved in this case. PSR bits are described in increasing bit number order. Any PSR bit numbers not specified are reserved and unchanged.

Table 11-18. State Requirements for PSR

PSR bit	Description	Entry	Exit	Class
be	big-endian memory access enable	0	0	preserved
up	user performance monitor enable	C	C	unchanged
ac	alignment check	C	C	preserved
mfl	floating-point registers f2-f31 written	C	C	preserved
mfh	floating-point registers f32-f127 written	C	C	preserved
ic	interruption state collection enable	0	0	unchanged
		1	1	preserved
i	interrupt enable	0	0	unchanged
pk	protection key validation enable	C	C	unchanged
dt	data address translation enable[a]	0	0	unchanged
		1	1	preserved
dfl	disabled FP register f2 to f31	0	0	unchanged
dfh	disabled FP register f32 to f127[b]	0	0	unchanged
		1	1	unchanged
sp	secure performance monitors	C	C	unchanged
pp	privileged performance monitor enable	C	C	unchanged
di	disable ISA transition	C	C	preserved
si	secure interval timer	C	C	unchanged
db	debug breakpoint fault enable	0	0	unchanged
lp	lower-privilege transfer trap enable	0	0	unchanged
tb	taken branch trap enable	0	0	unchanged
rt	register stack translation enable[a]	0	0	unchanged
		1	1	preserved
cpl	current privilege level	0	0	unchanged
is	instruction set	0	0	preserved
mc	machine check abort mask[c]	0	0	preserved
		1	1	unchanged
it	instruction address translation enable[a]	0	0	unchanged
		1	1	preserved
id	instruction debug fault disable	0	0	unchanged
da	data access and dirty-bit fault disable	0	0	unchanged
dd	data debug fault disable	0	0	unchanged
ss	single step trap enable	0	0	unchanged
ri	restart instruction	0	0	preserved

Table 11-18. State Requirements for PSR (Continued)

PSR bit	Description	Entry	Exit	Class
ed	exception deferral	0	0	preserved
bn	register bank	1	1	preserved
ia	instruction access-bit fault disable	0	0	unchanged

a. PAL procedures which are called in physical mode must remain in physical mode for the duration of the call. PAL procedures which are called in virtual mode, may perform specific actions in physical mode, but must return to the same virtual mode state before returning from the call.

b. PAL_TEST_PROC and an implementation-specific authentication procedure call need to be called with PSR.dfh equal to 0. If they are not they will return invalid argument. All other PAL procedure calls may be called with PSR.dfh equal to 0 or 1.

c. Most PAL runtime procedures should be called with PSR.mc = 0 except for code flow involved in handling machine checks.

11.9.2.2.1 Definition of Terms

The terms used in the definition of the requirements have the following meaning:

Table 11-19. Definition of Terms

Term	Description
entry	Start of the first instruction of the PAL procedure.
exit	Start of the first instruction after return to caller’s code.
0	Must be zero at entry to the procedure or on exit from the procedure. If the value at entry is not zero, the procedure may return an illegal argument or execute in an undefined manner.
1	Must be one at entry to the procedure or on exit from the procedure. If the value at entry is not one, the procedure may return an illegal argument or execute in an undefined manner.
reserved	When any input parameter is listed as reserved, this value must be zero. If an input value has a range of values, any values outside the range, listed as reserved, must not be used. For either case, the PAL procedure may return an illegal argument or execute in an undefined manner.
C	The state of bits marked with C are defined by the caller. If the value at exit is also C, it must be the same as the value at entry.
unchanged	The PAL procedure must not change these values from their entry values during execution of the procedure.
scratch	The PAL procedure may modify these values as necessary during execution of the procedure. The caller cannot rely on these values.
preserved	The PAL procedure may modify these values as necessary during execution of the procedure. However, they will be restored to their entry values prior to exit from the procedure.

11.9.2.2.2 System Registers

The PAL_TEST_PROC procedure may change system registers marked as unchanged in order to fully test the processor. When this is done, the values of the system registers will be preserved.

Table 11-20. System Register Conventions

Name	Description	Class
DCR	Default Control Register	preserved
ITM	Interval Timer Match Register	unchanged
IVA	Interruption Vector Address	preserved
PTA	Page Table Address	preserved

Table 11-20. System Register Conventions (Continued)

Name	Description	Class
IPSR	Interruption Processor Status Register	scratch
ISR	Interruption Status Register	scratch
IIP	Interruption Instruction Bundle Pointer	scratch
IFA	Interruption Faulting Address	scratch
ITIR	Interruption TLB Insertion Register	scratch
IIPA	Interruption Instruction Previous Address	scratch
IFS	Interruption Function State	scratch
IIM	Interruption Immediate Register	scratch
IHA	Interruption Hash Address	scratch
LID	Local Interrupt ID	unchanged
IVR	Interrupt Vector Register (read only)	unchanged
TPR	Task Priority Register	unchanged
EOI	End Of Interrupt	unchanged
IRR0-IRR3	Interrupt Request Registers 0-3 (read only)	unchanged
ITV	Interval Timer Vector	unchanged
PMV	Performance Monitoring Vector	unchanged
CMCV	Corrected Machine Check Vector	unchanged
LRR0-LRR1	Local Redirection Registers 0-1	unchanged
RR	Region Registers	preserved
PKR	Protection Key Registers	preserved
TR	Translation Registers	unchanged[a]
TC	Translation Cache	scratch
IBR/DBR	Break Point Registers	preserved
PMC	Performance Monitor Control Registers	preserved
PMD	Performance Monitor Data Registers	unchanged[b]

[a] If an implementation provides a means to read TRs for PAL, this should be preserved.

[b] No PAL procedure writes to the PMD. Depending on the PMC, the PMD may be kept counting performance monitor events during a procedure call. The exception is PAL_TEST_PROC, which tests the performance counters.

11.9.2.2.3 General Registers

PAL will use one of two general register calling conventions described in Section 11.9.2.1, depending on the availability of memory and the stacked registers at the time of the call. The following tables describe the contents of the general registers.

Table 11-21. General Registers - Static Calling Convention

Register	Conventions
GR0	always 0
GR1	preserved
GR2 - GR3	scratch, used with 22 bit immediate add
GR4 - GR7	preserved
GR8 - GR11	scratch, procedure return value
GR12	preserved
GR13	unchanged
GR14 - GR27	scratch

Table 11-21. General Registers - Static Calling Convention (Continued)

Register	Conventions
GR28 - GR31	input arguments, scratch (PAL index must be passed in GR28)
Bank 0 Registers (GR16 - GR23)	preserved
Bank 0 Registers (GR 24 - GR31)	scratch
GR32 - GR127	unchanged

Table 11-22. General Registers - Stacked Calling Conventions

Register	Conventions
GR0	always 0
GR1	preserved
GR2 - GR3	scratch, used with 22 bit immediate add
GR4 - GR7	preserved
GR8 - GR11	scratch, procedure return value
GR12	special, stack pointer (sp)
GR13	special, thread pointer (tp)
GR14 - GR27	scratch
GR28	input argument, scratch (PAL Index must be passed in GR28)
GR29-GR31	scratch
Bank 0 Registers (GR16 - GR23)	preserved
Bank 0 Registers (GR 24 - GR31)	scratch
GR32 - GR127	stacked registers; in0 -in95: input arguments (PAL index must be in0) loc0 - loc95: local variables out0 - out95: output arguments

The caller must initialize SP for physical and virtual procedure calls only prior to calling a PAL procedure. A minimum 8 KB of room must be available for the stack space of the PAL procedure. The caller to a PAL procedure should set up the RSE backing store before making any procedure calls using the stacked calling conventions. The backing store memory should have a minimum of 8 KB of room for RSE spills.

PAL shall be called with PSR.bn=1. The GR specifications for GR16 through GR31 are for bank one. The bank zero register requirements are specified as a separate line item.

11.9.2.2.4 Floating-point Registers

Although there is no PAL procedure that passes floating-point parameters, the floating-point register conventions are the same as those of the *Itanium Software Conventions and Runtime Architecture Guide*.

11.9.2.2.5 Predicate Registers

The conventions for the predicate registers follow the *Itanium Software Conventions and Runtime Architecture Guide*.

11.9.2.2.6 Branch Registers

The conventions for the branch registers follow the *Itanium Software Conventions and Runtime Architecture Guide*.

11.9.2.2.7 Application Registers

Table 11-23. Application Register Conventions

Register	Description	Class
KR0-7	Kernel Registers	unchanged
RSC	Register Stack Configuration Register	unchanged
BSP	Backing Store Pointer (read only)	unchanged[a]
BSPSTORE	Backing Store Pointer for Memory Stores	unchanged[a]
RNAT	RSE NaT Collection Register	unchanged[a]
FCR	IA-32 Floating-point Control Registers	preserved
EFLAG	IA-32 EFLAG register	preserved
CSD	IA-32 Code Segment Descriptor	preserved
SSD	IA-32 Stack Segment Descriptor	preserved
CFLG	IA-32 Combined CR0 and CR4 Register	preserved
FSR	IA-32 Floating-point Status Register	preserved
FIR	IA-32 Floating-point Instruction Register	preserved
FDR	IA-32 Floating-point Data Register	preserved
CCV	Compare and Exchange Compare Value Register	scratch
UNAT	User NaT Collection Register	according to GR class
FPSR	Floating-point Status Register	preserved
ITC	Interval Time Counter	unchanged[b]
PFS	Previous Function State	preserved
LC	Loop Counter Register	preserved
EC	Epilog Counter Register	preserved

a. BSP, BSPSTORE, and RNAT may not be changed by PAL, but the value at exit may be different due to RSE activity. PAL_TEST_PROC is an exception to this rule, and the RSE contents may not be relied on after making this procedure call.

b. No PAL procedure writes to the ITC. The value at exit is the value at entry plus the elapsed time of the procedure call.

PAL procedures that use the static calling conventions do not use stacked registers or the RSE. Therefore RSE internal state and the CFM are unchanged by these procedures.

11.9.2.3 Return Buffers

Any addresses passed to PAL procedures as buffers for return parameters must be 8-byte aligned. Unaligned addresses may cause undefined results.

11.9.2.4 Invalid Arguments

The PAL procedure calling conventions specify rules that must be followed. These rules specify certain PSR values, they specify that reserved fields and arguments must be zero filled and specify that values not defined in a range and defined as reserved must not be used.

If the caller of a PAL procedure does not follow these rules, an invalid argument return value may be returned or undefined results may occur during the execution of the procedure.

11.9.3 PAL Procedure Specifications

The following pages provide detailed interface specifications for each of the PAL procedures defined in this document. Included in the specification are the input parameters, the output parameters, and any required behavior.

Get Processor Bus Dependent Configuration Features

Purpose: Provides information about configurable processor bus features.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_BUS_GET_FEATURES within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_BUS_GET_FEATURES procedure.
features_avail	64-bit vector of features implemented. See Table 11-24. (1-implemented, 0=not implemented)
feature_status	64-bit vector of current feature settings. See Table 11-24.
feature_control	64-bit vector of features controllable by software. (1=controllable, 0= not controllable)

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: Table 11-24 defines the set of possible bus interface features and their bit position in the return vector. Different busses will implement similar features in different ways. For example, data error detection may be implemented by ECC or parity. In other cases, certain features may be tied together. In this case, enabling any one feature in a group will enable all features in the group, and similarly, disabling any one feature in a group will disable all features. Caller algorithms should be written to obtain desired results in these instances. Only those configuration features for which a 1 is returned in *feature_control* can be changed via PAL_BUS_SET_FEATURES.

For all values in Table 11-24, the *Class* field indicates whether a feature is required to be available (Req.) or is optional (Opt.). The *Control* field indicates which features are required to be controllable. These features will either be controllable through this PAL call or through other hardware means like forcing bus pins to a certain value during processor reset. The *control* field applies only when the feature is available. The sense of the bits is chosen so that for features which are controllable, the default hand-off value at exit from PALE_RESET should be 0. PALE_CHECK and PALE_INIT should not modify these features. An operating system should not modify bus features without detailed information about the platform it is running on.

Table 11-24. Processor Bus Features

Bit	Class	Control	Description
63	Opt.	Req.	Disable Bus Data Error Checking. When 0, bus data errors are detected and single bit errors are corrected. When 1, no error detection or correction is done.
62	Opt.	Req.	Disable Bus Address Error Signalling. When 0, bus address errors are signalled on the bus. When 1, no bus errors are signalled on the bus. If Disable Bus Address Error Checking is 1, this bit is ignored.
61	Opt.	Req.	Disable Bus Address Error Checking. When 0, bus errors are detected, single bit errors are corrected., and a CMCI or MCA is generated internally to the processor. When 1, no bus address errors are detected or corrected.
60	Opt.	Req.	Disable Bus Initialization Event Signalling. When 0, bus protocol errors are signalled on the bus.When 1, bus protocol errors are not signalled on the bus. If Disable Bus Initialization Event Checking is 1, this bit is ignored.

Table 11-24. Processor Bus Features (Continued)

Bit	Class	Control	Description
59	Opt.	Req.	Disable Bus Initialization Event Checking. When 0, bus protocol errors are detected and single bit errors are corrected, and a CMCI or MCA is generated internally to the processor. When 1, no bus protocol checking is done.
58	Opt.	Req.	Disable Bus Requester Bus Error Signalling. When 0, BERR# is signalled if a bus error is detected. When 1, bus errors are not signalled on the bus.
57	Opt.	Req.	Disable Bus Requester Internal Error Signalling. When 0, BERR# is signalled when internal processor requestor initiated bus errors are detected. When 1, internal requester bus errors are not signalled on the bus.
56	Opt.	Req.	Disable Bus Error Checking. When 0, the processor takes an MCA if BERR# is asserted. When 1, the processor ignores the BERR# signal.
55	Opt.	Req.	Disable Response Error Checking. When 0, the processor asserts BINIT# if it detects a parity error on the signals which identify the transactions to which this is a response. When 1, the processor ignores parity on these signals.
54	Opt.	Req.	Disable Transaction Queuing. When 0, the in-order transaction queue is limited only by the number of hardware entries. When 1, the processor's in-order transactions queue is limited to one entry.
53	Opt.	Req.	Enable a bus cache line replacement transaction when a cache line in the exclusive state is replaced from the highest level processor cache and is not present in the lower level processor caches. When 0, no bus cache line replacement transaction will be seen on the bus. When 1, bus cache line replacement transactions will be seen on the bus when the above condition is detected.
52	Opt.	Req.	Enable a bus cache line replacement transaction when a cache line in the shared state is replaced from the highest level processor cache and is not present in the lower level processor caches. When 0, no bus cache line replacement transaction will be seen on the bus. When 1, bus cache line replacement transactions will be seen on the bus when the above condition is detected.
51-32	N/A	N/A	Reserved
31	Opt.	Opt.	Enable Half transfer rate. When 0, the data bus is configured at the 2x data transfer rate.When 1, the data bus is configured at the 1x data transfer rate,
30	Opt.	Req.	Disable Bus Lock Mask. When 0, the processor executes locked transactions atomically. When 1, the processor masks the bus lock signal and executes locked transactions as a non-atomic series of transactions.
29	Req.	Req.	Request Bus Parking. When 0, the processor will deassert bus request when finished with each transaction. When 1, the processor will continue to assert bus request after it has finished, if it was the last agent to own the bus and if there are no other pending requests.
28-0	N/A	N/A	Reserved

Set Processor Bus Dependent Configuration Features

Purpose: Enables/disables specific processor bus features.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_BUS_SET_FEATURES within the list of PAL procedures.
feature_select	64-bit vector denoting desired state of each feature (1=select, 0=non-select).
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_BUS_SET_FEATURES procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Can not complete call without error

Description: PAL_BUS_GET_FEATURES should be called to ascertain the implemented processor bus configuration features, their current setting, and whether they are software controllable, before calling PAL_BUS_SET_FEATURES. The list of possible processor features is defined in Table 11-24. Attempting to enable or disable any feature that cannot be changed will be ignored.

Flush Data or Instruction Caches

Purpose: Flushes the processor instruction or data caches.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_CACHE_FLUSH within the list of PAL procedures.
cache_type	Unsigned 64-bit integer indicating which cache to flush. See Table 11-25.
operation	Formatted bit vector indicating the operation of this call. See Figure 11-18.
progress_indicator	Unsigned 64-bit integer specifying the starting position of the flush operation.

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_FLUSH procedure.
vector	Unsigned 64-bit integer specifying the vector number of the pending interrupt.
progress_indicator	Unsigned 64-bit integer specifying the starting position of the flush operation.
Reserved	0

Status:

Status Value	Description
2	Call completed without error, but a PMI was taken during the execution of this procedure.
1	Call has not completed flushing due to a pending interrupt
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: Flushes the instruction or data caches controlled by the processor as specified by the *cache_type* parameter. Encoding for the *cache_type* parameter follows:

Table 11-25. *cache_type* Encoding

Value	Description
1	Flush all caches containing instructions.
2	Flush all caches containing data.
3	Flush all caches (instruction and data).
4	Make local instruction caches coherent with the data caches.

All other values of *cache_type* are reserved. If the cache is unified, both instruction and data lines are flushed, regardless of the value of *cache_type*.

Flushing all caches containing instructions, causes the instruction and unified caches to be flushed. Flushing all caches containing data, causes all data and unified caches to be flushed. Flushing all caches causes all data, instruction, and unified caches to be flushed.

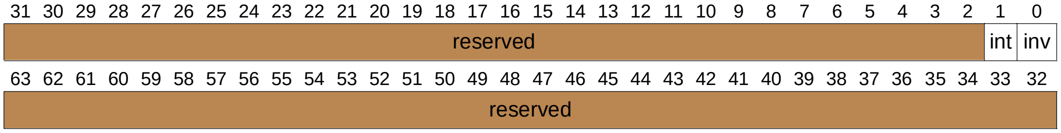
When the caller specifies to make local instruction caches coherent with the data caches, this procedure will ensure that the local instruction caches will see the effects of stores of instruction code done on the processor. Refer to Section 4.4.3 "Cacheability and Coherency Attribute" on page 2:65 for more information on stores and their coherency requirements with local instruction caches.

The effects of flushing data and unified caches is broadcast throughout the coherency domain. The effects of flushing instruction caches may or may not be broadcast throughout the coherency domain. The procedure will perform the necessary serialization and synchronization as required by the architecture.

This call does not ensure that data in the processors coalescing buffers are flushed to memory. See Section 4.4.5 on page 2:66 on how to flush the coalescing buffers.

The *operation* parameter controls how this call will operate. The *operation* parameter has the following format:

Figure 11-18. *operation* Parameter Layout



- *inv* - 1 bit field indicating whether to invalidate clean lines in the cache.

  If this bit is 0, all modified cache lines for the specified *cache_type* are copied back to memory. Optimally, modified and non-modified cache lines are left valid in the specified cache in a clean (non-modified) state. However based on the processor implementation cache lines in the specified cache may alternatively be invalidated.

  If this bit is 1, all modified cache lines for the specified *cache_type* are flushed by copying the cache line to memory. All cache lines in the specified cache are then invalidated.

  If *cache_type* is equal to 4 (make local instruction caches coherent with the data caches) the *inv* bit will be ignored.

  Table 11-26 will clarify the effects of the *inv* bit. The modified state represents a cache line that contains modified data. The clean state represents a cache line that contains no modified data.

- *int* - 1 bit field indicating if the processor will periodically poll for external interrupts while flushing the specified *cache_type*(s).

  If this bit is a 0, unmasked external interrupts will not be polled. The processor will ignore all pending unmasked external interrupts until all cache lines in the specified *cache_type*(s) are flushed. Depending on the size of the processor's caches, bus bandwidth and implementation characteristics, flushing the caches can take a long period of time, possibly delaying interrupt response times and potentially causing I/O devices to fail.

  If this bit is a 1, external interrupts will be polled periodically and will exit the procedure if one is seen. If an unmasked external interrupt becomes pending, this procedure will return and allow the caller to service the interrupt before all cache lines in the specified *cache_type*(s) are flushed.

Table 11-26. Cache Line State when *inv* = 0

Old State	New State	Comments
Invalid	Invalid	
Clean	Clean[a]	
Modified	Clean[a]	Modified data is copied back to memory

[a] Based on the processor implementation the cache line can be invalidated or left in a model-specific clean state

Table 11-27. Cache Line State when *inv* = 1

Old State	New State	Comments
Invalid	Invalid	
Clean	Invalid	
Modified	Invalid	Modified data is copied back to memory.

The *progress_indicator* is an unsigned 64-bit integer specifying the starting position of the flush operation. Values in this parameter are model specific and will vary across processor implementations.

The first time this procedure is called, the *progress_indicator* must be set to zero. If this procedure exits due to an external interrupt and this procedure is then again called to resume flushing, the *progress_indicator* must be set to the value previously returned by PAL_CACHE_FLUSH. Software must program no value other than zero or the value previously returned by PAL_CACHE_FLUSH otherwise behavior is undefined.

This procedure makes one flush pass through all caches specified by *cache_type* and all sets and associativities within those caches. The specified *cache_type*(s) are ensured to be flushed only of cache lines resident in the caches prior to PAL_CACHE_FLUSH initially being called with the *progress_indicator* set to 0.

This procedure ensures that prefetches initiated prior to making this call with *progress_indicator* set to 0 are flushed based on the *cache_type* argument passed.

- If *cache_type* specifies to flush all instruction caches then the call ensures all prior instruction prefetches are flushed.
- If *cache_type* specifies to flush all data caches then the call ensures all prior data prefetches are flushed.
- If *cache_type* specifies to flush all caches then the call ensures all prior instruction and data prefetches are flushed from the caches.
- If *cache_type* specifies to make local instruction caches coherent with the data caches, then the call will ensure all prior instruction prefetches are flushed.

Due to the following conditions, software cannot assume that when this procedure completes the entire flush pass that the specified *cache_type*(s) are empty of all clean and/or modified cache lines.

- After an interruption, the flush pass resumes at the interruption point (specified by *progress_indicator*). Due to execution of the interrupt handlers during the flush pass, the specified caches may contain new and possibly modified cache lines in sections of the caches already flushed. The caller specifies if this procedure should poll for interrupts via the *int* bit of the *operation* parameter.
- Prior prefetches initiated before this procedure is called are disabled and flushed from the cache as described above. However, if a speculative translation exists in either the ITLB or DTLB, speculative instruction or data prefetch operation could immediately reload a non-modified cache line after it was flushed. To ensure prefetches do not occur, software must remove all speculative translation before calling this routine. Alternatively, software can disable the TLBs by setting PSR.it, PSR.dt, and PSR.rt to 0.
- The specified caches may also contain PAL firmware code cache entries required to flush the cache.
- The specified caches may contain PAL and SAL PMI code if this call was made with PSR.ic = 1 and a PMI interrupt is seen during the execution of the call.
- The specified caches may contain SAL or OS machine check or INIT code if these handlers run in a cacheable mode and a machine check or INIT event is seen.

This procedure does ensure that all cache lines resident in the specified *cache_type*(s) prior to this procedure being initially called are flushed regardless of intervening external interrupts. It also ensures that prefetches initiated prior to the initial call to this procedure that affect the caches specified in *cache_type*, as described above, are flushed regardless of intervening external interrupts.

To ensure forward progress, PAL_CACHE_FLUSH advances through the cache flush sequence at least by one cache line before sampling for pending external interrupts. The amount of flushing that occurs before interrupts are polled will vary across implementations.

PAL_CACHE_FLUSH will return the following values to indicate to the caller the status of the call.

- *Status*

  When the call returns a 1, it indicates that the call did not have any errors but is returning due to a pending unmasked external interrupt. To continue flushing the caches, the caller must call PAL_CACHE_FLUSH again with the value returned in the *progress_indicator* return value.

  When the call returns a 0, it indicates that the call completed without any errors. All cache lines that were present in the cache (when the most recent call to PAL_CACHE_FLUSH with a *progress_indicator* of zero) are flushed and possibly invalidated. All intermediate calls must have used the proper *progress_indicator*, otherwise behavior is undefined.

  When the call returns a 2, it indicates that the call completed without any errors but that a PMI was taken during the execution of this call. This indicates to the caller that all cache lines that were present in the cache (when the most recent call to PAL_CACHE_FLUSH with a *progress_indicator* of zero) are flushed but that code and data related to handling PMIs may be present in the cache.
- *vector* - If the return status is 1 and this procedure exited due to a pending unmasked external interrupt, this field returns the interrupt vector number. The external interrupt will have been removed. The interrupt is considered to be “in-service” and software must service this interrupt for the specified vector and then issue EOI. If the return status is not 1, the values returned is undefined.
- *progress_indicator* - When the return status is 1, specifies the current position in the flush pass. The value returned is model specific and will vary across processor implementations. If the return status is not 1, the value returned is undefined.

Get Detailed Cache Information

Purpose: Returns information about a particular processor instruction or data cache at a specified level in the cache hierarchy.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_CACHE_INFO within the list of PAL procedures.
cache_level	Unsigned 64-bit integer specifying the level in the cache hierarchy for which information is requested. This value must be between 0 and one less than the value returned in the *cache_levels* return value from PAL_CACHE_SUMMARY.
cache_type	Unsigned 64-bit integer with a value of 1 for instruction cache and 2 for data or unified cache. All other values are reserved.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_INFO procedure.
config_info_1	The format of *config_info_1* is shown in Figure 11-19.
config_info_2	The format of *config_info_2* is shown in Figure 11-20.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This call describes in detail the characteristics of a given processor controlled cache in the cache hierarchy. It returns information in the *config_info_1* and *config_info_2* returns parameters.

The *config_info_1* return value has the following structure:

Figure 11-19. *config_info_1* Return Value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
stride								line_size								associativity								reserved					at		u

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
load_hints								store_hints								load_latency								store_latency							

- *u* - bit that is 1 if the cache is unified and 0 if the cache is split.
- *at* - 2-bit field denoting cache memory attributes, as follows:

Table 11-28. Cache Memory Attributes

Value	Description
0	Write through cache
1	Write back cache
2-3	Reserved

- *associativity* - unsigned 8-bit integer denoting the associativity of the cache. A value of 0 indicates a fully associative cache. A value of 1 indicates a direct mapped cache.
- *line_size* - unsigned 8-bit integer denoting the binary logarithm (log2) of the minimum write back size of a flush operation to memory or the line size of the cache if the cache contents never get flushed to memory (for example an instruction cache).
- *stride* - unsigned 8-bit integer denoting the binary log of the most effective stride in bytes for flushing the cache.
- *store_latency* - unsigned 8-bit integer denoting the number of cycles after issue until the value is written into the cache. If the cache cannot accept a store (like an instruction cache) the value returned will be 256 (0xff).
- *load_latency* - unsigned 8-bit integer denoting the number of processor cycles after issue until the value may be used if it is found in the cache.
- *store_hints* - 8-bit vector denoting hints implemented by the processor store instruction. For instruction caches this value will be a reserved value.

Table 11-29. Cache Store Hints

Bit #	Description
0	Temporal, level 1
1-2	Reserved
3	Non-temporal, all levels
4-7	Reserved

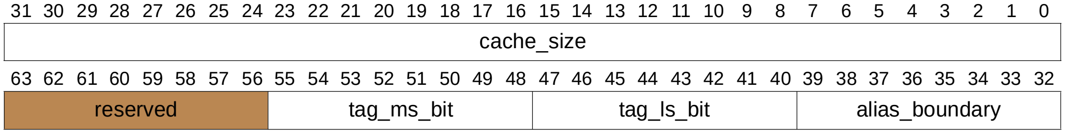
- *load_hints* - 8-bit vector denoting hints implemented by the processor load instruction.

Table 11-30. Cache Load Hints

Bit #	Hint
0	Temporal, level 1
1	Non-temporal, level 1
2	Reserved
3	Non-temporal, all levels
4-7	Reserved

The *config_info_2* return value has the following structure:

Figure 11-20. *config_info_2* Return Value



- *cache_size* - unsigned 32-bit integer denoting the size of the cache in bytes.
- alias_boundary - unsigned 8-bit integer indicating the binary log of the minimum number of bytes which must separate aliased addresses in order to obtain the highest performance.
- *tag_ls_bit* - unsigned 8-bit integer denoting the least-significant address bit of the tag.
- *tag_ms_bit* - unsigned 8-bit integer denoting the most-significant address bit of the tag.

Initialize Caches

Purpose: Initializes the processor controlled caches.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_CACHE_INIT within the list of PAL procedures.
level	Unsigned 64-bit integer containing the level of cache to initialize. If the cache level can be initialized independently, only that level will be initialized. Otherwise implementation-dependent side-effects will occur.
cache_type	Unsigned 64-bit integer with a value of 1 to initialize the instruction cache, 2 to initialize the data cache, or 3 to initialize both. All other values are reserved.
restrict	Unsigned 64-bit integer with a value of 0 or 1. All other values are reserved. If *restrict* is 1 and initializing the specified level and *cache_type* of the cache would cause side-effects, PAL_CACHE_INIT will return -4 instead of initializing the cache.

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_INIT procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-4	Call could not initialize the specified level and *cache_type* of the cache without side-effects and *restrict* was 1.

Description: Initializes one or all the processor's caches. The effect of this procedure is to initialize the caches without causing writebacks. This procedure cannot be used where coherency is required because any data in the caches will be lost.

The *level* argument must either be -1, indicating all cache levels, or a non-negative number indicating the specific level to initialize. In the latter case, *level* must be in the range from 0 up to one less than the *cache_levels* return value from PAL_CACHE_SUMMARY:

Table 11-31. PAL_CACHE_INIT *level* Argument Values

Value	Description
-1	Initializes all cache levels (*cache_type* and *restrict* are ignored)
0 to N	Initialize only the specified cache level.

The *restrict* argument specifies how to handle potential side-effects, where:

Table 11-32. PAL_CACHE_INIT *restrict* Argument Values

Value	Description
0	No restriction: initialize the specified level and *cache_type* of the cache, even if doing so will cause side effects in other caches.
1	Restrict initialization to the specified level and *cache_type* without side effects to other cache levels. If this cannot be done, return -4.

All other values of *restrict* are reserved.

Initialize a Data Cache line

Purpose: Initializes the tags and data of a data or unified cache line of a processor controlled cache to known values without the availability of backing memory.

Calling Conv: Static

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_CACHE_LINE_INIT within the list of PAL procedures.
address	Unsigned 64-bit integer value denoting the physical address from which the physical page number is to be generated. The address must be an implemented physical address, bit 63 must be zero.
data_value	64-bit data value which is used to initialize the cache line.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_LINE_INIT procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Can not complete call without error

Description: A line in the data or unified cache is initialized to the values passed in the arguments of this procedure. The physical page number of the line is derived from the *address* value passed. The tags of the line are set to Private, Dirty, and Valid. The cache line is initialized using *data_value* repeated until it fills the line. This procedure replicates *data_value* to a size equal to the largest line size in the processor-controlled cache hierarchy.

This procedure call cannot be used where coherency is required.

Get Detailed Cache Protection Information

Purpose: Returns protection information about a particular processor instruction or data cache at a specified level in the cache hierarchy.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_CACHE_PROT_INFO within the list of PAL procedures.
cache_level	Unsigned 64-bit integer specifying the level in the cache hierarchy for which information is requested. This value must be between 0 and one less than the value returned in the *cache_levels* return value from PAL_CACHE_SUMMARY.
cache_type	Unsigned 64-bit integer with a value of 1 for instruction cache and 2 for data or unified cache. All other values are reserved.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_PROT_INFO procedure.
config_info_1	The format of *config_info_1* is shown in Figure 11-21.
config_info_2	The format of *config_info_2* is shown in Figure 11-22.
config_info_3	The format of *config_info_3* is shown in Figure 11-23.

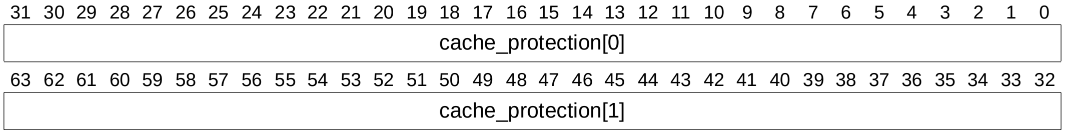
Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: PAL_CACHE_PROT_INFO returns information about the data and tag protection method for the specified cache. The three returns compose a six-element array of 32-bit protection information structures.

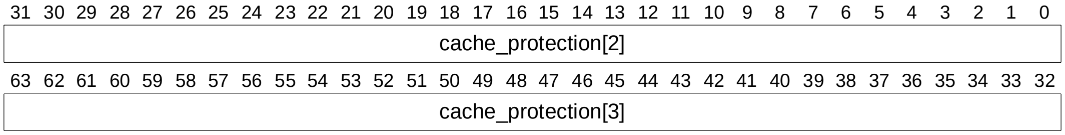
The *config_info_1* return value has the following structure:

Figure 11-21. *config_info_1* Return Value



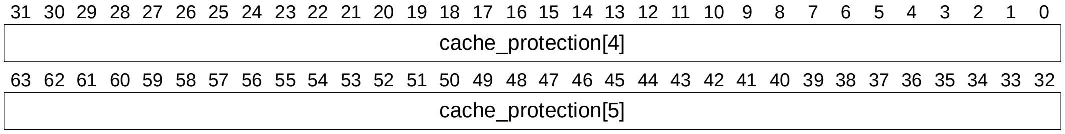
The *config_info_2* return value has the following structure:

Figure 11-22. *config_info_2* Return Value



The *config_info_3* return value has the following structure:

Figure 11-23. *config_info_3* Return Value



Each *cache_protection* element has the following structure:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
t_d		method				prot_bits						tagprot_msb						tagprot_lsb						data_bits							

- *data_bits* - unsigned 8-bit integer denoting the number of data bits that each unit of protection covers. For example, if the cache hardware generates 8 bits of ECC per 64 bits of data, *data_bits* would be 64. This field is only valid if *t_d* is 0, 2, or 3.
- *tagprot_lsb* - unsigned 6-bit integer denoting the least-significant tag address bit that this protection method covers. This field is only valid if *t_d* is 1, 2, or 3.
- *tagprot_msb* - unsigned 6-bit integer denoting the most-significant tag address bit that this protection method covers. This field is only valid if *t_d* is 1, 2, or 3.
- *prot_bits* - unsigned 6-bit integer denoting the number of protection bits generated for the field specified by the *t_d* element.
- *method* - unsigned 4-bit integer denoting the protection method, where:

Value	Description
0	no ECC or parity protection
1	odd parity protection
2	even parity protection
3	ECC protection

All other values of *method* are reserved.

- *t_d* - 2-bit field denoting whether this protection method applies to the tag, the data, or both. If over both, the tag and data unit could be concatenated with the tag either to the left (more significant) or to the right (less significant) than a unit of data. For the values of 2 and 3, the difference of *tagprot_msb* and *tagprot_lsb* indicates the number of tag bits that are protected with the data bits. The data bits are described by the *data_bits* field below. This field is encoded as follows:

Value	Description
0	Data protection
1	Tag protection
2	Tag+data protection (tag is more significant)
3	Data+tag protection (data is more significant)

When obtaining cache information via this call, information for the data cache should be obtained first, then if the *u* bit of the *config_info_1* parameter is not set, obtain the information for the instruction cache.

Read Values from the Processor Cache

Purpose: Reads the data and tag of a processor-controlled cache line for diagnostic testing.

Calling Conv: Stacked Registers

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_CACHE_READ within the list of PAL procedures.
line_id	8-byte formatted value describing where in the cache to read the data.
address	64-bit 8-byte aligned physical address from which to read the data. The address must be an implemented physical address on the processor model with bit 63 set to zero.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_READ procedure.
data	Right-justified value returned from the cache line.
length	The number of bits returned in *data*.
mesi	The status of the cache line.

Status:

Status Value	Description
1	The word at *address* was found in the cache, but the line was invalid.
0	Call completed without error.
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error.
-5	The word at *address* was not found in the cache.
-7	The operation requested is not supported for this *cache_type* and *level.*

Description: A value is read from the specified cache line, if present. This procedure allows reading cache data, tag, protection, or status bits.

The *line_id* argument is an 8-byte quantity in the following format:

Figure 11-24. Layout of *line_id* Return Value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
part								way								level								cache_type							

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved																															

- *cache_type* - unsigned 8-bit integer denoting whether to read from instruction (1) or data/unified (2) cache. All other values are reserved.
- *level* - unsigned 8-bit integer specifying which cache within the cache hierarchy to read. This value must be in the range from 0 up to one less than the *cache_levels* return value from PAL_CACHE_SUMMARY.
- *way* - unsigned 8-bit integer denoting within which cache way to read. If the cache is direct-mapped this argument is ignored.
- *part* - unsigned 8-bit integer denoting which portion of the specified cache line to read:

Value	Description
0	data
1	tag
2	data protection bits
3	tag protection bits
4	combined protection bits for data and tags[a]

[a] Note that for this *part* no data is returned. Only protection bits are returned.

All other values of *part* are reserved.

The *data* return value contains the value read from the cache. Its contents are interpreted according to the *line_id.part* field as follows:

Part	Data
0	64-bit data.
1	right-justified tag of the specified line.
2	right-justified protection bits corresponding to the 64 bits of data at *address*. If the cache uses less than 64-bits of data to generate protection, *data* will contain more than one value. For example if a cache generates parity for every 8-bits of data, this return value would contain 8 parity values. The PAL_CACHE_PROT_INFO call returns information on how a cache generates protection information in order to decode this return value. If a cache uses greater than 64-bits of data to generate protection, *data* will contain the value to use for the portion of the cache line indicated by *address*.
3	right-justified protection bits for the cache line tag.
4	right-justified protection bits for the cache line tag and 64 bits of data at *address*.

The *length* return value contains the number of valid bits returned in *data.*

The *mesi* return value contains the status bits of the cache line. Values are defined as follows:

Value	Description
0	invalid
1	shared
2	exclusive
3	modified

All other values of *mesi* are reserved.

To guarantee correct behavior for this procedure, it is required that there shall be no RSE activity that may cause cache side effects.

Get Cache Hierarchy Summary

Purpose: Returns summary information about the hierarchy of caches controlled by the processor.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_CACHE_SUMMARY within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_SUMMARY procedure.
cache_levels	Unsigned 64-bit integer denoting the number of levels of cache implemented by the processor. Strictly, this is the number of levels for which the cache controller is integrated into the processor (the cache SRAMs may be external to the processor).
unique_caches	Unsigned 64-bit integer denoting the number of unique caches implemented by the processor. This has a maximum of 2**cache_levels,* but may be less if any of the levels in the cache hierarchy are unified caches or do not have both instruction and data caches.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: Software is expected to call PAL_CACHE_SUMMARY before calling PAL_CACHE_INFO to determine the number of times PAL_CACHE_INFO should be called and the amount of storage that must be allocated to hold all of the information returned by PAL_CACHE_INFO.

Write Values into the Processor Cache

Purpose: Writes the data and tag of a processor-controlled cache line for diagnostic testing.

Calling Conv: Stacked Registers

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_CACHE_WRITE within the list of PAL procedures.
line_id	8-byte formatted value describing where in the cache to write the data.
address	64-bit 8-byte aligned physical address at which the data should be written. The address must be an implemented physical address on the processor model with bit 63 set to 0.
data	unsigned 64-bit integer value to write into the specified *part* of the cache.

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_WRITE procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error.
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error.
-7	The operation requested is not supported for this *cache_type* and *level.*

Description: The value of *data* is written into the specified level, way, and part of the cache. This procedure allows writing cache data, tag, protection, or status bits.

This procedure may also be used to seed errors into a cache line. It calculates the protection bits based on the value of *data*, then inverts a specified bit field before writing *data* to the cache. Bit field inversion is only used for writes to the cache data or tag.

If seeding an error into the instruction cache or seeding an unrecoverable error, then return back to the caller may not be possible.

This procedure call cannot be used where coherency is required.

The *line_id* argument is an 8-byte quantity in the following format:

Figure 11-25. Layout of *line_id* Return Value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
part								way								level								cache_type							

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
trigger								length								start								mesi							

- *cache_type* - unsigned 8-bit integer denoting whether to write to instruction (1) or data/unified (2) cache. All other values are reserved.
- *level* - unsigned 8-bit integer specifying which cache within the cache hierarchy to write *data*. This value must be in the range from 0 up to one less than the *cache_levels* return value from PAL_CACHE_SUMMARY.
- *way* - unsigned 8-bit integer denoting within which cache way to write *data*. If the cache is direct-mapped this argument is ignored.
- *part* - unsigned 8-bit integer denoting where to write *data* into the cache:

Value	Description
0	data
1	tag
2	data protection
3	tag protection
4	combined data and tag protection

All other values of *part* are reserved.

- *mesi* - unsigned 8-bit integer denoting whether the line should be written as clean or dirty, shared or exclusive. Though there may be multiple calls to PAL_CACHE_WRITE to the same cache line, the last call's *mesi* will be in effect. Values are defined as follows:

Value	Description
0	invalid
1	shared
2	exclusive
3	modified

All other values of *mesi* are reserved.

- *start* - unsigned 8-bit integer denoting the least-significant bit of the field in *data* to invert. If *length* is 0 or *part* is not 0 or 1, this field is ignored.
- *length* - unsigned 8-bit integer denoting the number of bits to invert. If *length* is 0, no bits are inverted and *start* is ignored. If *part* is not 0 or 1, this field is ignored.
- *trigger* - unsigned 8-bit integer denoting whether to trigger the error while in procedure. If *trigger* is 0, the procedure writes *data* and returns. If *trigger* is 1 and *cache_type* is data/unified, the procedure writes *data* and executes a 64-bit load from *address* before returning. If *trigger* is 1 and *cache_type* is set to instruction, the procedure writes *data* and branches to the *address*. All other values are reserved.

The *data* argument contains the value to write into the cache. Its contents are interpreted based on the *part* field as follows:

Part	Data
0	64-bit data to write to the specified line (with optional bit field inversion).
1	right-justified tag to write into the specified line (with optional bit field inversion).
2	right-justified protection bits corresponding to the 64 bits of data at *address*. If the cache uses less than 64-bits of data to generate protection, *data* will contain more than one value. For example if a cache generates parity for every 8-bits of data, this return value would contain 8 parity values. The PAL_CACHE_PROT_INFO call returns information on how a cache generates protection information in order to decode this return value. If a cache uses greater than 64-bits of data to generate protection, *data* will contain the value to use for the portion of the cache line indicated by *address*.
3	right-justified protection bits for the cache line tag.
4	right-justified protection bits for the cache line tag and 64 bits of data at *address*.

To guarantee correct behavior for this procedure, it is required that there shall be no RSE activity that may cause cache side effects.

Return Parameters to Copy PAL Code to Memory

Purpose: Returns the parameters needed to copy relocatable PAL code from the firmware address space to memory.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_COPY_INFO within the list of PAL procedures.
copy_type	Unsigned integer denoting type of procedures for which copy information is requested.
platform_info	8-byte formatted value describing the number of processors and the number of interrupt controllers currently enabled on the system.
mca_proc_state_info	Unsigned integer denoting the number of bytes that SAL needs for the min-state save area for each processor.

Returns:

Return Value	Description
status	Return status of the PAL_COPY_INFO procedure.
buffer_size	Unsigned integer denoting the number of bytes of PAL information that must be copied to main memory.
buffer_align	Unsigned integer denoting the starting alignment of the data to be copied.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This procedure is called to obtain the information needed to relocate runtime PAL procedures, PAL PMI code, and PAL code needed to support IA-32 operating systems from the firmware address space to memory. The information returned in this call is used by SAL to allocate a memory region on the required alignment, and call PAL_COPY_PAL to copy the relocatable PAL code.

The *copy_type* input argument indicates which type of procedure for which copying information is requested. A value of 0 denotes procedures required for SAL, PMI, and Itanium-based operating systems. A value of 1 denotes procedures required for IA-32 operating systems. All other values are reserved. If the copy_type is 0, then SAL shall call PAL_COPY_PAL call subsequently to copy the PAL procedures and PAL PMI code to the allocated memory region. If the copy_type is 1, SAL shall pass the allocated memory size and start address through the PAL_ENTER_IA_32_ENV call before booting an IA-32 OS.

The *platform_info* input argument is required only when *copy_typ*e = 1. If *copy_type* = 0, *platform_info* should be 0. *Platform_info* has the following format.

Figure 11-26. Layout of *platform_info* Input Parameter

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">num_iopics</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="32">num_procs</td></tr>
</table>

- *num_iopics* is the number of interrupt controllers currently enabled on the system.
- *num_procs* is the number of processors currently enabled on the system.

The *buffer_align* return value must be a power of two between 4 KB and 256 KB.

Copy PAL Code to Memory

Purpose: Copy relocatable PAL code from the firmware address space to memory.

Calling Conv: Stacked Registers

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_COPY_PAL within the list of PAL procedures.
target_addr	Physical address of a memory buffer to copy relocatable PAL procedures and PAL PMI code.
alloc_size	Unsigned integer denoting the size of the buffer passed by SAL for the copy operation.
processor	Unsigned integer denoting whether the call is being made on the boot processor or an application processor

Returns:

Return Value	Description
status	Return status of the PAL_COPY_PAL procedure.
proc_offset	Unsigned integer denoting the offset of PAL_PROC in the relocatable segment copied.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This procedure is called to relocate runtime PAL procedures and PAL PMI code from the firmware address space to main memory. This procedure also updates the PALE_PMI entrypoint in hardware. If the call is made on an application processor the copy is not performed. The *processor* argument denotes whether the call is being made on the boot processor (value of 0) or an application processor (value of 1). All other values are reserved.

PAL_COPY_INFO should be called first to determine the size and alignment requirements of the memory buffer to which the PAL code will be copied. Bit 63 of *target_addr* must be set consistently with the cacheability attribute of the memory buffer being copied to. It is PAL's responsibility to ensure that the firmware address space contents that are being copied from, are not in any processor caches. It is the caller's responsibility to ensure that the contents of the memory buffer copied to, are flushed out of the internal processor's data caches if *target_addr* has a cacheable memory attribute.

If a PAL procedure makes calls to internal PAL functions that execute only out of the firmware address space, that portion of code will continue to execute out of the firmware address space, even though the main procedure has been copied to RAM. This is true only for some PAL procedures that can be called only in physical mode.

PAL_COPY_PAL call is mandatory as part of the system boot process. Higher level firmware should guarantee that PAL_COPY_PAL is called on all processors before OS launch. This is to guarantee that full processor functionality is available. This procedure can be called more than once.

Get Debug Registers Information

Purpose: Returns the number of instruction and data debug register pairs.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Arguments:

Argument	Description
index	Index of PAL_DEBUG_INFO within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_DEBUG_INFO procedure.
i_regs	Unsigned 64-bit integer denoting the number of pairs of instruction debug registers implemented by the processor.
d_regs	Unsigned 64-bit integer denoting the number of pairs of data debug registers implemented by the processor.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This call returns the number of pairs of registers. Even numbered registers contain breakpoint addresses and odd numbered registers contain breakpoint mask conditions. For example if *i_regs* is 4, there are 8 instruction debug registers of which 4 are breakpoint address registers ($IBR_{0,2,4,6}$) and 4 are breakpoint mask registers ($IBR_{1,3,5,7}$). The minimum value for both *i_regs* and *d_regs* is 4.

On some implementations, a hardware debugger may use two or more debug register pairs for its own use. When a hardware debugger is attached, PAL_DEBUG_INFO may return a value for *i_regs* and/or *d_regs* less than the implemented number of debug registers. When a hardware debugger is attached, PAL_DEBUG_INFO may return a minimum value of 2 for *d_regs* and a minimum of 2 for *i_regs*.

Enter IA-32 System Environment

Purpose: This call configures the processor for execution of an IA-32 operating system and switches from the Itanium System Environment to the IA-32 System Environment.

Calling Conv: Static Registers Only*

Note: Since this is a special call, it does not follow the PAL static register calling convention. GR28 contains the index of PAL_ENTER_IA_32_ENV within the list of PAL procedures. All other input arguments including GR29-GR31 are setup by SAL to values as required by the IA-32 operating system defined in Table 11-33. The registers that are designated as preserved, scratch, input arguments and procedure return values by the static procedure calling convention are not followed by this call. For instance, GR5 and GR6 need not be preserved since these are regarded as scratch by the IA-32 operating system.

Note: In an MP system, this call must be COMPLETED on the first CPU to enter the IA-32 system environment (may or may not be the BSP) prior to being called on the remaining processors in the MP system.

Mode: Physical

Arguments: GR28 contains the index of the PAL_ENTER_IA_32_ENV call within the list of PAL procedures. All other input arguments are defined in Table 11-33.

Returns: This procedure continues to execute indefinitely in the IA-32 System Environment until power down, reset, an error condition, or a `jmpe` instruction is executed at privilege level 0. In case of an error condition or `jmpe`, the procedure transitions the processor back to Itanium System Environment and continues execution at the physical Itanium termination IP specified in GR3 by SAL as defined in Table 11-33. The register state at the physical Itanium termination IP is defined in Table 11-37.

Status: The status is returned in GR4 as defined in Table 11-37.

Description: This PAL firmware call configures the processor for execution of an IA-32 operating system and switches from the Itanium System Environment to the IA-32 System Environment.

Any required PAL firmware for supporting IA-32 operating systems is copied to the memory buffer pointed to by GR36. Firmware then configures the processor for execution in the IA-32 System Environment. This includes:

- Purging the TLB of all entries (both TRs and TCs)
- Programming all Itanium resources - general registers, floating-point registers, predicate, branch, RSE registers (RSC, BSP, BSPSTORE, RNAT), CCV, UNAT, FPSR, PFS, LC, EC, GPTA, ITM, TPR, RR and PKR, IBR, DBR, PMC, PMD registers to a state consistent with IA-32 System Environment.

The configuration of this state is implementation specific, based on implemented Itanium resources.

This PAL firmware call registers with SAL “call back” points for the following system related interrupts that may occur during the execution of the IA-32 system environment: OS_MCA and OS_INIT. SAL code MUST pass these events back through the “call back” points when these platform related interruptions occur. The PAL firmware also registers the machine check rendezvous and wake-up mechanisms to be used during machine check processing.

The firmware then initializes the processor state as supplied in the parameter list.

The IA-32 APIC is initially hardware enabled when the IA-32 System Environment is entered. The initial state of all APIC registers is extracted from the current interruption register values.

Note: Only NMI and ExtINT pending interrupts will be delivered per the IA-32 definition. All other existing pending interrupts in IRR0-3 are discarded.

MTRR physical memory attribute values and ranges are initialized to the same physical memory values specified by the SAL System Table.

Note: When the IA-32 System Environment is terminated, the SAL System Table will not reflect changes made to the MTRR physical attribute values by IA-32 code.

The processor will begin execution at the instruction and IA-32 mode (e.g. Real Mode, Protected Mode, VM86, 16/32-bit) as defined by the entry parameters in Table 11-33.

Table 11-33 describes the Itanium register state required at entry to the IA-32 System Environment:

Table 11-33. IA-32 System Environment Entry Parameters

Intel® Itanium™ Register	IA-32 State	Description
GR2{31:0}	ip	First IA-32 instruction set address. IA-32 physical address or virtual address if CR0.pg is 1. The upper 32-bits are ignored.
GR3		Termination IP. On termination of the IA-32 System Environment due to `jmpe` at ring 0 or an error condition, execution of Intel® Itanium™ instructions will continue at this 64-bit physical address. GR4 indicates the reason for termination.
GR4		Configuration Flags - flag{0} - if 1 indicates this call is being performed on the Boot Strap Processor (BSP), if 0 this call is being performed on a processor other than the BSP. flag{4:1} - Indicates the entry order in which the processor has been called to enter the IA-32 system environment. If first processor, the value will be zero; if second, the value will be one; and so on. ***Warning: If this flag value is incorrectly specified, the system may crash. Also, this value must be unique on each processor in an MP system.*** flag{63:5} - Reserved.
GR5-6	ignored	Ignored
GR7	fsd	Initial state of the IA-32 fs segment descriptor
GR8-15{31:0}	eax, ecx, edx, ebx, esp, ebp, esi, edi	Initial 32-bit state of all general purpose registers
GR16-17	gs, fs, es, ds, tr, ldt, ss, cs	Initial state of all IA-32 segment selectors
GR24,27	esd, dsd	Initial state of the IA-32 es and ds segment descriptors.
GR28	PAL index	PAL_ENTER_IA_32_ENV index value
GR29-GR31	gsd, ldtd, gdtd	Initial state of the IA-32 gs, ldt, and gdt segment descriptors.
AR25,26	csd, ssd	Initial state of the IA-32 cs and ss segment descriptors.
GR32	MP_Info_Table: Physical address of the MP Information Table described in Table 11-34 below.	
GR33	System_Table: Physical address of the SAL System Table. See the *SAL Specification* for details. The System Table defines the physical layout of the I/O Port Space, memory, and all physical memory attributes required for each section of physical memory. The System Table also defines regions of regular memory, I/O areas and where existing firmware resides. This information is used to initialize the IA-32 System Environment's MTRRs.	
GR34	Reserved	
GR35	Reserved	

Table 11-33. IA-32 System Environment Entry Parameters (Continued)

Intel® Itanium™ Register	IA-32 State	Description
GR36	MEMORY_BUFFER: Physical address of the buffer allocated for copying the PAL procedures to support IA-32 operating systems. Refer to PAL_COPY_INFO for details.	
GR37	MEMORY_BUFFER_LEN: Unsigned 64-bit integer containing the size of the buffer allocated for copying the PAL procedures to support IA-32 operating systems. Refer to PAL_COPY_INFO for details.	
GR38	mca_proc_state_info	This is the value that results from calling the SAL_GET_STATE_INFO_SIZE procedure with the arguments of mca and proc.
GR39	SAL_IO_Intercept_Function: Physical address of the SAL I/O Intercept callback function.	
GR40	SAL_IO_Intercept_Table: Physical address of the SAL I/O Intercept Table described in Table 11-35 below.	
FR8-15	fp0-7,mm0-7	Initial IA-32 FP, Intel® MMX™ technology register values
FR16-31	xmm0-7	Initial IA-32 Streaming SIMD Extension register state
AR21 (fcr)	fcw, mxcsr	Initial IA-32 numeric and Streaming SIMD Extension control values
AR24 (eflag)	eflags	Initial state of IA-32 flags
AR27 (cflg)	cr0/cr4	Initial values for CR0 and CR4
AR28 (fsr)	fsw, ftw, mxcsr	Initial IA-32 numeric and Streaming SIMD Extension status values
AR29 (fir)	fip, fcs, fop	Initial IA-32 numeric environment opcode, selector, and IP
AR30 (fdr)	fea, fds	Initial IA-32 numeric environment data selector and offset
KR1	tssd	Initial value for IA-32 TSSD
KR2	cr3/cr2	Initial values for CR3 and CR2
KR3	idtd	Initial value for IA-32 IDTD
CR9	cr0/cr4	Initial values for CR0 and CR4

PSR	--	PSR.ic =0, interrupt collection off
		PSR.i = 0, interrupts off
		PSR.it, PSR.dt, PSR.rt = 0[a]
		PSR.mc = 0, machine checks un-masked
		PSR.bn = 1, register bank 1 selected
		all other bits must be zero

DCR	--	All bits must be zero
PTA, GPTA	--	PTA.ve = 0, GPTA.ve=0, VHPT disabled
LID	--	Unique processor ID, EID address for this processor
ITC	tsc	ITC = time stamp counter

[a] virtual translations are off, ALL translations in the TRs and TCs will be ignored and invalidated

Table 11-34 describes the MP Information Table:

Table 11-34. MP Information Table

Offset (in bytes)	Length (in bytes)	Description
0	8	Address of Local APIC for use by IA-32 operating systems[a]
8	4	Number of I/O SAPICs on the system.
12	4	Number of processors on the system
16	7	Reserved (must be zero)
23	1	Checksum. This modulo sum of all the bytes in this table, including Checksum and Reserved bytes must add up to zero.
24	16	A 16-byte entry for each I/O SAPIC on the system containing the following information: Byte 0: • bits 0-3: I/O APIC ID of the I/O SAPIC for use by IA-32 operating systems[b] • bits 4-7: Must be zero Byte 1: • bit 0: 1 if the I/O SAPIC is enabled • bits 1-7: Must be 0 Bytes 2-7: Reserved Bytes 8-15: • Address of I/O APIC for use by IA-32 operating systems[a]
24+(16 * Number of I/O SAPICs)	8	A 8-byte entry for each processor on the system containing the following information: Byte 0: EID of the processor[c] Byte 1: ID of the processor[c] Byte 2: • bits 0-3: Local APIC ID of the processor for use by IA-32 operating systems[b] • bits 4-7: Must be zero Byte 3: • bit 0: 1 if the processor is enabled • bits 1-7: Must be 0 Bytes 4-7: Reserved

[a] SAL must ensure that this address does not conflict with other device addresses on the platform.

[b] SAL must generate a unique ID value and store the same ID in the MP table, for use by IA-32 operating systems. This must by the physical ID.

[c] This is the value set by SAL in the LID register of the processor (CR64)

Table 11-35 describes the SAL I/O Intercept Table. This table must be 8-byte aligned, with a minimum size of 8 bytes and a maximum size of 128 bytes. Also, the memory allocated for this table must be allocated in multiples of 8 bytes.

Table 11-35. SAL I/O Intercept Table

Offset (in bytes)	Length (in bytes)	Description
0	2	Number of IO Ports to be intercepted. This value must be between 0 and 63 inclusively.
2	2	A 2-byte entry for each intercepting port, specifying the intercepting port number. This word is little endian.
2+(2*Number of Intercepting Ports)	6 - (Number of intercepting Ports[1:0] * 2)	Reserved. This ensures that the table is a multiple of 8 bytes long.

Table 11-36 describes the IA-32 resource state set at entry to the IA-32 System Environment.

Note: SAL must initialize all the IA-32 resources to a known state, otherwise these resources may contain reset values based on the Itanium architecture and the IA-32 operating system and applications may not function properly.

Table 11-36. IA-32 Resources at IA-32 System Environment Entry

IA-32 Resource	Initial State
eflags	= AR24
eax-edi	= GR8-15{31:0}
cs:eip	= AR25:GR2
cr0, cr4	= AR27
cr2, cr3	= KR2

es, cs, ss, ds, fs, gs, ldt, tr	selector = GR16-17{63:0}
	descriptor = GR24,AR25,AR26,GR27-31{63:0}

Descriptor values for gs, fs, es, ds, ldt, gdt, ss, cs	=GR29,GR28,GR24,GR27,GR30,GR31,AR26,AR25{63:0}
idt	descriptor = KR3
fp st0-7, mm0-7	= FR8-15
xmm0-7	= FR16-31
fcw, mxcsr(control)	= fcr
fsw, mxcsr(status), ftw	= fsr
fop, fip, fcs	= fir
fea, fds	= fdr
dr0-3	= 0x0000, disabled debug registers
dr6	= 0xFFFF0FF0, disabled debug registers
dr7	= 0x00000400
TSC	= equal to interval timer (ITC)
Perf Monitors	= cleared
TLBs	= flushed
MCHK registers	= cleared
MTRRs	= MTRRs of IA-32 state are initialized to be consistent with the memory entries of the SAL System Table.
APIC	= disabled, initial support is for Intel 8259A compatible external interrupt controller

All other register values are ignored on input and may be modified by processor/firmware during execution within the IA-32 System Environment.

During the execution of the IA-32 System Environment, platform events for PAL_MCA, PAL_INIT, PAL_RESET and PAL_PMI will interrupt the IA-32 System Environment and vector to PAL firmware.

Execution continues indefinitely in the IA-32 System Environment until power down, an error condition occurs or until a `jmpe` instruction is executed at privilege level 0.

The state of all Itanium registers are left in an undefined state, code can only rely on the register state defined in Table 11-37 following termination. Allocated memory may be reclaimed by SAL or the Itanium-based OS.

When the IA-32 System Environment is terminated, the SAL System Table will not reflect changes made to the memory attribute values by IA-32 code.

Current pending interrupts are left pending.

When the IA-32 system mode is terminated, the auxiliary processors (APs) will exit the IA-32 system environment first, followed by the boot-strap processor (BSP). Upon termination, the APs will start execution in the Itanium instruction set at the termination address specified by the caller. The BSP will then start executing at the termination IP address after all of the APs have exited the IA-32 system environment. The SAL code at the termination address must ensure synchronization of all the processors in an MP system and then continue with the OEM dictated procedure.

Table 11-37 describes the Itanium register values at IA-32 System Environment termination:

Table 11-37. Register Values at IA-32 System Environment Termination

Intel® Itanium™ Register	IA-32 State	Description
GR1		Undefined
GR2	ip	Address of the IA-32 JMPE instruction that caused termination. IA-32 physical address or virtual address if CR0.pg is 1.
GR3		Number of processors that exited the IA-32 system environment.

Table 11-37. Register Values at IA-32 System Environment Termination (Continued)

Intel® Itanium™ Register	IA-32 State	Description
GR4		IA-32 System Environment Termination Reason: -1 Un-implemented procedure 0 JMPE detected at privilege level 0 1 SAL allocated buffer for IA-32 System Environment operation is too small 2 IA-32 Firmware Checksum Error 3 SAL allocated buffer for IA-32 system environment operation is not properly aligned 4 Error in SAL MP Info Table 5 Error in SAL Memory Descriptor Table 6 Error in SAL System Table 7 Inconsistent IA-32 state 8 IA-32 Firmware Internal Error 9 IA-32 Soft Reset (**Note**: remaining register state is undefined for this termination reason) 10 Machine Check Error 11 Error in SAL I/O Intercept Table 12 Processor exit due to other processor in MP system terminating the IA32 system environment. (**Note**: remaining register state is undefined for this termination reason.) 13 Itanium™-based state corruption by either SAL PMI handler or I/O Intercept callback function.
GR5-6		Undefined
GR7	apic id	The defined apic id for the processor from the apic lid register
GR8-15{31:0}	eax, ecx, edx, ebx, esp, ebp, esi, edi	Final 32-bit state of all general purpose registers
GR16-17	es, cs, ss, ds, fs, gs, ldt, tr	Final state of all IA-32 segment selectors (bank 1)
GR24,AR25, AR26, GR27-31	esd, csd, ssd, dsd, fsd, gsd, ldtd, gdtd	Final state of all IA-32 segment descriptors (bank 1)
GR18-23,25-26, 32-127		Undefined (bank 1)
GR16-31		Bank Register 0 - Undefined
FR8-15	fp0-7, mm0-7	Final IA-32 FP, Intel® MMX™ technology register values
FR16-31	xmm0-7	Final IA-32 Streaming SIMD Extension register values
FR2-7,32-127		Undefined
PR0-63		
BR0-7		
RSC, BSP, BSPSTORE, RNAT, CCV, UNAT, FPSR, PFS, LC, EC		
AR21 (fcr)	fcw, mxcsr	Final IA-32 numeric and Streaming SIMD Extension control values
AR24 (eflag)	eflags	Final state of IA-32 flags
AR27 (cflg)	cr0/cr4	Final values for CR0 and CR4

Table 11-37. Register Values at IA-32 System Environment Termination (Continued)

Intel® Itanium™ Register	IA-32 State	Description
AR28 (fsr)	fsw, ftw, mxcsr	Final IA-32 numeric and Streaming SIMD Extension values
AR29 (fir)	fip, fcs. fop	Final IA-32 numeric environment opcode, selector, and IP
AR30 (fdr)	fea, fds	Final IA-32 numeric environment data selector and offset
KR1	tssd	Final value for IA-32 TSSD
KR2	cr3/cr2	Final values for CR3 and CR2
KR3	idtd	Final value for IA-32 IDTD
KR0,4-7		Undefined

PSR	--	PSR.ic =0, interrupt collection off
		PSR.i = 0, interrupts off
		PSR.it, PSR.dt, PSR.rt = 0[a]
		PSR.mc = 0, machine checks un-masked
		PSR.bn = 1, register bank 1 selected
		all other bits are 0

DCR	--	Zeros
PTA	--	PTA.ve= 0, VHPT is disabled
GPTA	--	GPTA.ve = 0
LID	--	Received unique ID, EID value for this processor
ITC	tsc	ITC = final time stamp counter value
IFA, IIP, IPSR, ISR, IIM, IIPA, ITTR, IHA, IFS, IVA, GPTA, ITM, IVR, TPR, IRR0-3, ITV, PMV, LRR0, LRR1, CMCV		Undefined
TRs, TCs (TLBs)		Undefined
RR		
PKR		
IBR, DBR		
PMC, PMD		

[a] Virtual translations are off, ALL original translations in the TRs and TCs have been invalidated

Get Fixed Geographical Address of Processor

Purpose: Returns a unique geographical address of this processor on its bus.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Arguments:

Argument	Description
index	Index of PAL_FIXED_ADDR call within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_FIXED_ADDR procedure.
address	Fixed geographical address of this processor.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: The *address* return value will contain a unique unsigned integer denoting the position of this processor on the current bus. This is an arbitrary number which is expected to have geographical significance and be unique for the bus to which the processor is connected. If the processor is connected to multiple busses, the *address* return value must be unique among all such busses. For each implementation, the value should be the smallest unique value that can be returned on that implementation. For example, on a bus which could support 6 processors, the *address* return value should occupy no more than 3 bits. In any case, it will never be more than 16 bits.

Get Processor Base Frequency

Purpose: Returns the frequency of the output clock for use by the platform is generated by the processor.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Arguments:

Argument	Description
index	Index of PAL_FREQ_BASE within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_FREQ_BASE procedure.
base_freq	Base frequency of the platform if generated by the processor chip.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Can not complete call without error

Description: If the processor outputs a clock for use by the platform, the *base_freq* return parameter will be the frequency of this output clock in ticks per second. If the processor does not generate an output clock for use by the platform, this procedure will return with a status of -1.

Get Processor Frequency Ratios

Purpose: Returns the ratios of the processor frequency, bus frequency, and interval timer to the input clock of the processor, if the platform clock is generated externally or to the output clock to the platform, if the platform clock is generated by the processor.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Arguments:

Argument	Description
index	Index of PAL_FREQ_RATIOS within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_FREQ_RATIOS procedure.
proc_ratio	Ratio of the processor frequency to the input clock of the processor, if the platform clock is generated externally or to the output clock to the platform, if the platform clock is generated by the processor.
bus_ratio	Ratio of the bus frequency to the input clock of the processor, if the platform clock is generated externally or to the output clock to the platform, if the platform clock is generated by the processor.
itc_ratio	Ratio of the interval timer counter rate to input clock of the processor, if the platform clock is generated externally or to the output clock to the platform, if the platform clock is generated by the processor.

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Can not complete call without error

Description: Each of the ratios returns is an unsigned 64-bit value, where the upper 32 bits contain the numerator and the lower 32 bits contain the denominator of the ratio.

Halt Processor

Purpose: Causes the processor to enter the HALT state, or one of the implementation-dependent low-power states.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_HALT within the list of PAL procedures.
halt_state	Unsigned 64-bit integer denoting low power state requested.
io_detail_ptr	8-byte aligned physical address pointer to information on the type of I/O (load/store) requested.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_HALT procedure.
load_return	Value returned if a load instruction is requested in the *io_detail_ptr*
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: This call places the processor in a low power state designated by *halt_state.* This procedure can optionally let the platform know it is about to enter the low power state via an I/O transaction.

halt_state is an unsigned 64-bit integer denoting the low power state requested. The value passed must be a valid halt state in the range from 1 to 7, for which information is returned by PAL_HALT_INFO. All other values are reserved.

The processor informs the platform that it has entered the requested low-power state in an implementation-specific manner.

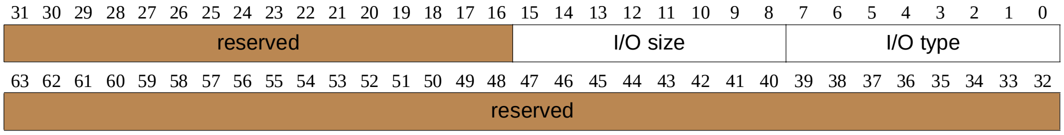
The layout of the information pointed to by the *io_detail_ptr* is shown Table 11-38.

Table 11-38. I/O Detail Pointer Description

Offset	Description
0x0	I/O size and type information
0x8	Address for I/O
0x10	Data value to store

- I/O size and type information has the format shown in Figure 11-27.

Figure 11-27. I/O Size and Type Information Layout



- *I/O type* is an unsigned 8-bit integer denoting the type of I/O transaction to complete.

Table 11-39. I/O Type Definition

Value	Description
0	No transaction
1	Perform a load
2	Perform a store

All other values for *I/O type* are reserved.

- *I/O size* is an unsigned 8-bit integer denoting the size of the I/O transaction to complete.

Table 11-40. I/O Size Definition

Value	Description
0	No transaction
1	1 byte size
2	2 byte size
4	4 byte size
8	8 byte size

All other values for *I/O size* are reserved.

- Address for the I/O transaction is a physical pointer for the load or store. The address passed should be aligned according to the size of the I/O transaction requested. The most significant bit (63) of the physical address should be set according to the cacheability attribute wanted for the I/O transaction.
- The data value to store is the value that will be stored out if the *io_type* is 2. If *io_type* is not equal to a 2, then this value is a don't care.

If an I/O transaction is requested by the caller, the processor will wait until this transaction has been received by the platform before entering the low power state.

On receipt of a PMI, machine check, INIT, reset, or unmasked external interrupt (including NMI), PAL transitions the processor to the normal state. An unmasked external interrupt is defined to be an interrupt that is permitted to interrupt the processor based on the current setting of the TPR.mic and TPR.mmi fields in the TPR control register. PAL sets the value in the *load_return* return parameter if the *io_type* is 1, otherwise this value is set to zero.

If the processor transitions to normal state via an unmasked external interrupt, execution resumes to the caller.

If the processor transitions to normal state via a PMI, execution resumes to the caller if PMIs are masked, otherwise execution will resume to the PMI handler.

If the processor transitions to the normal state via a machine check or INIT, execution resumes to the caller if machine checks and INITs are masked, otherwise execution will resume to the corresponding handler.

If the processor transitions to the normal state via a reset event, the processor will reset itself and start execution at the PAL reset address.

For more information on power management, please refer to Section 11.6 on page 2:281.

Get Halt State Information for Power Management

Purpose: Returns information about the processor’s power management capabilities.

Calling Conv: Stacked Registers

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_HALT_INFO within the list of PAL procedures.
power_buffer	64-bit pointer to a 64-byte buffer aligned on an 8-byte boundary.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_HALT_INFO procedure.
Reserved	0
Reserved	0
Reserved	0

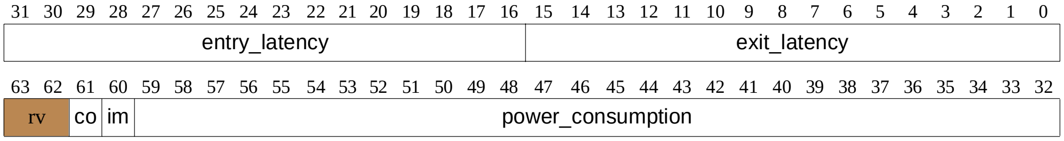
Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: The power information requested is returned in the data buffer referenced by *power_buffer*. Power information is returned about the 8 power states. The low power states are LIGHT_HALT, HALT, plus 6 other low power states. The LIGHT_HALT state is index 0 in the buffer, and the HALT state is index 1. All 8 low power states need not be implemented

The information returned is in the format of Figure 11-28. The information about the HALT states will be in ascending order of the index values.

Figure 11-28. Layout of *power_buffer* Return Value



- *exit latency* - 16-bit unsigned integer denoting the minimum number of processor cycles to transition to the NORMAL state.
- *entry_latency* - 16-bit unsigned integer denoting the minimum number of processor cycles to transition from the NORMAL state.
- *power_consumption* - 28-bit unsigned integer denoting the typical power consumption of the state, measured in milliwatts.
- *im* - 1-bit field denoting whether this low power state is implemented or not. A value of 1 indicates that the low power state is implemented, a value of 0 indicates that it is not implemented. If this value is 0 then all other fields are invalid.
- *co* - 1-bit field denoting if the low power state maintains cache and TLB coherency. A value of 1 indicates that the low power state keeps the caches and TLBs coherent, a value of 0 indicates that it does not.

The latency numbers given are the minimum number of processor cycles that will be required to transition the states. The maximum or average cannot be determined by PAL due to its dependency on outstanding bus transactions.

For more information on power management, please refer to Section 11.6 on page 2:281.

Cause Processor to Enter Coherent Halt State

Purpose: Causes the processor to enter the LIGHT HALT state, where prefetching and execution are suspended, but cache and TLB coherency is maintained.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_HALT_LIGHT within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_HALT_LIGHT procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This call places the processor in the LIGHT HALT state in an implementation-dependent fashion where cache and TLB coherency is maintained, but power consumption is minimized.

The processor acknowledges to the platform that it has entered the LIGHT HALT low-power state in an implementation-specific manner.

On receipt of a PMI, machine check, INIT, reset, or unmasked external interrupt (including NMI), PAL transitions the processor to the normal state. An unmasked external interrupt is defined to be an interrupt that is permitted to interrupt the processor based on the current setting of the TPR.mic and TPR.mmi fields in the TPR control register.

If the processor transitions to normal state via an unmasked external interrupt, execution resumes to the caller.

If the processor transitions to normal state via a PMI, execution resumes to the caller if PMIs are masked, otherwise execution will resume to the PMI handler.

If the processor transitions to the normal state via a machine check or INIT, execution resumes to the caller if machine checks and INITs are masked, otherwise execution will resume to the corresponding handler.

If the processor transitions to the normal state via a reset event, the processor will reset itself and start execution at the PAL reset address.

For more information on power management, please refer to Section 11.6 on page 2:281.

Clear Processor Error Logging Registers

Purpose: Clears all processor error logging registers and reset the indicator that allows the error logging registers to be written. This procedure also checks the pending machine check bit and pending INIT bit and reports their states.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_MC_CLEAR_LOG within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_CLEAR_LOG procedure.
pending	64-bit vector denoting whether an event is pending.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This procedure is called to clear processor error logging registers after all error information has been obtained. This procedures re-enables the logging registers in the case of a subsequent error. It clears any information that would be returned by either the PAL_MC_ERROR_INFO or PAL_MC_DYNAMIC_STATE procedures.

This procedure does not clear any pending machine checks. The *pending* return parameter returns a value of 0 if no subsequent event is pending, a 1 in bit position 0, if a machine check is pending, and/or a 1 in bit position 1 if an INIT is pending. All other values are reserved.

Figure 11-29. *Pending* Return Parameter

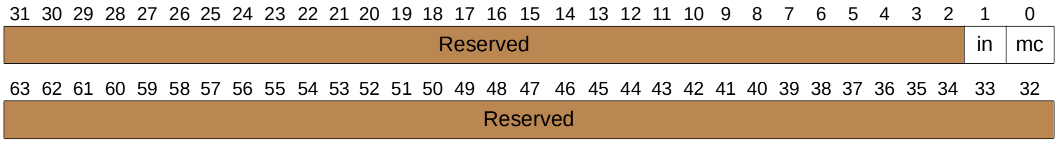


Table 11-41. Pending Return Parameter Fields

Field name	Description
mc	Pending machine check
in	Pending initialization event

Complete Outstanding Transactions

Purpose: Ensures that all outstanding transactions in a processor are completed or that any MCA due to these outstanding transactions is taken.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_MC_DRAIN within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_DRAIN procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This call causes all outstanding transactions in the processor to be completed (i.e., loads get their data returned, stores get issued to the bus, and prefetches are either completed or cancelled). As a result of completing these outstanding transactions Machine Check Aborts (MCAs) may be taken. This call is typically issued by code that needs to guarantee that no MCAs due to outstanding transactions will occur after a given point.

Returns Dynamic Processor State

Purpose: Returns the Machine Check Dynamic Processor State.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_MC_DYNAMIC_STATE within the list of PAL procedures.
offset	Offset of the next 8 bytes of Dynamic Processor State to return. (multiple of 8)
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_DYNAMIC_STATE procedure.
size	Unsigned 64-bit integer denoting bytes of Dynamic Processor State returned.
pds	Next 8 bytes of Dynamic Processor State.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: Returns the 8 bytes of Processor Dynamic State from the location specified by the *offset* argument. This data is returned in an 8-byte return values, *pds*. The *offset* argument specifies the offset from the start of the processor dependent error information area. The *size* return argument specifies the number of bytes actually returned. In order to obtain all of the error information, software must call PAL_MC_DYNAMIC_STATE with an initial *offset* value of 0, adding the *size* returned from the previous call, until it returns a Status of -2 or the *size* is equal to 0.

The Processor Dynamic State is implementation dependent.

The information returned by this procedure is cleared by PAL_MC_CLEAR_LOG.

Get Processor Error Information

Purpose: Returns the Processor Machine Check Information

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_MC_ERROR_INFO within the list of PAL procedures.
info_index	Unsigned 64-bit integer identifying the error information that is being requested. (See Table 11-42).
level_index	8-byte formatted value identifying the structure to return error information on.(See Figure 11-30).
err_type_index	Unsigned 64-bit integer denoting the type of error information that is being requested for the structure identified in *level_index*.

Returns:

Return Value	Description
status	Return status of the PAL_MC_ERROR_INFO procedure.
error_info	Error information returned. The format of this value is dependant on the input values passed.
inc_err_type	If this value is zero, all the error information specified by *err_type_index* has been returned. If this value is one, more structure specific error information is available and the caller needs to make this procedure call again with *level_index* unchanged and *err_type_index,* incremented.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-6	Argument was valid, but no error information was available

Description: This procedure returns error information for machine checks as specified by *info_index*, *level_index* and *err_type_index*. Higher level software is informed that additional machine check information is available when the processor state parameter *mi* bit is set to one. See Table 11-5, "Processor State Parameter Fields," on page 2:271 for more information on the processor state parameter and the *mi* bit description.

The *info_index* argument specifies which error information is being requested. See Table 11-42 for the definition of the *info_index* values.

Table 11-42. info_index Values

info_index	Error Information Type	Description
0	Processor Error Map	This *info_index* value will return the processor error map. This return value specifies the processor core identification, the processor thread identification, and a bit-map indicating which structure(s) of the processor generated the machine check. This bit-map has the same layout as the *level_index*. A one in the structure bit-map indicates that there is error information available for the structure. The layout of the *level_index* is described in Figure 11-30 on page 2:338.
1	Processor State Parameter	This *info_index* value will return the same processor state parameter that is passed at the PALE_CHECK exit state for a machine check event (provided a valid min-state save area has been registered) or will construct a processor state parameter for a corrected machine check events. This parameter describes the severity of the error and the validity of the processor state when the machine check or CMCI occurred. This procedure will not return a valid PSP for INIT events. The Processor State Parameter is described in Figure 11-11, "Processor State Parameter," on page 2:271.
2	Structure Specific Error Information	This *info_index* value will return error information specific to a processor structure. The structure is specified by the caller using the *level_index* and *err_type_index* input parameters. The value returned in *error_info* is specific to the structure and type of information requested.

All other values of *info_index* are reserved. When *info_index* is equal to 0 or 1, the *level_index* and *err_type_index* input values are ignored. When *info_index* is equal to 2, the *level_index* and *err_type_index* define the format of the *error_info* return value.

The caller is expected to first make this procedure call with *info_index* equal to zero to obtain the processor error map. This error map informs the caller about the processor core identification, the processor thread identification and indicates which structure(s) caused the machine check. If more than one structure generated a machine check, multiple structure bits will be set. The caller then uses this information to make sub-sequent calls to this procedure for each structure identified in the processor error map to obtain detailed error information.

The *level_index* input argument specifies which processor core, processor thread and structure for which information is being requested. See Table 11-43 on page 2:339 for the definition of the *level_index* fields. This procedure call can only return information about one processor structure at a time. The caller is responsible for ensuring that only one structure bit in the *level_index* input argument is set at a time when retrieving information, otherwise the call will return that an invalid argument was passed.

Figure 11-30. level_index Layout

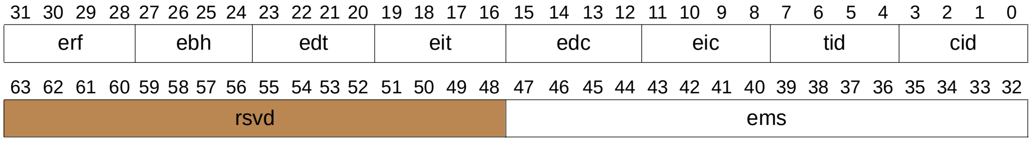


Table 11-43. level_index Fields

Field Name	Bit	Description
cid	3:0	Processor core ID (default is 0 for processors with a single core)
tid	7:4	Logical thread ID (default is 0 for processors that execute a single thread)
eic	11:8	Error information is available for 1st, 2nd, 3rd, and 4th level instruction caches
edc	15:12	Error information is available for 1st, 2nd, 3rd, and 4th level data/unified caches
eit	19:16	Error information is available for 1st, 2nd, 3rd, and 4th level instruction TLB
edt	23:20	Error information is available for 1st, 2nd, 3rd, and 4th level data/unified TLB
ebh	27:24	Error information is available for the 1st, 2nd, 3rd, and 4th level processor bus hierarchy
erf	31:28	Error information is available on register file structures
ems	47:32	Error information is available on micro-architectural structures
rsvd	63:48	Reserved

The convention for levels and hierarchy in the *level_index* field is such that the least significant bit in the error information bit-fields represent the lowest level of the structures hierarchy. For example bit 8 if the *eic* field represents the first level instruction cache.

The *erf* field is 4-bits wide to allow reporting of 4 concurrent register related machine checks at one time. One bit would be set for each error. The *ems* field is 16-bits wide to allow reporting of 16-concurrent micro-architectural structures at one time. There is no significance in the order of these bits. If only one register file related error occurred, it could be reported in any one of the 4-bits.

The *err_type_index* specifies the type of information will be returned in *error_info* for a particular structure. See Table 11-44 for the values of *err_type_index*

Table 11-44. err_type_index Values

err_type_index value mod 8	Return Value	Description
0	Structure specific error information specified by *level_index*	The information returned in *error_info* is dependant on the structure specified in *level_index*. See Table 11-45 for the error_info return formats.
1	Target address	The target address is a 64-bit integer containing the physical address where the data was to be delivered or obtained. The target address also can return the incoming address for external snoops and TLB shoot-downs that generated a machine check. The structure specific error information informs the caller if there is a valid target address to be returned for the requested structure.
2	Requester identifier	The requester identifier is a 64-bit integer that specifies the bus agent that generated the transaction responsible for generating the machine check. The structure specific error information informs the caller if there is a valid requester identifier.
3	Responder identifier	The responder identifier is a 64-bit integer that specifies the bus agent that responded to a transaction that was responsible for generating the machine check. The structure specific error information informs the caller if there is a valid responder identifier.

Table 11-44. err_type_index Values (Continued)

err_type_index value mod 8	Return Value	Description
4	Precise instruction pointer	The precise instruction pointer is a 64-bit virtual address that points to the bundle that contained the instruction responsible for the machine check. The structure specific error information informs the caller if there is a valid precise instruction pointer.
5-7	Reserved	Reserved

See Table 11-45 for the format of *error_info* when structure specific information is requested.

Table 11-45. error_info Return Format when info_index = 2 and err_type_index = 0

level_index field input	error_info return format
eic	cache_check return format
edc	cache_check return format
eit	tlb_check return format
edt	tlb_check return format
ebh	bus_check return format
erf	reg_file_check return format
ems	uarch_check return format

The structure specified by the *level_index* may have the ability to log distinct multiple errors. This can occur if the structure is accessed at the same time by more than one instruction and the processor can log machine check information for each access. To inform the caller of this occurrence, this procedure will return a value of one in the *inc_err_type* return value.

It is important to note, that when the caller sees that the *inc_err_type* return value is one, it should make a sub-sequent call with the *err_type_index* value incremented by 8. If the structure specific error information returns that there is a valid target address, requester identifier, responder identifier or precise instruction pointer these can be returned as well by incrementing the *err_type_index* value in the same manner. Refer to the following example for more information.

For example, to gather information on the first error of a structure that can log multiple errors, *err_type_index* would be called with the value of 0 first. The caller examines the information returned in *error_info* to know if there is a valid target address, requester identifier, responder identifier, or precise instruction pointer available for logging. If there is, it makes sub-sequent calls with *err_type_index* equal to 1, 2, 3 and/or 4 depending on which valid bits are set. Additionally if the *inc_err_type* return value was set to one, the caller knows that this structure logged multiple errors. To get the second error of the structure it sets the *err_type_index* = 8 and the structure specific information is returned in *error_info*. The caller examines this *error_info* to know if there is a valid target address, requester identifier, responder identifier, or precise instruction pointer available for logging on the second error. If there is, it makes sub-sequent calls with *err_type_index* equal to 9, 10, 11, and/or 12 depending on which valid bits are set. The caller continues incrementing the *err_type_index* value in this fashion until the *inc_err_type* return value is zero.

As shown in Table 11-45, the information returned in *error_info* varies based on which structure information is being requested on. The next sections describe the *error_info* return format for the different structures.

Cache_Check Return Format: The cache check return format is returned in *error_info* when the user requests information on any instruction or data/unified caches in the *level_index* input argument. The cache_check return format is a bit-field that is described in Figure 11-31 and Table 11-46.

Figure 11-31. Cache_Check Layout

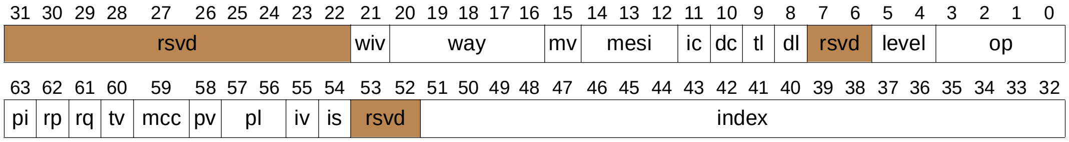


Table 11-46. Cache_Check Fields

Field name	Bits	Description
op	3:0	Type of cache operation that caused the machine check: 0 - unknown or internal error 1 - load 2 - store 3 - instruction fetch or instruction prefetch 4 - data prefetch (both hardware and software) 5 - snoop (coherency check) 6 - cast out (explicit or implicit write-back of a cache line) 7 - move in (cache line fill) All other values are reserved.
level	5:4	Level of cache where the error occurred. A value of 0 indicates the first level of cache.
rsvd	7:6	Reserved
dl	8	Failure located in the data part of the cache line.
tl	9	Failure located in the tag part of the cache line.
dc	10	Failure located in the data cache
ic	11	Failure located in the instruction cache
mesi	14:12	0 - cache line is invalid. 1 - cache line is held shared. 2 - cache line is held exclusive. 3 - cache line is modified. All other values are reserved.
mv	15	The *mesi* field in the cache_check parameter is valid.
way	20:16	Failure located in the way of the cache indicated by this value.
wiv	21	The *way* and *index* field in the cache_check parameter is valid.
rsvd	31:22	Reserved
index	51:32	Index of the cache line where the error occurred.
rsvd	53:52	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel® Itanium™ instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the cache_check parameter is valid.
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the cache_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.

Table 11-46. Cache_Check Fields (Continued)

Field name	Bits	Description
tv	60	Target address is valid: This bit is set to one to indicate that a valid target address has been logged.
rq	61	Requester identifier: This bit is set to one to indicate that a valid requester identifier has been logged.
rp	62	Responder identifier: This bit is set to one to indicate that a valid responder identifier has been logged.
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

TLB_Check Return Format: The tlb_check return format is returned in *error_info* when the user requests information on any instruction or data/unified TLB in the *level_index* input argument. The tlb_check return format is a bit-field that is described in Figure 11-32 and Table 11-47.

Figure 11-32. TLB_Check Layout

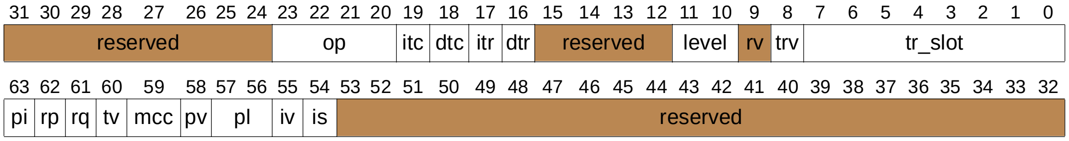


Table 11-47. TLB_Check Fields

Field name	Bits	Description
tr_slot	7:0	Slot number of the translation register where the failure occurred.
trv	8	The *tr_slot* field in the TLB_check parameter is valid.
rv	9	Reserved
level	11:10	The level of the TLB where the error occurred. A value of 0 indicates the first level of TLB
reserved	15:12	Reserved
dtr	16	Error occurred in the data translation registers
itr	17	Error occurred in the instruction translation registers
dtc	18	Error occurred in data translation cache
itc	19	Error occurred in the instruction translation cache
op	23:20	Type of cache operation that caused the machine check: 0 - unknown 1 - TLB access due to load instruction 2 - TLB access due to store instruction 3 - TLB access due to instruction fetch or instruction prefetch 4 - TLB access due to data prefetch (both hardware and software) 5 - TLB shoot down access 6 - TLB probe instruction (probe, tpa) 7 - move in (VHPT fill) All other values are reserved.
reserved	53:24	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel® Itanium™ instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the TLB_check parameter is valid.

Table 11-47. TLB_Check Fields (Continued)

Field name	Bits	Description
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the TLB_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
tv	60	Target address is valid: This bit is set to one to indicate that a valid target address has been logged.
rq	61	Requester identifier: This bit is set to one to indicate that a valid requester identifier has been logged.
rp	62	Responder identifier: This bit is set to one to indicate that a valid responder identifier has been logged.
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

Bus_Check Return Format: The bus_check return format is returned in *error_info* when the user requests information on any level of hierarchy of the processor bus structures as specified in the *level_index* input argument. The bus_check return format is a bit-field that is described in Figure 11-33 and Table 11-48.

Figure 11-33. Bus Check Layout

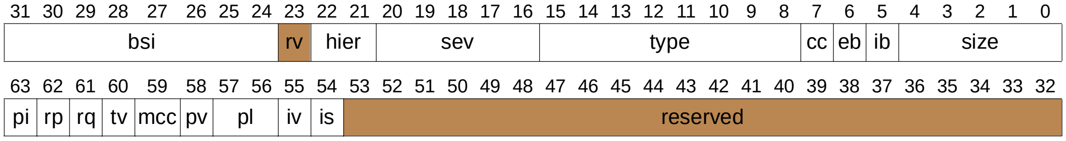


Table 11-48. Bus Check Fields

Field name	Bits	Description
size	4:0	Size in bytes of the transaction that caused the machine check abort.
ib	5	Internal bus error
eb	6	External bus error
cc	7	Error occurred during a cache to cache transfer.
type	15:8	Type of transaction that caused the machine check abort. 0 - unknown 1 - partial read 2 - partial write 3 - full line read 4 - full line write 5 - implicit or explicit write-back operation 6 - snoop probe 7 - incoming ptc.g 8 - WC transactions All other values are reserved
sev	20:16	Bus error severity. The encodings of error severity are platform specific.
hier	22:21	This value indicates which level or bus hierarchy the error occurred in. A value of 0 indicates the first level of hierarchy.
reserved	23	Reserved

Table 11-48. Bus Check Fields (Continued)

Field name	Bits	Description
bsi	31:24	Bus error status information. It describes the type of bus error. This field is processor bus specific.
reserved	53:32	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel® Itanium™ instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the bus_check parameter is valid.
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the bus_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
tv	60	Target address is valid: This bit is set to one to indicate that a valid target address has been logged.
rq	61	Requester identifier: This bit is set to one to indicate that a valid requester identifier has been logged.
rp	62	Responder identifier: This bit is set to one to indicate that a valid responder identifier has been logged.
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

Reg_File_Check Return Format: The reg_file_check return format is returned in *error_info* when the user requests information on any of the registers as specified in the *level_index* input argument. The reg_file_check return format is a bit-field that is described in Figure 11-34 and Table 11-49. When the reg_file_check return format is returned, the target address, the requester identifier and the responder identifier will always be invalid.

Figure 11-34. Reg_File_Check Layout

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15	14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
reserved	rnv	reg_num	op	id

63	62 61 60	59	58	57 56	55	54	53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
pi	rsvd	mcc	pv	pl	iv	is	reserved

Table 11-49. Reg_File_Check Fields

Field name	Bits	Description
id	3:0	Register file identifier: 0 - unknown/unclassified 1 - General register (bank1) 2 - General register (bank 0) 3- Floating-point register 4- Branch register 5- Predicate register 6- Application register 7- Control register 8- Region register 9- Protection key register 10- Data breakpoint register 11 - Instruction breakpoint register 12 - Performance monitor control register 13 - Performance monitor data register All other values are reserved
op	7:4	Identifies the operation that caused the machine check 0 - unknown 1 - read 2 - write All other values are processor specific
reg_num	14:8	Identifies the register number that was responsible for generating the machine check
rnv	15	Specifies if the *reg_num* field is valid
reserved	53:16	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel® Itanium™ instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the reg_file_check parameter is valid.
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the reg_file_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
reserved	62:60	Reserved
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

Uarch_Check Return Format: The uarch_check return format is returned in *error_info* when the user requests information on any of the micro-architectural structures as specified in the *level_index* input argument. The uarch_check return format is a bit-field that is described in Figure 11-35 and Table 11-50.

Figure 11-35. uarch_check Layout

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">reserved</td><td>xv</td><td>wv</td><td colspan="6">way</td><td colspan="4">op</td><td colspan="4">array_id</td><td colspan="3">level</td><td colspan="5">sid</td></tr>
</table>

<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td>pi</td><td>rp</td><td>rq</td><td>tv</td><td>mcc</td><td>pv</td><td colspan="2">pl</td><td>iv</td><td>is</td><td colspan="14">reserved</td><td colspan="8">index</td></tr>
</table>

Table 11-50. uarch_check Fields

Field name	Bits	Description
sid	4:0	Structure identification. These bits identify the micro-architectural structure where the error occurred. The definition of these bits are implementation specific.
level	7:5	Level of the micro-architectural structure where the error was generated. A value of 0 indicates the first level.
array_id	11:8	Identification of the array in the micro architectural structure where the error was generated. 0 - unknown/unclassified All other values are implementation specific
op	15:12	Type of operation that caused the error 0 - unknown 1 - read or load 2 - write or store All other values are implementation specific
way	21:16	Way of the micro-architectural structure where the error was located.
wv	22	The *way* field in the uarch_check parameter is valid.
xv	23	The *index* field in the uarch_check parameter is valid.
reserved	31:24	Reserved
index	39:32	Index or set of the micro-architectural structure where the error was located.
reserved	53:40	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel® Itanium™ instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the bus_check parameter is valid.
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the bus_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
tv	60	Target address is valid: This bit is set to one to indicate that a valid target address has been logged.
rq	61	Requester identifier: This bit is set to one to indicate that a valid requester identifier has been logged.
rp	62	Responder identifier: This bit is set to one to indicate that a valid responder identifier has been logged.
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

Set/Reset Expected Machine Check Indicator

Purpose: Informs PALE_CHECK whether a machine check is expected so that PALE_CHECK will not attempt to correct any expected machine checks.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_MC_EXPECTED within the list of PAL procedures.
expected	Unsigned integer with a value of 0 or 1 to set or reset the hardware resource PALE_CHECK examines for expected machine checks.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_EXPECTED procedure.
previous	Unsigned integer denoting whether a machine check was previously expected.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: If the argument *expected* contains a value of 1, an implementation-dependent hardware resource is set to inform PALE_CHECK to expect a machine check. If the argument *expected* is 0, the resource is reset, so that PALE_CHECK does not expect any following machine checks. All other values of *expected* are reserved.

The implementation-dependent hardware resource should be, by default, in the “not expected” state. Software or firmware should only call PAL_MC_EXPECTED immediately prior to issuing an instruction which might generated an expected machine check. It should then immediately reset the bit to the “not expected” state after checking the results of the operation.

The *previous* return parameter indicates the previous state of the hardware resource to inform PALE_CHECK of an expected machine check. A value of 0 indicates that a machine check was not expected. A value of 1 indicated that a machine check was expected. All other values of *previous* are reserved.

Register Memory with PAL for Machine Check and Init

Purpose: Registers a platform dependent location with PAL to which it can save minimal processor state in the event of a machine check or initialization event.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_MC_REGISTER_MEM within the list of PAL procedures.
address	Physical address of the buffer to be registered with PAL.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_REGISTER_MEM procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: PAL places the address passed in the XR0 register, which is used by PAL as the min-state save area in the event of a machine check or initialization event. The size and layout of the area referenced by the *address* parameter is defined in Section 11.3.2.3. The address must be aligned on a 512 byte boundary. The min-state save area must be in uncacheable memory.

Restore Minimal Architected State and Return

Purpose: Restores the minimal architectural processor state, sets the CMC interrupt if necessary, and resumes execution.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_MC_RESUME within the list of PAL procedures.
set_cmci	Unsigned 64 bit integer denoting whether to set the CMC interrupt. A value of 0 indicates not to set the interrupt, a value of 1 indicated to set the interrupt, and all other values are reserved.
save_ptr	Physical address of min-state save area used to used to restore processor state.
new_context	Unsigned 64-bit integer denoting whether the caller is returning to a new context. A value of 0 indicates the caller is returning to the interrupted context, a value of 1 indicates that the caller is returning to a new context.

Returns:

Return Value	Description
status	Return status of the PAL_MC_RESUME procedure[a].
Reserved	0
Reserved	0
Reserved	0

[a] This procedure returns to the caller only in an error situation.

Status:

Status Value	Description
-2	Invalid argument
-3	Call completed with error

Description: This procedure will restore the processor minimal architected state and optionally set the CMC interrupt.

If the *set_cmci* argument is set to one, this procedure will set the CMC interrupt and return to the interrupted context. The CMC interrupt handler will be invoked sometime after returning to the interrupted context.

The *save_ptr* argument specifies the processor min-state save area buffer from which the processor state will be restored. This pointer has the same alignment and size restrictions as the address passed to PAL_MC_REGISTER_MEM procedure on page 2:348.

This procedure is used to resume execution of the interrupted context for both machine check and initialization events. This procedure can resume execution to the same context or a new context. If software attempts to resume execution for these events without using this call, processor behavior is undefined.

If the caller is resuming to the same context, the *new_context* argument must be set to 0 and the *save_ptr* argument has to point to a copy of the min-state save area written by PAL when the event occurred.

If the caller is resuming to a new context, the new_context argument must be set to 1 and the save_ptr argument must point to a new min-state save area set up by the caller.

Please see Section 11.3.3 on page 2:276 3for more information on resuming to the interrupted context.

Get Memory Attributes

Purpose: Returns the memory attributes implemented by processor.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Arguments:

Argument	Description
index	Index of PAL_MEM_ATTRIB within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

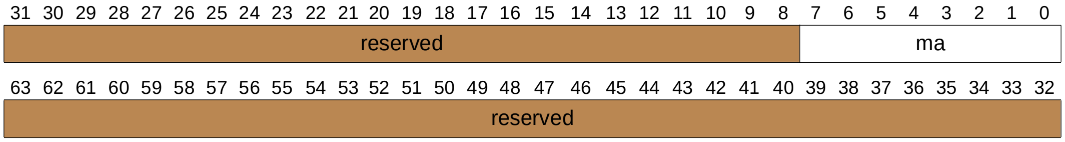
Return Value	Description
status	Return status of the PAL_MEM_ATTRIB procedure.
attrib	8-bit vector of memory attributes implemented by processor.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: Returns a 8-bit vector in the low order 8 bits of the return register that specifies the set of memory attributes implemented by the processor. The return register is formatted as follows:

Figure 11-36. Layout of *attrib* Return Value



Each bit in the bit field *ma* represents one of the eight possible memory attributes implemented by the processor. The bit field position corresponds to the numeric memory attribute encoding defined in “Memory Attributes”.

Get Processor Performance Monitor Information

Purpose: Returns Performance Monitor information about what can be counted and how to configure the monitors to count the desired events.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_PERF_MON_INFO within the list of PAL procedures.
pm_buffer	An address to an 8-byte aligned 128-byte memory buffer.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PERF_MON_INFO procedure.
pm_info	Information about the performance monitors implemented.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: PAL_PERF_MON_INFO is called to determine the number of performance monitors and the events which can be counted on the performance monitors. For more information on performance monitoring, see "Performance Monitoring". *pm_info* is a formatted 64-bit return register, as shown in Figure 11-37.

.

Figure 11-37. Layout of *PM_info* Return Value

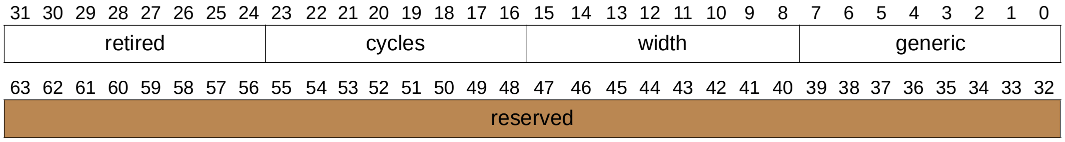


Table 11-51. *PM_info* Fields

Field name	Description
generic	Unsigned 8-bit number defining the number of generic PMC/PMD pairs.
width	Unsigned 8-bit number in the range 0:60 defining the number of implemented counter bits.
cycles	Unsigned 8-bit number defining the event type for counting processor cycles.
retired	Unsigned 8-bit number defining the event type for retired instruction bundles.

The *pm_buffer* argument points to a 128-byte memory area where mask information is returned. The layout of *pm_buffer* is shown in Table 11-52.

Table 11-52. PM_buffer Layout

Offset	Description
0x0	256-bit mask defining which PMC registers are implemented.
0x20	256-bit mask defining which PMD registers are implemented.
0x40	256-bit mask defining which registers can count cycles.
0x60	256-bit mask defining which registers can count retired bundles.

Set Processor Interrupt Block Address and I/O Port Space Address

Purpose: Specifies the physical address of the processor Interrupt Block and I/O Port Space.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Arguments:

Argument	Description
index	Index of PAL_PLATFORM_ADDR within the list of PAL procedures.
type	Unsigned 64-bit integer specifying the type of block. 0 indicates that the processor interrupt block pointer should be initialized. 1 indicates that the processor I/O block pointer should be initialized.
address	Unsigned 64-bit integer specifying the address to which the processor I/O block or interrupt block shall be set. The address must specify an implemented physical address on the processor model, bit 63 is ignored.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PLATFORM_ADDR procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: PAL_PLATFORM_ADDR specifies the physical address that the processor shall interpret as accesses to the SAPIC memory or the I/O Port space areas.

The default value for the Interrupt block pointer is 0x00000000 FEE00000. If an alternate address is selected by this call, it must be aligned on a 2 MB boundary, else the procedure will return an error status. The address specified must also not overlay any firmware addresses in the 16 MB region immediately below the 4GB physical address boundary.

The default value for the I/O block pointer is to the beginning of the 64 MB block at the highest physical address supported by the processor. Therefore, its physical address is implementation dependent. If an alternate address is selected by this call, it must be aligned on a 64MB boundary, else the procedure will return an error status. The address specified must also not overlay any firmware addresses in the 16 MB region immediately below the 4GB physical address boundary.

The Interrupt and I/O Block pointers should be initialized by firmware before any Inter-Processor Interrupt messages or I/O Port accesses. Otherwise the default block pointer values will be used.

Setup SAL PMI Entrypoint in Memory

Purpose: Sets the SAL PMI entrypoint in memory.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_PMI_ENTRYPOINT within the list of PAL procedures.
SAL_PMI_entry	256-byte aligned physical address of SAL PMI entrypoint in memory.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PMI_ENTRYPOINT procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This procedure is called to set the SAL PMI entrypoint so that the SAL PMI code shall be executed out of main memory instead of the firmware address space. Some processor implementations will allow initialization of the PMI entrypoint only once. Under those situations, this procedure may be called only once after a boot to initialize the PMI entrypoint register. Subsequent calls will return a status of -3. This call must be made before PMI is enabled by SAL.

Make Processor Prefetches Visible

Purpose: Used in the architected sequences for memory attribute transitions described in Section 4.4.11 "Memory Attribute Transition" to transition a page (or set of pages) from a one memory attribute to another.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_PREFETCH_VISIBILITY within the list of PAL procedures.
trans_type	Unsigned integer specifying the type of memory attribute transition that is being performed
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PREFETCH_VISIBILITY procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call completed without error; this call is not necessary on remote processors
0	Call completed without error; this call must also be performed on all remote processors in the coherence domain
-2	Invalid argument
-3	Call completed with error

Description: This call is intended to be used only in the architected sequences described in Section 4.4.11 "Memory Attribute Transition". Use of this procedure outside the context of this sequence results in undefined behavior.

The *trans_type* input indicates if a user is transitioning virtual addressing memory attributes (input value of 0) or physical addressing memory attributes (input value of 1). All other values are reserved.

This procedure, when used for transitioning virtual memory attributes, will ensure that all prefetches that were initiated by the processor to the cacheable, speculative memory prior to the call, will either not be cached; have been aborted; or are visible to subsequent `fc` instructions. (from both the local processor and from remote processors).

This procedure when used for transitioning physical memory attributes will ensure that all prefetches that were initiated by the processor to the cacheable, limited speculative memory prior to the call, will either not be cached; have been aborted; or are visible to subsequent `fc` instructions (from both the local processor and from remote processors). It will also terminate the ability for the processor to make speculative references to any limited speculation pages. For the processor to make any speculative reference to a limited speculation page after this call, there must be a non-speculative reference made to that page after this call.

If the processor implementation does not require this procedure call to be made on remote processors in the sequences, this procedure will return a 1 upon successful completion.

A return value of 0 upon successful completion of this procedure is an indication to software that the processor implementation requires that this call be performed on all processors in the coherence domain to make prefetches visible in the sequences.

These return code can be used to tune the architected sequence to the particular system on which is running; see Section 4.4.11 "Memory Attribute Transition" for details.

Get Processor Dependent Features

Purpose: Provides information about configurable processor features.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_PROC_GET_FEATURES within the list of PAL procedures.
Reserved	0
feature_set	Feature set information is being requested for.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PROC_GET_FEATURES procedure.
features_avail	64-bit vector of features implemented. See Table 11-53.
feature_status	64-bit vector of current feature settings. See Table 11-53.
feature_control	64-bit vector of features controllable by software.

Status:

Status Value	Description
1	Call completed without error; The *feature_set* passed is not supported but a *feature_set* of a larger value is supported
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-8	*feature_set* passed is beyond the maximum *feature_set* supported

Description: PAL_PROC_GET_FEATURES and PAL_PROC_SET_FEATURES procedure calls are used together to describe current settings of processor features and to allow modification of some of these processor features.

The *feature_set* input argument for PAL_PROC_GET_FEATURES describes which processor *feature_set* information is being requested. Table 11-53 describes processor *feature_set* zero. The *feature_set* values are split into two categories: architected and implementation-specific. The architected *feature_sets* have values from 0-15. The implementation-specific *feature_sets* are values 16 and above. The architected *feature_sets* are described in this document. The implementation-specific *feature_sets* are described in processor-specific documentation.

This procedure will return an invalid argument if an unsupported architectural *feature_set* is passed as an input. Implementation-specific *feature_sets* will start at 16 and will expand in an ascending order as new implementation-specific *feature_sets* are added. The return *status* is used by the caller to know which implementation-specific *feature_sets* are currently supported on a particular processor.

For each valid *feature_set*, this procedure returns which processor features are implemented in the *features_avail* return argument, the current feature setting is in *feature_status* return argument, and the feature controllability in the *feature_control* return argument. Only the processor features which are implemented and controllable can be changed via PAL_PROC_SET_FEATURES.

In Table 11-53, the *class* field indicates whether a feature is required to be available (*Req.*) or is optional (*Opt.*). The *control* field indicates which features are required to be controllable. *Req.* indicates that the feature must be controllable, *Opt.* indicates that the feature may optionally be controllable, and *No* indicates that the feature cannot be controllable. The *control* field applies only when the feature is available. The sense of the bits is chosen so that for features which are controllable, the default hand-off value at exit from PALE_RESET should be 0. PALE_CHECK and PALE_INIT will not modify these features.

Table 11-53. Processor Features

Bit	Class	Control	Description
63	Opt.	Req.	Enable BERR promotion. When 1, the Bus Error (BERR) signal is promoted to the Bus Initialization (BINIT) signal, and the BINIT pin is asserted on the occurrence of each Bus Error. Setting this bit has no effect if BINIT signalling is disabled. (See PAL_BUS_GET/SET_FEATURES)
62	Opt.	Req.	Enable MCA promotion. When 1, machine check aborts (MCAs) are promoted to the Bus Error signal, and the BERR pin is assert on each occurrence of an MCA. Setting this bit has no effect if BERR signalling is disabled. (See PAL_BUS_GET/SET_FEATURES)
61	Opt.	Req.	Enable MCA to BINIT promotion. When 1, machine check aborts (MCAs) are promoted to the Bus Initialization signal, and the BINIT pin is assert on each occurrence of an MCA. Setting this bit has no effect if BINIT signalling is disabled. (See PAL_BUS_GET/SET_FEATURES)
60	Opt.	Req.	Enable CMCI promotion When 1, Corrected Machine Check Interrupts (CMCI) are promoted to MCAs. They are also further promoted to BERR if bit 39, Enable MCA promotion, is also set and they are promoted to BINIT if bit 38, Enable MCA to BINIT promotion, is also set. This bit has no effect if MCA signalling is disabled (see PAL_BUS_GET/SET_FEATURES)
59	Opt.	Req.	Disable Cache. When 0, the processor performs cast outs on cacheable pages and issues and responds to coherency requests normally. When 1, the processor performs a memory access for each reference regardless of cache contents and issues no coherence requests and responds as if the line were not present. Cache contents cannot be relied upon when the cache is disabled. WARNING: Semaphore instructions may not be atomic or may cause Unsupported Data Reference faults if caches are disabled.
58	Opt.	Req.	Disable Coherency. When 0, the processor uses normal coherency requests and responses. When 1, the processor answers all requests as if the line were not present.
57	Opt.	Req.	Disable Dynamic Power Management (DPM). When 0, the hardware may reduce power consumption by removing the clock input from idle functional units. When 1, all functional units will receive clock input, even when idle.
56	Opt.	Req.	Disable a BINIT on internal processor time-out. When 0, the processor may generate a BINIT on an internal processor time-out. When 1, the processor will not generate a BINIT on an internal processor time-out. The event is silently ignored.
55	Opt.	Req.	Enable external notification when the processor detects hardware errors caused by environmental factors that could cause loss of deterministic behavior of the processor. When 1, this bit will enable external notification, when 0 external notification is not provided. The type of external notification of these errors is processor-dependent. A loss of processor deterministic behavior is considered to have occurred if these environmentally induced errors cause the processor to deviate from its normal execution and eventually causes different behavior which can be observed at the processor bus pins. Processor errors that do not have this effects (i.e., software induced machine checks) may or may not be promoted depending on the processor implementation.
54-48	N/A	N/A	reserved
47	Opt.	Opt.	Disable Dynamic branch prediction. When 0, the processor may predict branch targets and speculatively execute, but may not commit results. When 1, the processor must wait until branch targets are known to execute.
46	Opt	Opt.	Disable Dynamic Instruction Cache Prefetch. When 0, the processor may prefetch into the caches any instruction which has not been executed, but whose execution is likely. When 1, instructions may not be fetched until needed or hinted for execution. (Prefetch for a hinted branch is allowed even when dynamic instruction cache prefetch is disabled.)

Table 11-53. Processor Features (Continued)

Bit	Class	Control	Description
45	Opt.	Opt.	Disable Dynamic Data Cache Prefetch. When 0, the processor may prefetch into the caches any data which has not been accessed by instruction execution, but which is likely to be accessed. When 1, no data may be fetched until it is needed for instruction execution or is fetched by an lfetch instruction.
44	N/A	N/A	reserved
43	Opt.	Opt.	Disable Dynamic Predicate Prediction. When 0, the processor may predict predicate results and execute speculatively, but may not commit results until the actual predicates are known. When 1, the processor shall not execute predicated instructions until the actual predicates are known.
42	Opt.	No	XR1 through XR3 implemented. Denotes whether XR1 - XR3 are implemented for machine check recovery. This feature may only be interrogated by PAL_PROC_GET_FEATURES. It may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.
41	Opt.	No	XIP, XPSR, and XFS implemented. Denotes whether XIP, XPSR, and XFS are implemented for machine check recovery. This feature may only be interrogated by PAL_PROC_GET_FEATURES. It may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.
40-0	N/A	N/A	reserved

Set Processor Dependent Features

Purpose: Enables/disables specific processor features.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_PROC_SET_FEATURES within the list of PAL procedures.
feature_select	64-bit vector denoting desired state of each feature (1=select, 0=non-select).
feature_set	Feature set to apply changes to. See PAL_PROC_GET_FEATURES for more information on feature sets.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PROC_SET_FEATURES procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call completed without error; The *feature_set* passed is not supported but a *feature_set* of a larger value is supported
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-8	*feature_set* passed is beyond the maximum *feature_set* supported

Description: PAL_PROC_GET_FEATURES should be called to ascertain the implemented processor features and their current setting before calling PAL_PROC_SET_FEATURES. The list of possible processor features is defined in Table 11-53. Any attempt to set processor features which cannot be set will be ignored.

Get PTCE Purge Loop Information

Purpose: Returns information required for the architected loop used to purge (initialize) the entire TC.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_PTCE_INFO within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PTCE_INFO procedure.
tc_base	Unsigned 64-bit integer denoting the beginning address to be used by the first PTCE instruction in the purge loop.
tc_counts	Two unsigned 32-bit integers denoting the loop counts of the outer (loop 1) and inner (loop 2) purge loops. count1 (loop 1) is contained in bits 63:32 of the parameter, and count2 (loop 2) is contained in bits 31:0 of the parameter.
tc_strides	Two unsigned 32-bit integers denoting the loop strides of the outer (loop 1) and inner (loop 2) purge loops. stride1 (loop 1) is contained in bits 63:32 of the parameter, and stride2 (loop 2) is contained in bits 31:0 of the parameter.

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: No explicit hardware support is required by this call. See the purge loop example in the description of the `ptc.e` instruction in Chapter 2 in *Volume 3: Instruction Set Reference*.

Return Information about Implemented Processor Registers

Purpose: Returns information about implemented Application and Control Registers.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Arguments:

Argument	Description
index	Index of PAL_REGISTER_INFO within the list of PAL procedures.
info_request	Unsigned 64-bit integer denoting what register information is requested.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_REGISTER_INFO procedure.
reg_info_1	64-bit vector denoting information for registers 0-63. Bit 0 is register 0, bit 63 is register 63.
reg_info_2	64-bit vector denoting information for registers 64-127. Bit 0 is register 64, bit 63 is register 127.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

This procedure is called to obtain information about the implementation of Application Registers and Control Registers. Table 11-54 shows the information that is returned for each request.

Table 11-54. info_request Return Value

info_request	Meaning of Return Bit Vector
0	A 0-bit in the return vector indicates that the corresponding Application Register is not implemented, a 1-bit in the return vector indicates that the corresponding Application Register is implemented.
1	A 0-bit in the return vector indicated that the corresponding Application Register can be read without side effects, a 1-bit in the return vector indicated that the corresponding Application registers may cause side effects when read.
2	A 0-bit in the return vector indicates that the corresponding Control Register is not implemented, a 1-bit in the return vector indicates that the corresponding Control Register is implemented.
3	A 0-bit in the return vector indicated that the corresponding Control Register can be read without side effects, a 1-bit in the return vector indicated that the corresponding Control Register may cause side effects when read.
All others	Reserved.

Get RSE Information

Purpose: Returns information about the register stack and RSE for this processor implementation.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Arguments:

Argument	Description
index	Index of PAL_RSE_INFO within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_RSE_INFO procedure.
phys_stacked	Number of physical stacked general registers.
hints	RSE hints supported by processor.
Reserved	0

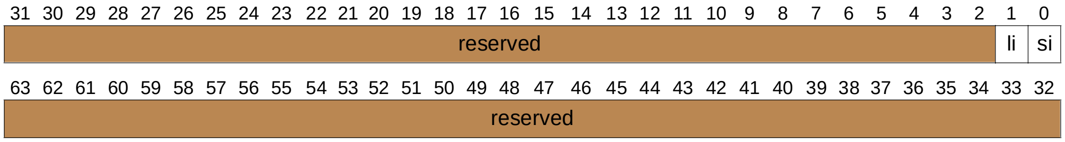
Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: The return parameter *phys_stacked* contains a 64-bit unsigned integer that specifies the number of physical registers implemented by the processor for the stacked general registers, r32-r127. *phys_stacked* will be an integer multiple of 16 greater than or equal to 96.

The return parameter *hints* contains a 2-bit field that specifies which RSE load/store hints are implemented.

Figure 11-38. Layout of *hints* Return Value



A bit field value of 1 specifies that the corresponding mode is implemented; a value of 0 specifies that the mode is not implemented. The bit field encodings are:

Table 11-55. RSE Hints Implemented

li	si	RSE Hints	Class
0	0	enforced lazy	Required
0	1	eager stores	Optional
1	0	eager loads	Optional
1	1	eager stores and loads	Optional

“Lazy” is the default RSE mode and must be implemented. Hardware is not required to implement any of the other modes.

Information for Processor Self-test

Purpose: Returns the alignment and size requirements needed for the memory buffer passed to the PAL_TEST_PROC procedure as well as information on self-test control words for the processor self-tests.

Calling Conv: Static Registers Only

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_TEST_INFO within the list of PAL procedures.
test_phase	Unsigned integer that specifies which phase of the processor self-test information is being requested on. A value of 0 indicates the phase two of the processor self-test and a value of 1 indicates phase one of the processor self-test. All other values are reserved.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_TEST_INFO procedure.
bytes_needed	Unsigned 64-bit integer denoting the number of bytes of main memory needed to perform the second phase of processor self-test.
alignment	Unsigned 64-bit integer denoting the alignment required for the memory buffer.
st_control	48-bit wide bit-field indicating if control of the processor self-tests is supported and which bits of the *test_control* field are defined for use.

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: PAL_TEST_INFO returns the size and alignment requirements for the memory buffer that is passed to the PAL_TEST_PROC procedure and returns information on the implementation of the self-test control word based on the *test_phase* input argument. Please see Section 11.2.3, “PAL Self-test Control Word” on page 2:267 for more information on the self-test control word.

When *test_phase* is equal to zero, information is returned about phase two of the processor self-test. These are the tests that require external memory to execute properly. When *test_phase* is equal to one, information is returned about phase one of the processor self-test. These are the tests that are normally run during PALE_RESET and do not require external memory to properly execute. When information is requested about phase one of the processor self-test a memory buffer and alignment argument will be returned as well since these tests may need to save and restore processor state to this memory buffer if executed from the PAL_TEST_PROC procedure.

Perform a Processor Self-test

Purpose: Performs the second phase of processor self test.

Calling Conv: Stacked Registers

PAL_TEST_PROC may modify some registers marked unchanged in the Stacked Register calling convention. See additional description below.

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_TEST_PROC within the list of PAL procedures.
test_address	64-bit physical address of main memory area to be used by processor self-test. The memory region passed must be cacheable, bit 63 must be zero.
test_info	Input argument specifying the size of the memory buffer passed and the phase of the processor self-test that should be run. See Figure 11-39.
test_params	Input argument specifying the self-test control word and the allowable memory attributes that can be used with the memory buffer. See Figure 11-40.

Returns:

Return Value	Description
status	Return status of the PAL_TEST_PROC procedure.
self-test_state	Formatted 8-byte value denoting the state of the processor after self-test. The format is described in Section 11.2.2.2, “Definition of Self Test State Parameter” on page 2:266.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: The PAL_TEST_PROC procedure will perform a phase of the processor self-tests as directed by the *test_info* and the *test_control* input parameters.

test_address points to a contiguous memory region to be used by PAL_TEST_PROC. This memory region must be aligned as specified by the alignment return value from PAL_TEST_INFO, otherwise this procedure will return with an invalid argument return value. The PAL_TEST_PROC routine requires that the memory has been initialized and that there are no known uncorrected errors in the allocated memory.

The *test_info* input parameter specifies the size of the memory buffer passed to the procedure and which phase of the processor self-test is requested to be run (either phase one or phase two).

Figure 11-39. Layout of *test_info* Argument

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">buffer_size</td></tr>
</table>

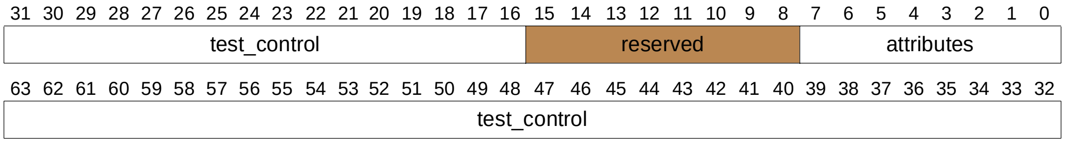
<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="8">test_phase</td><td colspan="24">buffer_size</td></tr>
</table>

- *buffer_size* indicates the size in bytes of the memory buffer that is passed to this procedure. *buffer_size* must be greater than or equal in size to the *bytes_needed* return value from PAL_TEST_INFO, otherwise this procedure will return with an invalid argument return value.
- *test_phase* defines which phase of the processor self-tests are requested to be run. A value of zero indicates to run phase two of the processor self-tests. Phase two of the processor self-tests are ones that require external memory to execute correctly. A value of one indicates to run phase one of the processor self-tests. Phase one of the processor self-tests are tests run during

PALE_RESET and do not depend on external memory to run correctly. When the caller requests to have phase one of the processor self-test run via this procedure call, a memory buffer may be needed to save and restore state as required by the PAL calling conventions. The procedure PAL_TEST_INFO informs the caller about the requirements of the memory buffer.

The *test_params* input argument specifies which memory attributes are allowed to be used with the memory buffer passed to this procedure as well as the self-test control word. The self-test control word *test_control* controls the run-time and coverage of the processor self-test phase specified in the *test_phase* parameter.

Figure 11-40. Layout of *test_param* Argument



- *attributes* specifies the memory attributes that are allowed to be used with the memory buffer passed to this procedure. The *attributes* parameter is a vector where each bit represents one of the virtual memory attributes defined by the architecture. The bit field position corresponds to the numeric memory attribute encoding defined in Section 4.4 "Memory Attributes". The caller is required to support the cacheable attribute for the memory buffer, otherwise an invalid argument will be returned.
- *test_control* is the self-test control word corresponding to the *test_phase* passed. This *test_control* directs the coverage and run-time of the processor self-tests specified by the *test_phase* input argument. Information about the self-test control word can be found in Section 11.2.3, "PAL Self-test Control Word" on page 2:267 and information on if this feature is implemented and the number of bits supported can be obtained by the PAL_TEST_INFO procedure call. If this feature is implemented by the processor, the caller can selectively skip parts of the processor self-test by setting *test_control* bits to a one. If a bit has a zero, this test will be run. The values in the unimplemented bits are ignored. If PAL_TEST_INFO indicated that the self-test control word is not implemented, this procedure will return with an invalid argument status if the caller sets any of the *test_control* bits.

PAL_TEST_PROC will classify the processor after the self-test in one of four states: CATASTROPHIC FAILURE, FUNCTIONALLY RESTRICTED, PERFORMANCE RESTRICTED, or HEALTHY. These processor self-test states are described in Figure 11-9 on page 2:266. If PAL_TEST_PROC returns in the FUNCTIONALLY RESTRICTED or PERFORMANCE RESTRICTED states the *self-test_status* return value can provide additional information regarding the nature of the failure. In the case of a CATASTROPHIC FAILURE, the procedure does not return.

The procedure will only perform memory accesses to the buffer passed to it using the memory attributes indicated in the *attributes* bit-field. The caller must ensure that the memory region passed to the procedure is in a coherent state.

PAL_TEST_PROC may modify PSR bits or system registers as necessary to test the processor. These bits or registers must be restored upon exit from PAL_TEST_PROC with the exception of the translation caches, which are evicted as a result of testing. PAL_TEST_PROC is free to invalidate all cache contents. If the caller depends on the contents of the cache, they should be flushed before making this call. PAL_TEST_PROC requires that the RSE is set up properly to handle spills and fills to a valid memory location if the contents of the register stack are needed. PAL_TEST_PROC requires that the memory buffer passed to it is not shared with other processors running this procedure in the system at the same time. PAL_TEST_PROC will use this memory region in a non-coherent manner.

Get PAL Version Number Information

Purpose: Returns PAL version information.

Calling Conv: Static registers only

Mode: Physical or Virtual

Arguments:

Argument	Description
index	Index of PAL_VERSION within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VERSION procedure.
min_pal_ver	8-byte formatted value returning the minimum PAL version needed for proper operation of the processor. See Figure 11-41.
current_pal_ver	8-byte formatted value returning the current PAL version running on the processor. See Figure 11-41.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: PAL_VERSION provides the caller the minimum PAL version needed for proper operation of the processor as well as the current PAL version running on the processor.

The *min_pal_ver* and *current_pal_ver* return values are 8-byte values in the following format:

Figure 11-41. Layout of *min_pal_ver* and *current_pal_ver* Return Values

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PAL_vendor								Reserved								PAL_B_version															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Reserved																PAL_A_version															

- *PAL_B_version* is a 16-bit binary coded decimal (BCD) number that provides identification information about the PAL_B firmware.
- *PAL_vendor* is an unsigned 8-bit integer indicating the vendor of the PAL code.
- *PAL_A_version* is a 16-bit binary coded decimal (BCD) number that provides identification information about the PAL_A firmware.

The version numbers selected for the PAL_A and PAL_B firmware is specific to the *PAL_vendor.* The version numbers selected will always have the property that later versions of firmware will have a higher number than earlier versions of firmware.

Get Virtual Memory Information

Purpose: Return information about the virtual memory characteristics of the processor implementation.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_VM_INFO within the list of PAL procedures.
tc_level	Unsigned 64-bit integer specifying the level in the TLB hierarchy for which information is required. This value must be between 0 and one less than the value returned in the *vm_info_1.num_tc_levels* return value from PAL_VM_SUMMARY.
tc_type	Unsigned 64-bit integer with a value of 1 for instruction translation cache and 2 for data or unified translation cache. All other values are reserved.
Reserved	0

Returns:

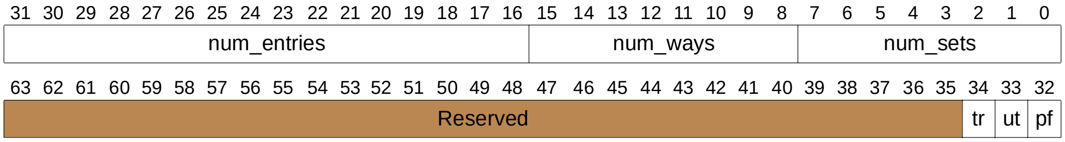
Return Value	Description
status	Return status of the PAL_VM_INFO procedure.
tc_info	8-byte formatted value returning information about the specified TC.
tc_pages	64-bit vector containing a bit for each page size supported in the specified TC, where bit position n indicates a page size of 2**n.
Reserved	0

Status:

Status Value	Description
0	Call completed without error.
-2	Invalid argument.
-3	Call completed with error.

Description: The *tc_info return* is an 8-byte quantity in the following format:

Figure 11-42. Layout of *tc_info* Return Value



- *num_sets* - unsigned 8-bit integer denoting the number of hash sets for the specified level (1=fully associative)
- *num_ways* - unsigned 8-bit integer denoting the associativity of the specified level (1=direct).
- *num_entries* - unsigned 16-bit integer denoting the number of entries in the specified TC.
- *pf* - flag denoting whether the specified level is optimized for the region's preferred page size (1=optimized). *tc_pages* indicates which page sizes are usable by this translation cache.
- *ut* - flag denoting whether the specified TC is unified (1=unified).
- tr - flag denoting whether installed translation registers will reduce the number of entries within the specified TC.

The *num_entries* will always equal *num_ways* * *num_sets*. For a direct mapped TC, *num_ways* = 1 and *num_sets* = *num_entries*. For a fully associative TC, *num_sets* = 1 and *num_ways* = *num_entries*.

Get Virtual Memory Page Size Information

Purpose: Returns page size information about the virtual memory characteristics of the processor implementation.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_VM_PAGE_SIZE within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VM_PAGE_SIZE procedure.
insertable_pages	64-bit vector containing a bit for each architected page size that is supported for TLB insertions and region registers.
purge_pages	64-bit vector containing a bit for each architected page size supported for TLB purge operations.
Reserved	0

Status:

Status Value	Description
0	Call completed without error.
-2	Invalid argument
-3	Call completed with error.

Description: The values returned from this call are all 64-bit bitmaps. One bit is set for each page size implemented by the processor where bit n represents a page size of 2**n. Please refer to Table 4-4 on page 2:47 for the minimum page sizes that are supported.

The *insertable_pages* returns the page sizes that are supported for TLB insertions and region registers.

The *purge_pages* returns the page sizes that are supported for the TLB purge operations.

Get Virtual Memory Summary Information

Purpose: Returns summary information about the virtual memory characteristics of the processor implementation.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_VM_SUMMARY within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

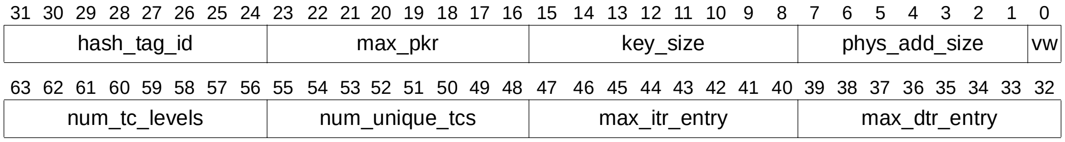
Return Value	Description
status	Return status of the PAL_VM_SUMMARY procedure.
vm_info_1	8-byte formatted value returning global virtual memory information.
vm_info_2	8-byte formatted value returning global virtual memory information.
Reserved	0

Status:

Status Value	Description
0	Call completed without error.
-2	Invalid argument
-3	Call completed with error.

Description: The *vm_info_1* return is an 8-byte quantity in the following format:

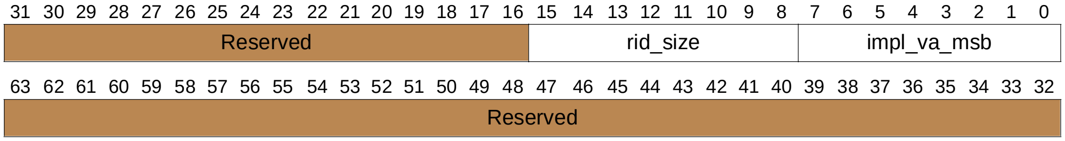
Figure 11-43. Layout of *vm_info_1* Return Value



- *vw* - 1-bit flag indicating whether a hardware TLB walker is implemented (1 = walker present).
- *phys_add_size* - unsigned 7-bit integer denoting the number of bits of physical address implemented.
- *key_size* - unsigned 8-bit integer denoting the number of bits implemented in the PKR.key field.
- *max_pkr* - unsigned 8-bit integer denoting the maximum PKR index (number of PKRs-1).
- *hash_tag_id* - unsigned 8-bit integer which uniquely identifies the processor hash and tag algorithm.
- *max_dtr_entry* - unsigned 8 bit integer denoting the maximum data translation register index (number of dtr entries - 1).
- *max_itr_entry* - unsigned 8 bit integer denoting the maximum instruction translation register index (number of itr entries - 1).
- *num_unique_tcs* - unsigned 8-bit integer denoting the number of unique TCs implemented. This is a maximum of 2**num_tc_levels*.
- *num_tc_levels* - unsigned 8-bit integer denoting the number of TC levels.

The *vm_info_2* return is an 8-byte quantity in the following format:

Figure 11-44. Layout of *vm_info_2* Return Value



- *impl_va_msb* - unsigned 8-bit integer denoting the bit number of the most significant virtual address bit. This is the total number of virtual address bits - 1.
- *rid_size* - unsigned 8-bit integer denoting the number of bits implemented in the RR.rid field.

Read a Translation Register

Purpose: Reads a translation register.

Calling Conv: Stacked Registers

Mode: Physical

Arguments:

Argument	Description
index	Index of PAL_VM_TR_READ within the list of PAL procedures.
reg_num	Unsigned 64-bit number denoting which TR to read.
tr_type	Unsigned 64-bit number denoting whether to read an ITR (0) or DTR (1). All other values are reserved.
tr_buffer	64-bit pointer to the 32-byte memory buffer in which translation data is returned.

Returns:

Return Value	Description
status	Return status of the PAL_VM_TR_READ procedure.
TR_valid	Formatted bit vector denoting which fields are valid. See Figure 11-45.
Reserved	0
Reserved	0

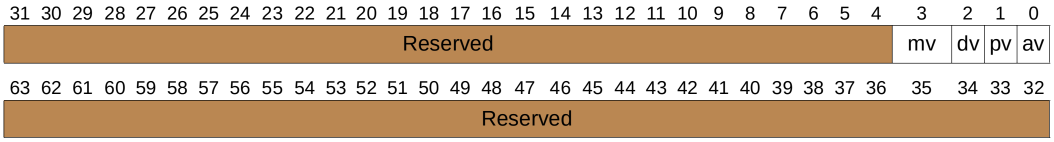
Status:

Status Value	Description
0	Call completed without error.
-2	Invalid argument
-3	Call completed with error.

Description: This procedure reads the specified translation register and returns its data in the buffer starting at *tr_buffer*. The format of the data is returned in Translation Insertion Format, as described in Figure 4-5, "Translation Insertion Format," on page 2:44. In addition, bit 0 of the IFA in Figure 4-5 (an ignored field in the figure) will return whether the translation is valid. If bit 0 is 1, the translation is valid.

Some fields of the translation register returned may be invalid. The validity of these fields is indicated by the return argument *TR_valid*. If these fields are not valid, the caller should ignore the indicated fields when reading the translation register returned in *tr_buffer*.

Figure 11-45. Layout of *TR_valid* Return Value



- av - denotes that the access rights field is valid
- pv - denotes that the privilege level field is valid
- dv - denotes that the dirty bit is valid
- mv - denotes that the memory attributes are valid.

A value of 1 denotes a valid field. A value of 0 denotes an invalid field. Any value returned in an invalid field must be ignored.

The *tr_buffer* parameter should be aligned on an 8 byte boundary.

Warning: This procedure may have the side effect of flushing all the translation cache entries depending on the implementation.

Part II: System Programmer’s Guide

About the System Programmer's Guide 1

Part II: System Programmer's Guide is intended as a companion section to the information presented in *Part I: System Architecture Guide*. While *Part I* provides a crisp and concise architectural definition of the Itanium instruction set, *Part II* provides insight into programming and usage models of the Itanium system architecture. This section emphasizes how the various architecture features fit together and explains how they contribute to high performance system software.

The intended audience for this section is system programmers who would like to better understand the Itanium system architecture. The goal of this document is to:

- Familiarize system programmers with Itanium system architecture principles and usage models.
- Provide recommendations, code examples, and performance guidelines.

This section does not re-define the Itanium instruction set. Please refer to *Part I: System Architecture Guide* as the authoritative definition of the system architecture.

The reader is expected to be familiar with the contents of *Part I* and is expected to be familiar with modern virtual memory and multi-processing concepts. Furthermore, this document is platform architecture neutral (i.e. no assumptions are made about platform architecture capabilities, such as busses, chipsets, or I/O devices).

1.1 Overview of the System Programmer's Guide

The Itanium architecture provides numerous performance enhancing features of interest to the system programmer. Many of these instruction set features focus on reducing overhead in common situations. The chapters outlined below discuss different aspects of the Itanium system architecture.

Chapter 2, "MP Coherence and Synchronization" describes Itanium-based multi-processing synchronization primitives and the Itanium memory ordering model. This chapter also discusses programming rules for self- and cross-modifying code. This chapter is useful for application and system programmers who write multi-threaded code.

Chapter 3, "Interruptions and Serialization" discusses how the Itanium architecture, despite its explicitly parallel instruction execution semantics, provides the system programmer with a precise interruption model. This chapter 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. This chapter introduces the interrupt vector table and describes how low-level kernel code is expected to transfer control to higher level operating system code written in a high-level programming language. This chapter is useful for operating system and firmware programmers.

Chapter 4, “Context Management” describes how operating systems need to preserve Itanium register contents. In addition to spilling and filling a register’s data value, the Itanium architecture also requires software to preserve control and data speculative state associated with that register, i.e. its NaT bit and ALAT state. This chapter also discusses system architecture mechanisms that allow an operating system to significantly reduce the number of registers that need to be spilled/filled on interruptions, system calls, and context switches. These optimizations improve the performance of an Itanium-based operating system by reducing the amount of required memory traffic. This chapter is useful for operating system programmers.

Chapter 5, “Memory Management” introduces various memory management strategies in the Itanium architecture: region register model, protection keys, and the virtual hash page table usage models are described. This chapter is of interest to virtual memory management software developers.

Chapter 6, “Runtime Support for Control and Data Speculation” describes the operating system support that is required for control and data speculation. This chapter describes various speculation software models and their associated operating system implications. This chapter is of interest to operating system developers and compiler writers.

Chapter 7, “Instruction Emulation and Other Fault Handlers” describes a variety of instruction emulation handlers that Itanium-based operating systems are expected to support. This chapter is useful for operating system developers.

Chapter 8, “Floating-point System Software” discusses how processors based on the Itanium architecture handle floating-point numeric exceptions and how the Itanium-based software stack provides complete IEEE-754 compliance. This includes a discussion of the floating-point software assist firmware, the FP SWA EFI driver. This chapter also describes how Itanium-based operating systems are expected to support IEEE floating-point exception filters. This chapter is useful for operating system developers and floating-point numerics experts.

Chapter 9, “IA-32 Application Support” outlines how software needs to perform instruction set transitions, and what low-level kernel handlers are required in an Itanium-based operating system to support IA-32 applications. This chapter is useful for operating system developers.

Chapter 10, “External Interrupt Architecture” describes the external interrupt architecture with a focus on how external asynchronous interrupt handling can be controlled by software. Basic interrupt prioritization, masking, and harvesting capabilities are discussed in this chapter. This chapter is of interest to operating system developers and to device driver writers.

Chapter 11, “I/O Architecture” describes the I/O architecture with a focus on platform considerations and support for the existing IA-32 I/O port space platform infrastructure. This chapter is of interest to operating system developers and to device driver writers.

Chapter 12, “Performance Monitoring Support” describes the performance monitor architecture with a focus on what kind of operating system support is needed from Itanium-based operating systems. This chapter is of interest to operating system and performance tool developers.

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. This chapter also discusses how firmware layers and the operating system work together to provide error detection, error logging, as well as fault containment capabilities. This chapter is of interest to platform firmware and operating system developers.

1.2 Related Documents

The following documents are referred to fairly often in this document. For more details on software conventions and platform firmware, please consult these manuals (available at http://developer.intel.com).

[SWC] *“Itanium Software Conventions and Runtime Architecture Guide”*

[EFI] *“Extensible Firmware Interface (EFI) Specification”*

MP Coherence and Synchronization 2

This chapter describes how to enforce an ordering of memory operations, how to update code images, and presents examples of several simple multiprocessor synchronization primitives on a processor based on the Itanium architecture. These topics are relevant to anyone who writes either user- or system-level software for multiprocessor systems based on the Itanium architecture.

The chapter begins with a brief overview of Itanium memory access instructions intended to summarize the behaviors that are relevant to later discussions in the chapter. Next, this chapter presents the Itanium memory ordering model and compares it to a sequentially-consistent ordering model. It then explores versions of several common synchronization primitives. This chapter closes by describing how to correctly update code images to implement self-modifying code, cross-modifying code, and paging of code using programmed I/O.

2.1 An Overview of Intel® Itanium™ Memory Access Instructions

The Itanium architecture provides load, store, and semaphore instructions to access memory. In addition, it also provides a memory fence instruction to enforce further ordering relationships between memory accesses. As Section 4.4.7, "Memory Access Ordering" in Volume 1 describes, memory operations in the Itanium architecture come with one of four semantics: unordered, acquire, release, or fence. Section 2.2 on page 2:378 describes how the memory ordering model uses these semantics to indicate how memory operations can be ordered with respect to each other.

Section 2.1.1 defines the four memory operation semantics. Section 2.2, Section 2.3, and Section 2.4 present brief outlines of load and store, semaphore, and memory fence instructions in the Itanium architecture. Refer to Section 2, "Instruction Reference" in Volume 3 for more information on the behavior and capabilities of these instructions.

2.1.1 Memory Ordering of Cacheable Memory References

The Itanium architecture has a relaxed memory ordering model which provides unordered memory opcodes, explicitly ordered memory opcodes, and a fencing operation that software can use to implement stronger ordering. Each memory operation establishes an ordering relationship with other operations through one of four semantics:

- *Unordered* semantics imply that the instruction is made visible in any order with respect to other orderable instructions.
- *Acquire* semantics imply that the instruction is made visible prior to all subsequent orderable instructions.
- *Release* semantics imply that the instruction is made visible after all prior orderable instructions.
- *Fence* semantics combine acquire and release semantics (i.e. the instruction is made visible after all prior orderable instructions and before all subsequent orderable instructions).

In the above definitions "prior" and "subsequent" refer to the program-specified order. An "orderable instruction" is an instruction that the memory ordering model can use to establish ordering relationships[1]. The term "visible" refers to all architecturally-visible (from the standpoint of multi-processor coherency) effects of performing an instruction. Specifically,

- Accesses to uncacheable or write-coalescing memory regions are visible when they reach the processor bus.
- Loads from cacheable memory regions are visible when they hit a non-programmer-visible structure such as a cache or store buffer.
- Stores to cacheable memory regions are visible when they enter a snooped (in a multi-processor coherency sense) structure.

Memory access instructions typically have an ordered and an unordered form (i.e. a form with unordered semantics and a form with either acquire, release, or fence semantics). The Itanium architecture does not provide all possible combinations of instructions and ordering semantics. For example, the Itanium instruction set does not contain a store with fence semantics.

Section 4.4.7, "Memory Access Ordering" in Volume 1 and Section 4.4.7, "Sequentiality Attribute and Ordering" discuss ordering, orderable instructions, and visibility in greater depth.

Section 2.2 on page 2:378 describes how the ordering semantics affect the Itanium memory ordering model.

2.1.2 Loads and Stores

In the Itanium architecture, a load instruction has either unordered or acquire semantics while a store instruction has either unordered or release semantics. By using acquire loads (`ld.acq`) and release stores (`st.rel`), the memory reference stream of an Itanium-based program can be made to operate according to the IA-32 ordering model. The Itanium architecture uses this behavior to provide IA-32 compatibility. That is, an Itanium acquire load is equivalent to an IA-32 load and an Itanium release store is equivalent to an IA-32 store, from a memory ordering perspective.

Loads can be either speculative or non-speculative. The speculative forms (`ld.s`, `ld.sa`, and `ld.a`) support control and data speculation.

2.1.3 Semaphores

The Itanium architecture provides a set of three semaphore instructions: exchange (`xchg`), compare and exchange (`cmpxchg`), and fetch and add (`fetchadd`). Both `cmpxchg` and `fetchadd` may have either acquire or release semantics depending on the specific opcode chosen. The `xchg` instruction always has acquire semantics. These instructions read a value from memory, modify this value using an instruction-specific operation, and then write the modified value back to memory. The read-modify-write sequence is atomic by definition.

[1] The ordering semantics of an instruction *do not* imply the orderability of the instruction. Specifically, unordered ordering semantics alone *do not* make an instruction unorderable; there are orderable instructions with each of the four ordering semantics.

2.1.3.1 Considerations for using Semaphores

The memory location on which a semaphore instruction operates on must obey two constraints. First, the location must be cacheable (the `fetchadd` instruction is an exception to this rule; it may also operate on exported uncacheable locations, UCE). Thus, with the exception of `fetchadd` to UCE locations, the Itanium architecture does not support semaphores in uncacheable memory. Second, the location must be naturally-aligned to the size of the semaphore access. If either of these two constraints are not met, the processor generates a fault.

The exported uncacheable memory attribute, UCE, allows a processor based on the Itanium architecture to export fetch and add operations to the platform. A processor that does not support exported `fetchadd` will fault when executing a `fetchadd` to a UCE memory location. If the processor supports exported `fetchadd` but the platform does not, the behavior is undefined when executing a `fetchadd` to a UCE memory location.

Sharing locks between IA-32 and Itanium-based code does work with the following restrictions:

- Itanium-based code can only manipulate an IA-32 semaphore if the IA-32 semaphore is aligned.
- Itanium-based code can only manipulate an IA-32 semaphore if the IA-32 semaphore is allocated in write-back cacheable memory.

An Itanium-based operating system can emulate IA-32 uncacheable or misaligned semaphores by using the technique described in the next section.

2.1.3.2 Behavior of Uncacheable and Misaligned Semaphores

A processor based on the Itanium architecture raises an Unsupported Data Reference fault if it executes a semaphore that accesses a location with a memory attribute that the semaphore does not support.

If the alignment requirement for Itanium-based semaphores is not met, a processor based on the Itanium architecture raises an Unaligned Data Reference fault. This fault is taken regardless of the setting of the user mask alignment checking bit, UM.ac.

The DCR.lc bit controls how the processor behaves when executing an atomic IA-32 memory reference under an external bus lock. When the DCR.lc bit (see Section 3.3.4.1, “Default Control Register (DCR – CR0)”) is 1 and an IA-32 atomic memory reference requires a non-cacheable or misaligned read-modify-write operation, an IA-32_Intercept(Lock) fault is raised. Such memory references require an external bus lock to execute correctly. To preserve `LOCK` pin functionality, an Itanium-based operating system can virtualize the bus lock by implementing a shared cacheable global `LOCK` variable.

To support existing IA-32 atomic read-modify-write operations that require the `LOCK` pin, an Itanium-based operating system can use the DCR.lc bit to intercept all external IA-32 read-modify-write operations. Then, the IA-32_Intercept(Lock) handler can emulate these operations by first acquiring a cacheable virtualized `LOCK` variable, then performing the required memory operations non-atomically, and then releasing the virtualized `LOCK` variable. This emulation allows the read-modify-write sequence to appear atomic to other processors that use the semaphore.

2.1.4 Memory Fences

The memory fence instruction (`mf`) is the only instruction in the Itanium instruction set with fence semantics. This instruction serializes the set of memory accesses before the memory fence in program order with respect to the set of memory accesses that follow the fence in program order.

2.2 Memory Ordering in the Intel® Itanium™ Architecture

Understanding a system's memory ordering model is key to writing either user- or system-level multiprocessor software that uses shared memory to communicate between processes and also that executes correctly on a shared-memory multiprocessor system. For a general introduction to memory ordering models, see Adve and Gharachorloo [AG95].

Four factors determine how a processor or system based on the Itanium architecture orders a group of memory operations with respect to each other:

- *Data dependencies* define the relationship between operations from the same processor that have register or memory dependencies on the same address[1]. This relationship need only be honored by the local processor (i.e. the processor that executes the operations).
- The *memory ordering semantics* define the relationship between memory operations from a particular processor that reference different addresses. For cacheable references, this relationship is honored by *all* observers in the coherence domain.
- Aligned *release stores* and *semaphore operations* (both require and release forms) become visible to all observers in the coherence domain in a single total order except each processor may observe its own release stores (via loads or acquire loads) prior to their being observed globally[2].
- Non-programmer-visible state, such as *store buffers, processor caches,* or any logically-equivalent structure, may satisfy read requests from loads or acquire loads on the local processor before the data in the structure is made globally visible to other observers.

In the Itanium architecture, dependencies between operations by a processor have implications for the ordering of those operations at that processor. The discussion in Section 2.2.1.6 on page 2:382 and Section 2.2.1.7 on page 2:383 explores this issue in greater depth.

The following sections examine the Itanium ordering model in detail. Section 2.2.1 presents several memory ordering executions to illustrate important behaviors of the model. Section 2.2.2 discusses how memory attributes and the ordering model interact. Finally, Section 2.2.3 describes how the Itanium memory ordering model compares with other memory ordering models.

[1] That is, A precedes B in program order and A produces a value that B consumes. This relationship is transitive.

[2] Consequently, each such operation appears to become visible to each observer in the coherence domain at the same time, with the exception that a release store can become visible to the storing processor before others.

2.2.1 Memory Ordering Executions

Multiprocessor software that uses shared memory to communicate between processes often makes assumptions about the order in which other agents in the system will observe memory accesses. As Section 2.1.1 on page 2:375 describes, the Itanium architecture provides a rich set of ordering semantics that allows software to express different ordering constraints on a memory operation, such as a load. Writing correct multiprocessor software requires that the programmer (or compiler) select the ordering semantic appropriate to enforce the expected behavior.

For example, an algorithm that requires two store operations A and B become visible to other processors in the order {A, B} will use stores with different ordering semantics than an algorithm that does not require any particular ordering of A and B. Although it is always safe to enforce stricter ordering constraints than an algorithm requires, doing so may lead to lower performance. If the ordering of memory operations is not important, software should use unordered ordering semantics whenever possible for best possible performance.

This section presents multiprocessor executions to demonstrate the ordering behaviors that the Itanium architecture allows and to contrast the Itanium ordering model with other ordering models. The executions consist of sequences of memory accesses that execute on two or more processors and highlight outcomes that the Itanium memory ordering model either allows or disallows once all accesses on all processors complete. A programmer can use these executions as a guide to determine which Itanium memory ordering semantics are appropriate to ensure a particular visibility order of memory accesses.

Section 2.2.1.1 presents the assumptions and notational conventions that the upcoming discussions use to examine the executions. The remaining eleven sections each explore a different facet of the Itanium ordering model:

- Relaxed ordering of unordered memory operations (Section 2.2.1.2).
- Using acquire and release semantics to order operations (Section 2.2.1.3).
- Loads may pass stores (Section 2.2.1.4) and how to prevent this behavior (Section 2.2.1.5).
- When dependencies do or do not establish memory ordering (Section 2.2.1.6 and Section 2.2.1.7).
- Satisfying loads from store buffers (Section 2.2.1.8) and how to prevent this behavior (Section 2.2.1.9).
- Semaphore operations and local bypass (Section 2.2.1.10).
- Global visibility order of memory operations (Section 2.2.1.11 and Section 2.2.1.12).

This presentation is organized to begin with simple behaviors and move to increasingly complex behaviors.

2.2.1.1 Assumptions and Notation

The discussions of the multiprocessor executions in the upcoming sections adopt two main notational conventions.

First, the memory accesses in the executions in this document are written using a pseudo-Itanium-based assembly language that allows a store to write an immediate operand to memory. All memory locations are cacheable and aligned. Unless stated otherwise, memory locations do not overlap. Initially, all registers and memory locations contain zero.

Second, given two different memory operations X and Y, $X \gg Y$ specifies that X precedes Y in program order and $X \rightarrow Y$ indicates that X is visible if Y is visible (i.e. X becomes visible before Y).

Using this notation, Figure 2-1 expresses the Itanium ordering semantics from Section 2.1.1 on page 2:375 and also Section 4.4.7, “Memory Access Ordering” in Volume 1. There are no implications regarding the ordering of the visibility for the following pairs of operations: a release followed by an unordered operation; a release followed by an acquire; an unordered operation followed by another; or an unordered operation followed by an acquire.

Figure 2-1. Intel® Itanium™ Ordering Semantics

$$\begin{gathered}
\text{Acquire} \gg X \Rightarrow \text{Acquire} \rightarrow X \\
X \gg \text{Release} \Rightarrow X \rightarrow \text{Release} \\
X \gg \text{Fence} \Rightarrow X \rightarrow \text{Fence} \\
\text{Fence} \gg Y \Rightarrow \text{Fence} \rightarrow Y
\end{gathered}$$

In Figure 2-1, “Acquire”, “Release”, and “Fence” represent an orderable instruction with the corresponding memory ordering semantics whereas “X” and “Y” indicate any orderable instruction.

2.2.1.2 The Intel® Itanium™ Architecture Provides a Relaxed Ordering Model

The Itanium memory ordering model is a relaxed model. As a result, the Itanium architecture permits any outcome when executing the code shown in Table 2-1.

Table 2-1. Intel® Itanium™ Architecture Provides a Relaxed Ordering Model

Processor #0			Processor #1		
st	[x] = 1	// M1	ld	r1 = [y]	// M3
st	[y] = 1	// M2	ld	r2 = [x]	// M4

Outcomes: all are allowed

Because all of the operations in Table 2-1 are unordered, the Itanium memory ordering model does not place any constraints on the order in which a processor based on the Itanium architecture makes the operations visible.

Observing a particular value in r2, for example, does not allow any inferences to be made about the value of r1 because the pair of stores on Processor #0 may become visible in any order. Therefore, all outcomes are possible as the system may interleave M1, M2, M3, and M4 in any order without violating the memory ordering constraints.

2.2.1.3 Enforcing Basic Ordering

Using acquire and release ordering semantics enforces an ordering between both the Processor #0 operations M1 and M2 and the Processor #1 operations M3 and M4 from the Table 2-1 execution as shown in Table 2-1.

Table 2-2. Acquire and Release Semantics Order Intel® Itanium™ Memory Operations

Processor #0			Processor #1		
st	[x] = 1	// M1	ld.acq	r1 = [y]	// M3
st.rel	[y] = 1	// M2	ld	r2 = [x]	// M4

Outcome: only r1 = 1 and r2 = 0 is not allowed

The Itanium ordering model only disallows the outcome r1 = 1 and r2 = 0 in this execution. The release semantics on M2 and acquire semantics on M3 affect the following ordering constraints:

$$\mathrm{M1} \rightarrow \mathrm{M2}$$
$$\mathrm{M3} \rightarrow \mathrm{M4}$$

Given the code in Table 2-2, these two ordering constraints along with the assumption that the outcome is r1 = 1 and r2 = 0 together imply that:

$$r1 = 1 \Rightarrow \mathrm{M2} \rightarrow \mathrm{M3} \Rightarrow \mathrm{M1} \rightarrow \mathrm{M4} \text{ (because M1} \rightarrow \text{M2 and M3} \rightarrow \text{M4)} \Rightarrow r2 = 1$$

This contradicts the postulated outcome r1 = 1 and r2 = 0 and thus the Itanium ordering model disallows the r1 = 1 and r2 = 0 outcome.

In operational terms, if Processor #1 observes M2, the release store to y (i.e. r1 is 1), it must have also observed M1, the unordered store to x (i.e. r2 is 1 as well), given the ordering constraints. Therefore, the Itanium ordering model must disallow the outcome r1 = 1 and r2 = 0 in this execution as this outcome violates these constraints.

Stronger ordering models that do not relax load-to-load and store-to-store ordering, such as sequential consistency, impose these same ordering constraints on M1, M2, M3, and M4 and therefore also do not allow the outcome r1 = 1 and r2 = 0.

2.2.1.4 Allow Loads to Pass Stores to Different Locations

The Itanium memory ordering model allows loads to pass stores as shown in the execution sequence in Table 2-3. Permitting this behavior can improve performance by allowing the processor to complete loads that follow a store that misses the cache.

The Itanium ordering semantics always allow a processor to make operations that follow a release visible before the release and to make operations that precede an acquire visible after the acquire.

Table 2-3. Loads May Pass Stores to Different Locations

Processor #0			Processor #1		
st.rel	[x] = 1	// M1	st.rel	[y] = 1	// M3
ld.acq	r1 = [y]	// M2	ld.acq	r2 = [x]	// M4

Outcomes: all are allowed

Like the execution shown in Table 2-1, the Itanium memory ordering model does not place any constraints on the ordering of the operations on each processor in this execution either.

Therefore, for reasons similar to those given in Section 2.2.1.2 for the execution shown in Table 2-1, the Itanium memory ordering model allows any outcome in this execution as well. Further, the Itanium memory ordering model also allows all outcomes in similar executions that differ only in the ordering semantics of the load and store operations (e.g. those that replace M1 with an unordered store, etc.). There is no combination of legal ordering semantics on these operations (recall that the Itanium instruction set does not provide stores with acquire or fence semantics) that enforce either $M1 \rightarrow M2$ or $M3 \rightarrow M4$.

2.2.1.5 Preventing Loads from Passing Stores to Different Locations

The only way to prevent the loads from moving ahead of the stores in the Table 2-3 execution is to separate them with a memory fence as the execution in Table 2-4 illustrates.

Table 2-4. Loads May Not Pass Stores in the Presence of a Memory Fence

Processor #0			Processor #1		
st	[x] = 1	// M1	st	[y] = 1	// M4
mf		// M2	mf		// M5
ld	r1 = [y]	// M3	ld	r2 = [x]	// M6

Outcome: only r1 = 0 and r2 = 0 is not allowed

The Itanium memory ordering model only disallows the outcome r1 = 0 and r2 = 0 in this execution. The memory fences on Processor #0 and Processor #1 (operations M2 and M5) force the load and store memory accesses to be made visible in program order; no re-ordering is permitted across the fence. Thus, the following ordering constraints must be met:

$$M1 \rightarrow M2 \rightarrow M3$$
$$M4 \rightarrow M5 \rightarrow M6$$

Given the code in Table 2-4, these two constraints along with the assumption that the outcome is r1 = 0 and r2 = 0 together imply that

$$r1 = 0 \Rightarrow M3 \rightarrow M4 \Rightarrow M3 \rightarrow M6 \text{ because } M4 \rightarrow M5 \rightarrow M6$$
$$r1 = 0 \Rightarrow M1 \rightarrow M3 \text{ because } M1 \rightarrow M2 \rightarrow M3$$
$$M1 \rightarrow M3 \text{ and } M3 \rightarrow M6 \Rightarrow M1 \rightarrow M6 \Rightarrow r2 = 1$$

This contradicts the postulated outcome r1 = 0 and r2 = 0 and thus the Itanium memory ordering model disallows the r1 = 1 and r2 = 0 outcome. Specifically, if M3 reads 0, then M4, M5, and M6 may not yet be visible but M1 and M2 must be visible. Thus, when M6 becomes visible it must observe x = 1 because M1 is already visible.

2.2.1.6 Data Dependency Does Not Establish MP Ordering

The dependency rules define the relationship between memory operations that access the same address. Specifically, the Itanium architecture resolves read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies through memory in program order on the local processor. As Section 2.2 discusses, dependencies are fundamentally different from the ordering semantics even though both affect ordering relationships between groups of memory accesses.

The execution shown in Table 2-5 illustrates this difference.

Table 2-5. Dependencies Do Not Establish MP Ordering (1)

Processor #0	Processor #1

st [x] = 1 ;; // M1
ld r1 = [x] ;; // M2
st [y] = r1 ;; // M3
	ld.acq r2 = [y] // M4
	ld r3 = [x] // M5

Outcomes: r1 = 1, r2 = 1, and r3 = 0 is allowed

The following discussion focuses on the outcome r1 = 1, r2 = 1, and r3 = 0. This outcome is allowed only because the Itanium architecture treats data dependencies and the ordering semantics differently.

The ordering semantics require $M4 \rightarrow M5$, but do not place any constraints on the relative order of operations M1, M2, or M3. Due to the register and memory dependencies between the instructions on Processor #0, these operations complete *in program order* on Processor #0 and also become *locally* visible in this order. However, the operations need *not* be made visible to remote processors in program order. In this outcome it appears to Processor #0 as if $M1 \rightarrow M3$ while to Processor #1 it appears that $M3 \rightarrow M1$. There are two things to note here. First, the behavior is another example of the local bypass behavior that Section 2.2.1.8 presents on page 2:385. Second, there are no dependencies *directly* between M1 and M3 that requires them to become globally visible in program order.

Note: All processors will observe the order established by a particular processor in case of a WAW memory dependency to the same location. For example, all processors in the coherence domain eventually see a value of 1 in location x in the following code:

```
  st      [x] = 0        //  M1: set [x] to 0
  st      [x] = 1        //  M2: set [x] to 1, cannot move above M1 due to
WAW
```

because there is a WAW memory dependency between from M2 to M1 and the Itanium architecture requires that the local processor resolves RAW, WAR, and WAW dependencies between its memory accesses in program order. Thus, $M1 \rightarrow M2$ even though the ordering semantics do not place any constraints on the relative ordering of M1 and M2.

2.2.1.7 Data Dependency Establishes Local Ordering

In the Itanium architecture, a dependency (e.g., a later operation reading the value written by an earlier operation) can imply a local ordering relationship between the two operations. This section focuses on dependencies through registers only. Section 2.2.1.6 discusses dependencies and MP ordering.

The execution shown in Table 2-6 illustrates how data dependency and memory ordering interact in a simple “pointer chase”.

Table 2-6. Memory Ordering and Data Dependency

Processor #0	Processor #1

st [x] = 1 // M1
st.rel [y] = x // M2
	ld r1 = [y] ;; // M3
	ld r2 = [r1] // M4

Outcome: r1 = x and r2 = 0 is not allowed

In this example, Processor #0 could be executing code that updates a shared object with M1 and then publishes a pointer to the object with M2. Processor #1 then loads the pointer and dereferences it to read the contents of the shared object. The outcome r1 = x and r2 = 0 implies that Processor #1 observes the new value of the object pointer, y, but the old value of the data field, x.

The ordering semantics require $M1 \rightarrow M2$ but place no requirements on the relative ordering of M3 and M4.

Thus, the memory semantics alone would allow the outcome r1 = x and r2 = 0 in the absence of other constraints. Using an acquire load for M3 can avoid this outcome as doing so forces $M3 \rightarrow M4$ and thus prevents the outcome. However, this use of acquire is non-intuitive given the RAW dependency through register r1 between M3 and M4. That is, M3 produces a value that M4 requires in order to execute so how should it be possible for them to go out of order? Further, using an acquire in this case prevents any memory operation following M3 from moving above M3, even if they are completely independent of M3.

To avoid this potential confusion and performance issue, the Itanium architecture treats data dependency and memory ordering in the same fashion on the local processor. That is, if $A » B$ and A produces a value that B consumes, then $\rightarrow B$ on the local processor. This relationship is also transitive as the execution in Table 2-7 illustrates.

Table 2-7. Memory Ordering and Data Dependency Through a Predicate Register

Processor #0			Processor #1			
st	[x] = 1	// M1		ld	r1 = [y]	// M3
st.rel	[y] = x	// M2		cmp.eq	p1, p2 = r1, x ;;	// C1
			(p1)	ld	r2 = [x]	// M4

Outcome: r1 = x and r2 = 0 is not allowed

The Processor #0 code is the same as in Table 2-6. The Processor #1 now performs the following operation: if the pointer value y is equal to x, load a value from x.

The Itanium architecture does not allow the outcome r1 = x and r2 = 0 in this execution either. Unlike the execution in Table 2-6, there is no *direct* dependency between the values that M3 produces and the values that M4 consumes. However, there is a RAW through register r1 from M3 to C1 and a RAW through register p1 from C1 to M4. Thus, by transitivity, $M3 \rightarrow M4$.

The execution in Table 2-8 illustrates a similar construct but introduces a control dependency.

Table 2-8. Memory Ordering and Data and Control Dependencies

Processor #0			Processor #1			
st	[x] = 1	// M1		ld	r1 = [y] ;;	// M3
st.rel	[y] = x	// M2		cmp.eq	p1, p2 = r1, x	// C1
			(p2)	br	t	// B1
				ld	r2 = [x]	// M4
			t:			

Outcome: r1 = x and r2 = 0 is not allowed

This execution is semantically the same as the execution in Table 2-7; however, this execution uses a control dependency rather than predication to conditionally execute M4. As a result, the outcome r1 = x and r2 = 0 is not allowed in the Table 2-8 execution.

The execution of the load M4 is data-dependent on the value of p2 that the branch B1 uses to resolve. Further, p2 is dependent on the value of r1 that the load M3 produces through the compare C1. Thus, $\mathrm{M3} \rightarrow \mathrm{M4}$.

The execution in Table 2-9 is a variation on the execution from Table 2-8 where the loads are truly independent.

Table 2-9. Memory Ordering and Control Dependency

Processor #0	Processor #1
st [x] = 1 // M1	ld r1 = [y] // M3
st.rel [y] = x // M2	cmp p1, p2 = r3, x // C1
	(p2) br t // B1
	ld r2 = [x] // M4
	t:

Outcome: all are allowed

In this execution, there is no dependency between M3 and M4, and thus, there are no constraints on the relative ordering of M3 and M4. Like the execution in Table 2-8, M4 is data-dependent on the value of p2 that the branch B1 uses to resolve. However, p2 is *independent* of the value that the load M3 produces (specifically, because the compare does not use the value of register r1 that the load produces). Thus, there is no chain of dependencies between M3 and M4 and therefore there are no constraints on the relative ordering of M3 and M4. As a result, all outcomes are allowed in this execution.

2.2.1.8 Store Buffers May Satisfy Local Loads

In the Itanium memory ordering model, store buffers (or other logically-equivalent structures) may satisfy local read requests from loads or acquire loads even if the stored data is not yet visible to other agents in the coherency domain. Such bypassing must honor any ordering semantics in the memory reference stream. Table 2-10 and Table 2-11 that Section 2.2.1.9 presents illustrate this behavior.

Table 2-10. Store Buffers May Satisfy Loads if the Stored Data is Not Yet Globally Visible

Processor #0	Processor #1
st.rel [x] = 1 // M1	st.rel [y] = 1 // M4
ld.acq r1 = [x] // M2	ld.acq r3 = [y] // M5
ld r2 = [y] // M3	ld r4 = [x] // M6

Outcome: r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is allowed

In this sequence, each processor bypasses its locally-written value from a store buffer before the value becomes visible to the other processor. This behavior may make accesses of different sizes that have overlapping memory addresses appear to complete non-atomically.

The following discussion focuses on the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 because this outcome is allowed if and only if store buffers can satisfy local loads (other outcomes are allowed but do not depend on being able to satisfy local loads from a store buffer).

The Itanium memory ordering semantics only require that $\mathrm{M2} \rightarrow \mathrm{M3}$ and $\mathrm{M5} \rightarrow \mathrm{M6}$. There are no constraints on the relative ordering of M1 and M2 or M3 nor on the relative ordering of M4 and M5 or M6.

Remember that both dependencies and the memory ordering model place requirements on the manner in which a processor based on the Itanium architecture may re-order accesses. Even though the Itanium memory ordering model allows loads to pass stores, a processor based on the Itanium architecture cannot re-order the following sequence:

```
st.rel      [x] = r0        // M1: store 0 to [x]
ld.acq      r1 = [x]        // M2: cannot move above st.rel due to RAW
```

This is because there is a RAW dependency through memory between M1 and M2 and the Itanium memory ordering model requires that the local processor resolve RAW, WAR, and WAW dependencies between its memory accesses in program order. Thus, $M1 \rightarrow M2$ even though the ordering semantics place no constraints on the relative ordering of M1 and M2.

Because there is a RAW dependency through memory between M1 and M2 and between M4 and M5, the ordering constraints *effectively* become:[1]

$$M1 \rightarrow M2 \rightarrow M3$$
$$M4 \rightarrow M5 \rightarrow M6$$

to account for both the memory ordering semantics and dependencies. It is important to keep in mind that the observance of a dependency between two operations does not imply an ordering relationship (from the standpoint of the memory ordering model) between the operations as Section 2.2.1.6 describes.

Assuming that a processor can bypass locally-written values before they are made globally-visible implies that there is a local and a global visibility points for a memory operation where a value always becomes locally visible before it becomes globally visible. Since M1 and M4 can have local visibility with respect to M2 and M5 as well as global visibility,

$$m1 \rightarrow M2 \rightarrow M3;\ m1 \rightarrow M1$$
$$m4 \rightarrow M5 \rightarrow M6;\ m4 \rightarrow M4$$

where m1 and M1 represent local and global visibility of memory operation 1, respectively. There are two things to note. First, the ordering of the local visibilities of operations M1 and M4 (m1 and m4, respectively) allow each processor to honor its data dependencies. That is, Processor #2 honors the RAW dependency through memory between M1 and M2 by requiring m1 to become visible before M2. Second, that these requirements do not place any constraints on the relative ordering perceived by a *remote* observer of operation M1 with M2 and M3 or of operation M4 with M5 and M6 (as the local visibilities meet the *local* ordering constraints that the dependencies impose).

The code in Table 2-10 and these constraints together imply that

$$r1 = 1 \Rightarrow m1 \rightarrow M2$$
$$r3 = 1 \Rightarrow m4 \rightarrow M5$$
$$r2 = 0 \Rightarrow M3 \rightarrow M4 \Rightarrow m1 \rightarrow M6 \text{ because } m1 \rightarrow M3 \text{ and } M3 \rightarrow M4 \text{ and } M4 \rightarrow M6$$
$$r4 = 0 \Rightarrow M6 \rightarrow M1$$
$$m1 \rightarrow M6 \text{ and } M6 \rightarrow M1 \Rightarrow m1 \rightarrow M1$$

Thus, the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is allowed because these statements are consistent with our definition of local and global visibility. Specifically, a value becomes locally visible before it becomes globally visible. Similar reasoning can show that the constraints also imply that $m4 \rightarrow M4$.

[1] That is, the store operations must become visible to the local processors before their loads that read the stored value.

2.2.1.9 Preventing Store Buffers from Satisfying Local Loads

In the code shown in Table 2-10 from Section 2.2.1.8, there are no ordering constraints between the store and acquire load from the standpoint of memory ordering semantics (however, there is a RAW dependency through memory that forces the acquire load to follow the store). Bypassing may not occur if doing so violates the memory ordering constraints of memory operations between the store and the bypassing read. Table 2-11 presents a variation on the execution in Table 2-10 from Section 2.2.1.8 that illustrates this behavior.

Table 2-11. Preventing Store Buffers from Satisfying Local Loads

Processor #0			Processor #1		
st	[x] = 1	// M1	st	[y] = 1	// M5
mf		// M2	mf		// M6
ld.acq	r1 = [x]	// M3	ld.acq	r3 = [y]	// M7
ld	r2 = [y]	// M4	ld	r4 = [x]	// M8

Outcome: r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is not allowed

Like Section 2.2.1.8, the discussion in this section focuses on the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 because it is allowed if and only if store buffers can satisfy local loads. The line of reasoning to show that the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is not allowed in Table 2-11 is similar to the reasoning used to show that this outcome is allowed in the Table 2-10 execution from Section 2.2.1.8 on page 2:385.

By the definition of the Itanium memory ordering semantics,

$$\begin{gathered} \mathrm{M1} \rightarrow \mathrm{M2} \rightarrow \mathrm{M3} \rightarrow \mathrm{M4} \\ \mathrm{M5} \rightarrow \mathrm{M6} \rightarrow \mathrm{M7} \rightarrow \mathrm{M8} \end{gathered}$$

By allowing local and global visibility of operations M1 and M5 (similar to the discussion in Section 2.2.1.8), this assumption, along with the above constraints, together imply that,

$$\begin{gathered} \mathrm{m1} \rightarrow \mathrm{M1} \Rightarrow \mathrm{m1} \rightarrow \mathrm{M2} \rightarrow \mathrm{M3} \rightarrow \mathrm{M4} \\ \mathrm{m5} \rightarrow \mathrm{M5} \Rightarrow \mathrm{m5} \rightarrow \mathrm{M6} \rightarrow \mathrm{M7} \rightarrow \mathrm{M8} \end{gathered}$$

Consider these constraints on the Processor #0 operations m1, M1, M2, M3, and M4. Making m1 visible before M2, M3, and M4 correctly honors the data dependency through memory on Processor #0. However, unless it constrains the global visibility of M1 to occur before M2, M3, and M4, Processor #0 violates the Itanium ordering semantics. Specifically, the memory fence M2 must always be made visible after the store M1. Allowing global and local visibilities of M1 in this case violates this constraint, and thus, is not allowed. Essentially, by allowing M1 to become locally visible early, M3 would see M1 before the fence semantics for M2 were met (namely, that M1 be visible before M2 and thus M3). Without local and global visibility of M1 and M5, the ordering constraints are as this example originally postulated.

The code in Table 2-11 and these constraints together imply that

$$\mathrm{r2} = 0 \ \Rightarrow \mathrm{M4} \rightarrow \mathrm{M5} \Rightarrow \mathrm{M1} \rightarrow \mathrm{M8} \text{ because } \mathrm{M1} \rightarrow \mathrm{M4} \text{ and } \mathrm{M4} \rightarrow \mathrm{M5} \text{ and } \mathrm{M5} \rightarrow \mathrm{M8} \Rightarrow \mathrm{r4} = 1$$

This contradicts the r1 = 1, r3 = 1, r2 = 0, and r4 = 0 outcome. The visibility of the memory fence, M2, implies that all prior operations including the store to x, M1, are globally visible. Thus, the load from x on Processor #1, M8, must observe the new value of x and $\mathrm{M1} \rightarrow \mathrm{M8}$ but the outcome requires $8 \rightarrow \mathrm{M1}$.

2.2.1.10 Semaphores Do Not Locally Bypass

As Section 2.2.1.8 and Section 2.2.1.9 discuss, loads and acquire loads may be satisfied with values placed in local store buffers (or other logically-equivalent structures) by stores or release stores before the stored data becomes visible to other agents in the coherency domain. The Itanium architecture explicitly prohibits such local bypass either to or from semaphore operations. That is, semaphore operations cannot be satisfied in this way nor can the data they store be used to satisfy loads or acquire loads in this way.

The execution in Table 2-12 illustrates a variation on the execution in Table 2-10 where the acquire loads have been replaced with exchange semaphore operations (which also have acquire semantics).

Table 2-12. Bypassing to a Semaphore Operation

Processor #0			Processor #1		
st.rel	[x] = 1	// M1	st.rel	[y] = 1	// M4
xchg	r1 = [x], r5	// M2	xchg	r3 = [y], r6	// M5
ld	r2 = [y]	// M3	ld	r4 = [x]	// M6

Outcome: r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is not allowed

Although each semaphore operation can be decomposed into a read access followed by a write access, the Itanium architecture does *not* allow a read request by a semaphore to be satisfied from a store buffer (or other logically-equivalent structure). As a result, the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is not allowed. The reasoning is similar to that presented in Section 2.2.1.9.

Specifically, by the definition of the Itanium memory ordering semantics, $M2 \rightarrow M3$ and $M5 \rightarrow M6$. The relative ordering between operation M1 and operations M2 or M3 is not constrained. Likewise, the relative ordering between operation M4 and operations M5 and M6.

Now, assume the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0. Given that r1 = 1, r3 = 1, and r2 = 0, we observe the following:

$$
\begin{gathered}
r1 = 1 \Rightarrow M1 \rightarrow M2 \\
r3 = 1 \Rightarrow M4 \rightarrow M5 \\
r2 = 0 \Rightarrow M3 \rightarrow M4 \\
M3 \rightarrow M4 \Rightarrow M1 \rightarrow M6 \text{ because } M1 \rightarrow M3 \rightarrow M4 \rightarrow M6 \\
M1 \rightarrow M6 \Rightarrow r4 = 1
\end{gathered}
$$

This conclusion contradicts the assumed outcome where r4 = 0 and thus the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is not allowed. Because M1 and M4 cannot become locally-visible to M2 and M5 before they become globally-visible to M6 and M3 (as read accesses from semaphores may not bypass from store buffers or other logically-equivalent structures), it is not possible to avoid this contradiction.

The Itanium architecture also prohibits local bypass from a semaphore operation to a local read access from a load or acquire load as shown in the execution in Table 2-13.

Table 2-13. Bypassing from a Semaphore Operation

Processor #0			Processor #1		
fetchadd.rel	r5 = [x], 1	// M1	fetchadd.rel	r6 = [y], 1	// M4
ld.acq	r1 = [x]	// M2	ld.acq	r3 = [y]	// M5
ld	r2 = [y]	// M3	ld	r4 = [x]	// M6

Outcome: r1 = 1, r3 = 1, r2 = 0, r4 = 0, r5 = 0, and r6 = 0 is not allowed

A store buffer may not provide a local read operation early access to a value written by a semaphore operation. Therefore, the outcome r1 = 1, r3 = 1, r2 = 0, r4 = 0, r5 = 0, and r6 = 0 in the Table 2-13 execution is not allowed. The reasoning is similar to that used in the previous execution.

2.2.1.11 Ordered Cacheable Operations are Seen in the Same Order by All Observers

The Itanium memory ordering model requires that release stores and semaphore operations (both acquire and release forms) become visible to all observers in the coherence domain in a single total order with the exception that each processor may observe (via loads or acquire loads) its own update early. Thus, each observer in the coherence domain sees the same interleaving of release stores and semaphores (both acquire and release forms) from the other processors in the coherence domain except that each processor may observe its own release stores (via loads or acquire loads) prior to their being observed globally. Table 2-14 illustrates this behavior.

Table 2-14. Enforcing the Same Visibility Order to All Observers in a Coherency Domain

Processor #0	Processor #1	Processor #2	Processor #3
st.rel [x] = 1 // M1	ld.acq r1 = [x] // M2 ld r2 = [y] // M3	st.rel [y] = 1 // M4	ld.acq r3 = [y] // M5 ld r4 = [x] // M6

Outcome: only r1 = 1, r3 = 1, r2 = 0, and r4 =0 is not allowed

The Itanium memory ordering model only disallows the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 in this execution. By the definition of the Itanium memory ordering semantics,

$$M2 \rightarrow M3$$
$$M5 \rightarrow M6$$

The Itanium memory ordering model does not permit the r1 = 1, r3 = 1, r2 = 0, and r4 = 0 outcome as this would require that Processors #1 and #3 observe the release stores to x and y in different orders. Specifically, assuming that the outcome is r1 = 1, r3 = 1, r2 = 0, and r4 = 0:

$$r1 = 1 \Rightarrow M1 \rightarrow M2$$
$$r3 = 1 \Rightarrow M4 \rightarrow M5$$
$$r2 = 0 \Rightarrow M3 \rightarrow M4 \Rightarrow M1 \rightarrow M4 \text{ because } M1 \rightarrow M2,\ M2 \rightarrow M3, \text{ and } M3 \rightarrow M4$$
$$r4 = 0 \Rightarrow M6 \rightarrow M1 \Rightarrow M4 \rightarrow M1 \text{ because } M4 \rightarrow M5,\ M5 \rightarrow M6, \text{ and } M6 \rightarrow M1$$

The final two statements are inconsistent since both $M1 \rightarrow M4$ and $M4 \rightarrow M1$ cannot be true unless Processors #1 and #3 are allowed to see the release stores to x and y in different orders.

The Itanium memory ordering model allows the r1 = 1, r3 = 1, r2 = 0, and r4 = 0 outcome if either one or both of the release stores M1 and M4 are unordered since unordered operations need not be seen in the same total order by all observers in the coherence domain. Thus, in a version of the execution shown in Table 2-14 with unordered stores, Processor #2 observes $\mathrm{M1} \rightarrow \mathrm{M4}$ while Processor #4 observes $\mathrm{M4} \rightarrow \mathrm{M1}$.

The Itanium memory ordering model also allows this outcome if the release stores M1 and M4 are replaced with a memory fence followed by an unordered store. From the standpoint of a single processor, a release store has equivalent ordering semantics on the local processor to a memory fence followed by an unordered store. However, because the store in the memory fence/unordered store pair is unordered, it does not have any ordering requirements with respect to a remote processor. Even when processors are allowed to construct different interleavings, the ordering of an individual processor's memory references within the interleaving must always respect the ordering constraints placed on those references.

2.2.1.12 Obeying Causality

As noted in Section 2.2.1.11, the Itanium memory ordering model requires that release stores and semaphore operations (both acquire and release forms) become visible to all observers in the coherence domain in a single total order with the exception that each processor may observe (via loads or acquire loads) its own update early. Thus, each observer in the coherence domain sees the same interleaving of release stores, and semaphores operations from the other processors in the coherence domain.

A consequence of this is the fact that the Itanium memory ordering model respects causality in a certain way. Specifically, if a release store or semaphore operation causally precedes any store or semaphore operation, then the two operations will become visible to all processors in the causality order. Table 2-1 illustrates this behavior. Suppose that M2 reads the value written by M1. In this case, there is a causal relationship from M1 to M3 (a control dependency could also establish such a relationship). The fact that the store to x is a release store implies that, since there is a causal relationship from M1 to M3, M1 must become visible to processor #2 before M3.

Table 2-15. Intel® Itanium™ Architecture Obeys Causality

Processor #0			Processor #1			Processor #2		
st.rel	[x] = 1	// M1	ld.acq	r1 = [x]	// M2	ld.acq	r2 = [y]	// M4
			st	[y] = 1	// M3	ld	r3 = [x]	// M5

Outcome: only r1 = 1, r2 = 1, and r3 = 0 is not allowed

The Itanium memory ordering model disallows the outcome r1 = 1, r2 = 1, and r3 = 0 in this execution (all other outcomes are allowed). To see this, we note the following. If r1 = 1, then $\mathrm{M1} \rightarrow \mathrm{M2}$ at Processor #1. Because M2 is an acquire load and $\mathrm{M2} \gg \mathrm{M3}$, $\mathrm{M2} \rightarrow \mathrm{m3}$, where m3 represents the local visibility of memory operation 1 (see Section 2.2.1.8). Thus, $\mathrm{M1} \rightarrow \mathrm{m3}$. Since M1 is a release store, it appears to become visible to all processors at the same time. This fact and $\mathrm{m3} \rightarrow \mathrm{M3}$ together imply $\mathrm{M1} \rightarrow \mathrm{M3}$.

If r2 = 1, $\mathrm{M3} \rightarrow \mathrm{M4}$. Because M4 is an acquire load, $\mathrm{M4} \rightarrow \mathrm{M5}$. If r3 = 0, then $\mathrm{M5} \rightarrow \mathrm{M1}$. Together, these imply $\mathrm{M3} \rightarrow \mathrm{M1}$, which contradicts the observation from the previous paragraph. Thus, the outcome r1 = 1, r2 = 1, and r3 = 0 is disallowed.

The indicated outcome would also be disallowed if M1 were a semaphore operation because, like release stores, each semaphore must appear to become visible at all processors at the same time. The indicated outcome would be allowed if M1 were a weak store, as a weak store may appear to become visible at different times to different processors.

2.2.2 Memory Attributes

In addition to the ordering semantics and data dependencies, the memory attributes of the page that is being referenced also influence access ordering and visibility. Using memory attributes allows the Itanium architecture to match the performance and the usage model to the type of device (e.g. main memory, memory-mapped I/O device, frame buffer, locations with side-effects, etc.) that backs a page of memory. Typically, memory with side-effects is mapped uncacheable while memory without side-effects is mapped as write-back cacheable.

Section 4.4, "Memory Attributes" describes memory attributes in the Itanium architecture in greater depth.

Memory with the uncacheable UC or UCE attributes is sequential by definition. A processor based on the Itanium architecture ensures that accesses to sequential memory locations reach a peripheral domain (a platform-specific collection of uncacheable locations, colloquially known as "a device") in program order with respect to all other accesses to sequential locations in the same peripheral domain. The sequential behavior of UC or UCE memory is independent of the ordering semantics (i.e. acquire, release, fence, or unordered) attached to the accesses.

Other observers (e.g. processors or other peripheral domains) need not see references to UC or UCE memory in sequential order if at all. When multiple agents are writing to the same device, it is up to software to synchronize the accesses to the device to ensure the proper interleaving.

The ordering semantics of an access to sequential memory determines how the access becomes visible to the peripheral domain with respect to other operations. For example, consider the code sequence shown in Figure 2-2.

Figure 2-2. Interaction of Ordering and Accesses to Sequential Locations

```
sequential_example:
           st       [data_0] = 0     // M1: put data in cacheable mem
           st       [data_1] = 0     // M2: put data in cacheable mem
           st.rel   [ready] = 1      // M3: tell device to get ready
           st       [start] = 1      // M4: tell device to start
```

In this code, assume that `data_0` and `data_1` are cacheable locations and `start` and `ready` are an uncacheable UC or UCE locations.

Sequentiality ensures that M3 and M4 reach the peripheral domain in program order (i.e. M3 before M4). Further, the release semantics on M3 ensures that it is not made visible to the peripheral domain until after M1 and M2 are made visible to the coherence domain. The M1 and M2 accesses may become visible to the coherency domains in any order as they both have unordered semantics. Even though the memory ordering semantics allow M4 to become visible before M3, the processor must make M3 visible before M4 because both `ready` and `start` are sequential locations.

2.2.3 Understanding Other Ordering Models: Sequential Consistency and IA-32

To provide a point of reference, it is helpful to understand other memory ordering models. These ordering models affect not only the programmer's view of the system, but also the overall system performance and design. Processors with relaxed memory ordering models may achieve higher performance than those with strict ordering models.

The most intuitive memory ordering model is "sequential consistency" (SC) which Lamport formally defines in [L79]. In sequential consistency, all processors see the memory references from a given processor in program order, and, in addition, all processors see the same system-wide interleaving of memory references from each processor.

The SC model precludes many common optimizations made in modern microprocessors to enhance performance. For example, in an SC system, a load may not pass a prior store until that store becomes globally visible (because all memory operations must become visible in program order). This requirement prevents the SC system from using a store buffer to hide the latency of store traffic by allowing loads that hit the cache to be serviced under a prior store that miss the cache.

To address such performance issues, many memory ordering models have been developed that relax the constraints of sequential consistency. Adve categorizes these memory models by noting how they relax the ordering requirements between reads and writes and if they allow writes to be read early [AG95]. The Itanium architecture allows for relaxed ordering between reads and writes and also allows writes to be read early under certain circumstances.

Aside from disallowing any relaxation of memory references, sequential consistency has two other subtle differences from the Itanium memory ordering model. First, it requires a total order of operations whereas the Itanium memory ordering model only requires a total order for release stores and semaphores. Second, remote processors must always honor data dependencies since the local processor does not have the option of re-ordering such accesses as can occur.

The IA-32 memory ordering relaxes write to read ordering and allows a processor to read its own writes before they are globally visible. Further, IA-32 allows each processor in the coherence domain to interleave the reference streams from other processors in the coherence domain in a different order. The per-processor orders must meet some additional constraints to ensure they are consistent with each other (enumerating and explaining these constraints is beyond the scope of this document). For more information on the IA-32 ordering model see Section 6.3.2, "IA-32 Segmentation" in Volume 1.

2.3 Where the Intel® Itanium™ Architecture Requires Explicit Synchronization

The Itanium architecture requires a memory synchronization (`sync.i`) and a memory fence (`mf`) during a context switch to ensure that all memory operations prior to the context switch are made visible before the context changes. Without this requirement, the ordering constraints may be violated if the process migrates to a different processor. For example, consider the example shown in Figure 2-3

Figure 2-3. Why a Fence During Context Switches is Required in the Intel® Itanium™ Architecture

```
// Process A begins executing on Processor #0...

            ld.acq    r1 = [x]         // load executes on processor #0

// 1) Context switch occurs
// 2) O/S migrates Process A from Processor #0 to Processor #1
// 3) Process A resumes at the instruction following the ld.acq

            st        [y] = r2         // store executes on processor #1
```

In this example, Processor #1 may make the unordered store visible to the coherency domain before Processor #0 makes the acquire load visible. This violates the ordering constraints. Executing a memory fence during the context switch handler ensures that this violation can not occur.

See Section 4.5, "Context Switching" on page 2:422 on context management in a processor based on the Itanium architecture.

Interruptions do not affect memory ordering. On entry to an interrupt handler, memory operations from the interrupted program may still be in-flight and not yet visible to other processors in the coherence domain. A handler that expects that all memory operations that precede the interruption to be visible must enforce this requirement by executing a memory fence at the beginning of the handler.

2.4 Synchronization Code Examples

There are many synchronization primitives that software uses in multiprocessor or multi-threaded environments to coordinate the activities of different code streams. In this section, we present several typical examples to illustrate how some common constructs translate to the Itanium instruction set. In addition, the discussions identify special considerations with various implementations.

The examples use the syntax "[foo]" to indicate the memory location that holds the variable foo. Actual Itanium-based assembly language would first move the address of foo into a register and then use this register as an operand to a memory access instruction. The alternate syntax is chosen to simplify and clarify the examples.

2.4.1 Spin Lock

Software commonly uses spin locks to guard access to a critical region of code. In these locks, the software "spins" while waiting for a shared lock variable to indicate that the critical region can be safely accessed. Typically, the lock code uses atomic operations such as compare and exchange or fetch and add to update the shared lock variable. Figure 2-4 shows a spin lock based on the cmpxchg instruction.

Figure 2-4. Spin Lock Code

```
// available. If it is 1, another process is in the critical section.
//
spin_lock:
            mov    ar.ccv = 0             // cmpxchg looks for avail (0)
            mov    r2 = 1                 // cmpxchg sets to held (1)

spin:
            ld8    r1 = [lock] ;;         // get lock in shared state
            cmp.ne p1, p0 = r1, r2        // is lock held (ie, lock ==
1)?
(p1)        br.cond.spnt  spin ;;         // yes, continue spinning

            cmpxchg8.acqr1 = [lock], r2 ;;// attempt to grab lock
            cmp.ne p1, p0 = r1, r2        // was lock empty?
(p1)        br.cond.spnt  spin ;;         // bummer, continue spinning

cs_begin:
            // critical section code goes here...
cs_end:

              st8.rel[lock] = r0 ;;         // release the lock
```

The spin lock code first initializes `ar.ccv` and a register with the values that indicate that the lock is available and held, respectively. A compare and exchange obtains the lock by exchanging `lock` with `1` if it currently holds `0`. Next, the first loop ensures that the code spins in cache while the lock is held by someone else. Once this loop finds that the lock is available, a compare and exchange instruction attempts to obtain the lock. If this instruction fails (e.g. because someone else obtained the lock in the meantime), the code resumes spinning in the first loop.

Spinning using only the `cmpxchg`/`cmp`/`br` loop may generate excessive coherency traffic. For example, if the `cmpxchg` always stores to memory (even if the comparison fails) and the lock is highly-contested, the platform may have to generate a number of read for ownership transactions causing `lock` to move around the system. Using the first `ld8`/`cmp`/`br` loop avoids this problem by obtaining `lock` in a shared state. In the worst case, when `lock` is not contested, this loop adds only the overhead of the additional compare and branch.

The initial `ld8` need not be an acquire load because of the control-flow in the spin loop: this load must become visible before the `cmpxchg8` because the load must return data in order for the compare and branch to resolve. Further, the store that relinquishes the lock after the critical section uses release semantics to prevent memory references from the critical from moving after the reference that releases the lock. Finally, the branches use “static predict not taken” hints to optimize for the case where the lock is not highly contested.

2.4.2 Simple Barrier Synchronization

A barrier is a common synchronization primitive used to hold a set of processes at a particular point in the program (the barrier) until all processors reach the location. Once all processes arrive at the barrier, they may all continue to execute. Figure 2-5 shows a sense-reversing barrier synchronization based on the `fetchadd` instruction from Hennessy and Patterson [HP96].

This type of barrier prevents a process that races ahead to the next instance of the barrier from trapping other (slow) processors that are in the process of leaving the barrier.

Figure 2-5. Sense-reversing Barrier Synchronization Code

```
// The total shared variable is one less than the number of processors
// that wait at the barrier.
// The release shared variable indicates if the processor must wait at
// the barrier (initially, this variable is 0).
// local_sense is a per-processor local variable that indicates the
// "sense" of the barrier (initially, this variable is 0).

sr_barrier:
            fetchadd8.acq r1 = [count], 1// update counter
            ld8       r2 = [total]           // get number of procs - 1
            ld8       r3 = [local_sense] ;;  // get local “sense” variable
            xor       r3 = 1, r3             // local_sense =! local_sense
            cmp.eq    p1, p2 = r1, r2;;      // p1 => last proc to arrive
            st8       [local_sense] = r3     // save new value of local_sense
(p1)        st8       [count] = r0           // last resets count to 0
(p1)        st8.rel   [release] = r3 ;;      // last allows other to leave

wait_on_others:
(p2)        ld8          r1 = [release] ;;   // p2 => more procs to come
(p2)        cmp.ne.and   p0, p2 = r1, r3     // have all arrived yet?
(p2)        br.cond.sptk wait_on_others ;;   // nope, continue waiting

            // This mf prevents memory operations that follow the barrier code
            // from moving ahead of memory operations that precede the barrier
            // code
            mf ;;
```

The barrier code begins by atomically updating the number of processors that are waiting at the barrier, `count`, using a `fetchadd` instruction. For the last processor that reaches the barrier, the `fetchadd` instruction returns the same value as the `total` shared variable, which is one less than the number of processors that wait at the barrier. Other processors each get a unique value on the interval [0, `total`) based on the order in which they arrive at the barrier.

All processors except the last processor wait in the `wait_on_others` loop for the signal that all have arrived at the barrier. The last processor to arrive at the barrier provides this signal.

The signal to leave the barrier is deduced from the value of the `release` shared variable and the `local_sense` local variable. Upon entering the barrier, each processor complements the value in its private `local_sense` variable. Once in the barrier, all processors always have the same value in their `local_sense` variables. This variable indicates the value that `release` must have before the processor can leave the barrier. The last processor to arrive at the barrier releases the other processors by setting `release` to the new `local_sense` value.

The `mf` instruction in Figure 2-5 is necessary only if the programmer wishes to ensure that memory operations performed before the barrier code are visible to memory operations performed by any processor after the barrier code.

2.4.3 Dekker’s Algorithm

Dekker’s algorithm [D65] is a common synchronization construct that arbitrates for a resource through the use of several shared variables that indicate which processor is using the resource. Each processor has its own flag variable that it shares with all other processors in the system. When a processor attempts to enter the critical section, it sets its flag to one and checks to make sure the flags for the other processors are all zero.

The code in Figure 2-6 illustrates the core of this algorithm for a two-way multi-processor system. In this example, a processor makes a single attempt to acquire the resource; typically, this code would appear in a loop. Although there is an array of per-processor flag variables, the code uses `flag_me` and `flag_you` to indicate to the flag variables for the processor attempting to obtain the resource and the other remote processor, respectively.

Dekker’s algorithm assumes a sequential consistency ordering model. Specifically, it assumes that loading zero from `flag_you` implies that a processor’s load and stores to the flag variables occur before the other processor’s load and store to the flag variables. If this is not the case, both processors can enter the critical section at the same time.

Using unordered loads or stores to access the `flag_me` and `flag_you` variables does not guarantee correct behavior as the processor may re-order the accesses as it sees fit. Using an acquire load and release store is also not sufficient to ensure correct behavior because the ordering semantics always allow acquire loads to move earlier and release stores to move later. In the absence of the `mf`, it is possible for the load from `flag_you` to occur before the store to `flag_me`; even with acquire and release operations.

The first `ld8` need not be an acquire load because of the control-flow that skips the critical section: this load must become visible before any memory operations in the critical section because the load must return data in order for the compare and branch to resolve.

Figure 2-6. Dekker’s Algorithm in a 2-way System

```
// The flag_me variable is zero if we are not in the synchronization and
// critical section code and non-zero otherwise; flag_you is similarly
set
// for the other processor. This algorithm does not retry access to the
// resource if there is contention.
//
dekker:
            mov    r1 = 1 ;;                 // my flag = 1 (i want
access!)
            st8    [flag_me] = r1
            mf ;;                            // make st visible first
            ld8    r2 = [flag_you] ;;        // is other’s flag 0?
            cmp.eq p1, p0 = 0, r2
(p1)        br.cond.spnt  cs_skip ;;         // if not, resource in use

cs_begin:
            // critical section code goes here...
cs_end:

cs_skip:
            st8.rel[flag_me] = r0 ;;         // release lock
```

2.4.4 Lamport's Algorithm

Like Dekker's Algorithm, Lamport's Algorithm [L85] also provides mutual exclusion for critical sections of code. Lamport's algorithm is very simple and, in the case of non-contested locks, only requires two read and two write memory accesses to enter the critical section. The algorithm uses two shared variables, x and y, and a shared array, b, that identify the process entering and using the critical section. Figure 2-7 presents Lamport's Algorithm 2 [L85].

Lamport's algorithm expects that a processor that enters the critical section performs the set of operations: S = {store x, load y, store y, load x}[1]. To enforce this ordering, the Itanium architecture requires a memory fence in the middle of the {store x, load y} sequence and the {store y, load x} sequence. No combination of ordered semantics on the operations in each of these sequences will guarantee the correct ordering.

It is not possible for the store y in the second sequence to pass the load y in the first sequence because of the data dependency from the load y to the compare and branch. If the processor reaches the store y in the second sequence, the load of y from the first sequence must be visible. Likewise, it is not possible for memory operations in the critical section to move ahead of the final load x because of the data dependency between this load and the compare and branch that guards the critical section.

The accesses to the b array allow the algorithm to correctly handle contention for the lock. In such cases, the algorithm backs off and re-trys.

[1] There are some additional operations on the b array that are interposed in this sequence when contention for the resource occurs.

Figure 2-7. Lamport's Algorithm

```
// The proc_id variable holds a unique, non-zero id for the process that
// attempts access to the critical section. x and y are the synchronization
// variables that indicate who is in the critical section and who is attempting
// entry. ptr_b_1 and ptr_b_id point at the 1'st and id'th element of b[].
//
lamport:
            ld8    r1 = [proc_id] ;;          // r1 = unique process id
start:
            st8    [ptr_b_id] = r1            // b[id] = "true"
            st8    [x] = r1                   // x = process id
            mf                                // MUST fence here!
            ld8    r2 = [y] ;;
            cmp.ne p1, p0 = 0, r2;;           // if (y != 0) then...
(p1)        st8    [ptr_b_id] = r0            // ... b[id] = "false"
(p1)        br.cond.sptk  wait_y              // ... wait until y == 0

            st8    [y] = r1                   // y = process id
            mf                                // MUST fence here!
            ld8    r3 = [x] ;;
            cmp.eq p1, p0 = r1, r3 ;;         // if (x == id) then...
(p1)        br.cond.sptk  cs_begin            // ... enter critical section

            st8    [ptr_b_id] = r0            // b[id] = "false"
            ld8    r3 = [ptr_b_1]             // r3 = &b[1]
            mov    ar.lc = N-1 ;;             // lc = number of processors - 1
wait_b:
            ld8    r2 = [r3] ;;
            cmp.ne p1, p0 = r1, r2            // if (b[j] != 0) then...
(p1)        br.cond.spnt  wait_b ;;           // ... wait until b[j] == 0
            add    r3 = 8, r3                 // r3 = &b[j+1]
            br.cloop.sptk wait_b ;;           // loop over b[j] for each j

            ld8 r2 = [y] ;;
            cmp.ne p1, p0 = r2, r1 ;;         // if (y != id) then...
(p1)        br.cond.sptk  cs_begin            // ... enter critical section
wait_y:
            ld8    r2 = [y] ;;                // wait until y == 0
            cmp.ne p1, p2 = 0, r2
(p1)        br.cond.spnt  wait_y
            br     start                      // back to start to try again

cs_begin:
            // critical section code goes here...
cs_end:

            st8    [y] = r0                   // release the lock
            st8.rel[ptr_b_id] = r0;;          // b[id] = "false"
```

2.5 Updating Code Images

There are four general techniques for updating code images in order to modify the code stream of a local or remote processor.

- Self-modifying code or code that modifies its own image.
- Cross-modifying code or code that modifies the image of code running concurrently on another processor.
- Programmed I/O for paging of code pages.
- DMA for paging of code pages.

The next four sections discuss these techniques in greater depth.

To illustrate the code sequences for self- and cross-modifying code, the examples in this section use the syntax "`st [foo] = new`" to represent a group of aligned stores that change the instruction at address `foo` to the instruction "`new`". The Itanium architecture requires that the instruction stream see aligned stores atomically. In addition, the syntax "`fc foo`" represents a group of flush cache instructions that flush the cache line(s) that contain the instruction at address `foo`. Updating more than one instruction simply requires the appropriate store/flush "pair" for each updated instruction[1].

2.5.1 Self-modifying Code

Figure 2-8 presents the Itanium instruction sequence necessary to update a code image location on the local processor only.

Figure 2-8. Updating a Code Image on the Local Processor

```
patch_local:
        st     [code] = new_inst      // write new instruction
        fc     code ;;                // flush new instruction
        sync.i ;;                     // sync i stream with store
        srlz.i ;;                     // serialize

        // Local caches and pipeline are now coherent with
new_inst...
```

This code fragment changes the instruction at the address `code` to the new instruction `new_inst`. After executing this code, the change is visible to both the local processor's caches and its pipeline.

The `st` and `fc` instructions first update the code image and then invalidate the cache line(s) that contain the updated instruction. The `fc` is necessary because the Itanium architecture does not require instruction caches to be coherent with data stores for Itanium-based code. Next, the `sync.i` ensures that the code update is visible to the instruction stream of the local processor and orders the cache flush with respect to subsequent operations by waiting for the prior `fc` instructions to be made visible. Finally, the `srlz.i` instruction forces the pipeline to re-initiate any instruction group fetches it performed after the `srlz.i` and also waits for the `sync.i` to complete; effectively making the pipeline coherent with the updated code image.

[1] This description hides some of the complexity involved. Specifically, the flush and store operations have different sizes. Whereas multiple store instructions are necessary to update a 16 byte instruction, a single cache line flush invalidates at least two 16 byte instructions.

The serialization instruction is not necessary if software can *guarantee* that the processor encounters an event that re-initiates code fetches performed after the sync.i, such as an interruption or an rfi, before executing the new code. Events such as an interrupt or rfi both perform an instruction serialization which in this example waits for the sync.i to complete and then re-initiates code fetches.

2.5.2 Cross-modifying Code

Consider a multi-threaded program for a multiprocessor system that dynamically updates some procedure that any processor in the system may execute. The program maintains several disjoint buffers to hold the new code and requires a processor to execute an IP-relative branch instruction at some address x to reach the code. In this scenario, the program updates the procedure by emitting the new code into a different buffer and then patching the branch at address x to target this new buffer. By carefully writing the update code, software can ensure that any processor in the system sees either:

- The original branch at address x that targets the original code in the old buffer along with the original code, or
- The new branch at address x that targets the new code in the new buffer along with the new code.

The code in Figure 2-9 illustrates an optimized Itanium-based code sequence that implements the cross-modifying code for this example.

Figure 2-9. Supporting Cross-modifying Code without Explicit Serialization

```
patch:
            st    [new_code] = new_inst      // write new instruction
            fc    new_code ;;                // flush new instruction
            sync.i ;;                        // sync i stream with store

// Update the target of the branch that jumps to the updated code. This
// branch MUST be ip-relative. Before executing the following store,
// the branch jumps to somewhere other than “new_code”.
//
            st.rel [x] = “branch <new_code>”

// If it is desired to propagate “branch <new_code>” to all other
// processors now, the following code is also necessary:
//
            fc    x ;;                       // flush branch
            sync.i ;;                        // sync i stream with store
            mf ;;                            // fence
```

To reach the new code at new_code, the processor executes the branch instruction at x. Initially, this branch jumps to an address other than new_code.

The release store ensures a processor cannot see the new branch at address x and the original code at address new_code. That is, if a processor encounters “branch <new_code>” at address x, then the processor’s instruction cache must be coherent with the code image updates applied before the release store that updates the branch.

If remote processors may see either the old or new code sequence, the final three instructions in Figure 2-9 are not necessary. In this case, the remote processors see the code image updates at some point in the future. In the meantime, they continue to execute the old code.

The release store ensures that the code image updates are made visible to the remote processors in the proper order (i.e. `new_code` is updated before the branch at address `x` is updated). Using the final three instructions ensures that the remote processors will see the new code the next time they execute the branch at address `x`.

On the local processor, the branch at address `x` also serves to force the pipeline to be coherent with the code image update the machine without requiring an interrupt, `rfi` instruction, or `srlz.i` instruction. Table 2-16 enumerates the potential pipeline behaviors to illustrate this point.

Table 2-16. Potential Pipeline Behaviors of the Branch at `x` from Figure 2-9

Pipeline Operation	Scenario #1	Scenario #2	Scenario #3	Scenario #4
Fetch branch at x	Old branch	Old branch	New branch	New branch
Predict branch at x	Old target	New target	Old target	New target
Code at target	Old instruction	"New" instruction (but could be stale)	Old instruction	New instruction
Retire branch at x	Old retires	Must flush due to misprediction	Must flush due to misprediction	New retires

In the first and fourth scenarios, the pipeline fetches and executes either the old branch and old target instruction or the new branch and new target instruction. Note that if the pipeline sees the new branch, it must also see the new target instruction by virtue of the way the code in Figure 2-9 is written. Either of these behaviors is consistent.

In the second and third scenarios, the pipeline obtains a mix of the old or new branch and the old or new target instruction. In these cases, the pipeline must flush because the predicted target will not agree with the branch instruction.

This behavior is not guaranteed unless the branch at address `x` is IP-relative and taken. The branch must be IP-relative to ensure that both the instruction and target address can be atomically updated (this is only possible with an IP-relative branch because in this type of branch, the target address is part of the instruction).

2.5.3 Programmed I/O

Programmed I/O requires that the CPU copy data from the device controller to main memory using load instructions to read from the device and store instructions to write data into cacheable memory (page-in).

To ensure correct operation, Itanium-based software must exercise care in the presence of Programmed I/O due to two features of the architecture. First, the Itanium architecture does not require an implementation to maintain coherency between local instruction and data caches for Itanium-based code. Second, the Itanium architecture allows aggressive instruction prefetching. Specifically, an implementation can move any location from a cacheable page into its instruction cache(s) any time a translation for the location indicates that the page is present (i.e. the p bit of the translation is set).

A system that performs Programmed I/O can use a sequence similar to that shown in Figure 2-8 to perform the data movement. Figure 2-10 presents a code sequence that updates a code image on both the local and remote processors.

Figure 2-10. Updating a Code Image on a Remote Processor

```
patch_l_and_r:
            st    [code] = new_inst          // write new instruction
            fc    code ;;                    // flush new instruction
            sync.i ;;                        // sync i stream with store

// If the local processor must ensure that remote processors see the
// preceding memory updates before any subsequent memory operations,
// the following code is also necessary.
//
            mf ;;                            // make store visible to
others

// If the local processor is going to execute the code and cannot
// cannot ensure instruction stream serialization, the following code
// is also necessary,
//
            srlz.i ;;                           // serialize my pipeline

// Local caches and pipeline are now coherent with new_inst, remote
// caches are now coherent with new_inst...
```

This code fragment changes the instruction at the address `code` to the new instruction `new_inst`. After executing this code, the change is visible to the local and remote processor's caches and to the local processor's pipeline, but may not be visible to remote processor's pipelines.

The sequence in Figure 2-10 is similar to the code from Figure 2-8 except an `mf` instruction occurs between the `sync.i` and `srlz.i` instructions. The fence is necessary if software must ensure that the code image update is made visible to all remote processors before any subsequent memory operations from the local processor. Although the `sync.i`, which orders the `st`/`fc` pair, has unordered semantics, it is an orderable operation and thus obeys the release or fence semantics of subsequent instructions (unlike an `fc` instruction; see Section 4.4.7, "Sequentiality Attribute and Ordering" for more information).

Because the pipeline is not snooped, the code in Figure 2-10 cannot ensure that a remote processor's pipeline is coherent with the code image update. In the local case shown in Figure 2-8, the `srlz.i` instruction enforces this coherency. As a result, the remote processor must serialize its instruction stream before it executes the updated code in order to ensure that a stale copy of some of the updated code is not present in the pipeline. This can be accomplished by explicitly executing a `srlz.i` before executing the updated code or by forcing an event that re-initiates any code fetches performed after the `fc` is observed to occur, such as an interruption or `rfi`.

Several optimizations to this code are possible depending on how software uses the updated code. Specifically, the `mf` and `srlz.i` can be eliminated under certain circumstances.

The `srlz.i` is not necessary if the local processor that updates the code image does not ever execute the new code. In this case, the local processor does not require its pipeline to be coherent with the changes to the code image. The fence is not necessary if the code image update can be made visible to remote processors in any relationship with subsequent memory operations from the local processor.

Finally, software may also eliminate the `mf` or `srlz.i` instructions if it *guarantees* that these operations will take place elsewhere (e.g. in the operating system) before the processor attempts to execute the updated code. For example, context switch routines must contain a memory fence (see Section 2.3 on page page 2:392). Thus, the fence is not required if a context switch *always* occurs before any program can use the updated code.

2.5.4 DMA

Unlike Programmed I/O, which requires intervention from the CPU to move data from the device to main memory, data movement in DMA occurs without help from the CPU. A processor based on the Itanium architecture expects the platform to maintain coherency for DMA traffic. That is, the platform issues snoop cycles on the bus to invalidate cacheable pages that a DMA access modifies. These snoop cycles invalidate the appropriate lines in both instruction and data caches and thus maintain coherency. This behavior allows an operating system to page code pages without taking explicit actions to ensure coherency.

Software must maintain coherency for DMA traffic through explicit action if the platform does not maintain coherency for this traffic. Software can provide coherency by using the flush cache instruction, `fc`, to invalidate the instruction and data cache lines that a DMA transfer modifies. Code such as that shown in Figure 2-8 on page 2:399 and Figure 2-10 on page 2:402 accomplish this task.

2.6 References

[AG95] S. V. Adve and K. Gharachorloo. “Shared memory consistency models: A Tutorial,” Rice University ECE Technical Report 9512, September 1995.

[L79] L. Lamport. “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Transactions on Computers*, C-28(9):690-691, September 1979.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, second edition, Morgan-Kaufmann, 1996.

[D65] E. W. Dijkstra. “Cooperating sequential processes,” Eindhoven, the Netherlands, Technological University Technical Report EWD-123, 1965.

[L85] L. Lamport. “A Fast Mutual Exclusion Algorithm,” Compaq Systems Research Center Technical Report 7, November 1985.

Interruptions and Serialization 3

This chapter discusses the interruption and serialization model. Although the Itanium architecture is an explicitly parallel architecture, faults and traps are delivered in program order based on IP, and from left-to-right in each instruction group. In other words, faults and traps are reported precisely on the instruction that caused them.

3.1 Terminology

In the Itanium architecture, an **interruption** is an event which causes the hardware automatically to stop execution of the current instruction stream, and start execution at the instruction address corresponding to the **interruption handler** for that interruption. When this happens, we say that an interruption has been **delivered** to the processor core.

There are two classes of interruptions in the Itanium architecture. **IVA-based interruptions** are handled by the operating system (OS), at an address determined by the location of the interrupt vector table (IVT) and the particular interruption that has occurred. **PAL-based interruptions** are handled by the processor firmware. PAL-based interruptions are not visible to the OS, though PAL may notify the OS that a PAL-based interruption has occurred; see Section 13.3, “Event Handling in Firmware” on page 2:489.

The architecture supports several different types of interruptions. These are defined below:

- A **fault** occurs when OS intervention is required before the current instruction can be executed. For example, if the current instruction misses the TLBs on a data reference, a Data TLB Miss fault may be delivered by the processor. Faults are delivered precisely on the instruction that caused the fault. The faulting instruction and all subsequent instructions do not update any architectural state (with the possible exception of subsequent instructions which violate a resource dependency[1]). All instructions executed prior to the faulting instruction update all their architectural state before the fault handler begins execution.
- A **trap** occurs when OS intervention is required after the current instruction has completed. For example, if the last instruction executed was a branch and PSR.tb is 1, a Taken Branch trap will be delivered after the instruction completes. Traps are delivered precisely on the instruction following the trapping instruction. The trapping instruction and all prior instructions update all their architectural state before the trap handler begins execution. All instructions subsequent to the trapping instruction do not update any architectural state.[1]
- When an external or independent agent (I/O device, timer, another processor) requires attention from the processor, an **interrupt** occurs. There are several types of interrupts. An initialization interrupt occurs when the processor has received an initialization request. A **Platform Management Interrupt** (PMI) can be generated by the platform to request features

[1] When an interruption is delivered on an instruction whose instruction group contains one or more illegal dependency violations, instructions which follow the interrupted instruction in program order and which violate the resource dependency may appear to complete before the interruption handler begins execution. Software cannot rely upon the value(s) written to the resource(s) whose dependencies have been violated; the value(s) are undefined. For details refer to Section 3.4, “Instruction Sequencing Considerations” in Volume 1.

such as power management. Initialization interrupts and PMIs are PAL-based interruptions. An **external interrupt** occurs when an agent in the system requires the OS to perform some service on its behalf. External interrupts are IVA-based interruptions. Interrupts are delivered asynchronously with respect to program execution. The instruction upon which an interrupt is delivered may or may not be related to the interrupt itself.

- An **abort** is generated by the processor when a malfunction (Machine Check) is detected, or when a processor reset occurs. Aborts are asynchronous with respect to program execution. If caused by a particular instruction, an abort may be delivered sometime after that instruction completes. Aborts are PAL-based interruptions.

An interruption handler returns from interruption when it executes an `rfi` instruction. The `rfi` instruction copies state from specific control registers known as **interruption registers** into their corresponding architectural state (e.g. IIP is copied into IP and execution begins at that instruction address). Whether or not the state that is restored by the `rfi` is the same state that was captured when the interruption occurred is up to the operating system.

3.2 Interruption Vector Table

The Interruption Vector Address (IVA) control register defines the base address of the interruption vector table (IVT). Each IVA-based interruption has its own architected offset into this table as defined in Section 5.7, “IVA-based Interruption Vectors”. For the remainder of this section, “interruption” refers to an IVA-based interruption, unless otherwise noted.

When an interruption occurs, the processor stops execution at the current IP, sets the current privilege level to 0, and begins fetching instructions from the address of the entry point to the interruption handler for the particular interruption that occurred. The address of this entry point is defined by the base address of the IVT contained in the IVA register and the architected offset into the table according to the interruption that occurred.

The IVT is 32Kbytes long and contains the code for the interruption handlers. Execution of the interruption handler begins at the entry point. The interruption handler may be contained entirely in the IVT, or the handler may branch to code outside the IVT if more space is needed.

When an interruption occurs, if the processor is operating with instruction address translation enabled (PSR.it is 1), then the address in IVA is treated as a virtual address; otherwise, it is treated as a physical address. Whenever an interruption may occur (i.e. whenever external interrupts are not masked or disabled, or whenever an instruction may raise a fault or trap), the software must ensure that the processor can safely reference the IVT. As a result, the IVT must be permanently resident in physical memory. If instruction address translation is enabled, the IVT must be mapped by an instruction translation register and must point at a valid physical page frame. When instruction address translation is disabled, the IVA register should contain the physical address of the base of the IVT. Software must further ensure that instruction and memory references from low-level interruption handlers do not generate additional interruptions until enough state has been saved and interruption collection can be re-enabled.

There are many more interruptions than there are interruption vectors in the IVT. As specified in Section 5.6, “Interruption Priorities” there is a many-to-one relationship between interruptions and interruption vectors. The interruptions that share a common interruption vector (and hence, the code for an interruption handler) can determine which interruption occurred by reading the

Interruption Status Register (ISR) control register. See Chapter 8, “Interruption Vector Descriptions” and Chapter 9, “IA-32 Interruption Vector Descriptions” for details of the specific ISR settings for each unique interruption.

3.3 Interruption Handlers

3.3.1 Execution Environment

As defined in Section 5.5, “IVA-based Interruption Handling”, the processor automatically clears the PSR.i and PSR.ic bits when an interruption is delivered. This disables external interrupts and interrupt state collection, respectively. PMI delivery is also disabled while PSR.ic is 0; other PAL-based interruptions can be delivered at any point during the execution of the interruption handler, regardless of the state of PSR.i and PSR.ic.

In addition to clearing the PSR.i and PSR.ic bits, the processor also automatically clears the PSR.bn bit when an interruption is delivered, switching to bank 0 of general registers GR16 - GR31. This provides the interruption handler with its own set of registers which can be used without spilling any of the interrupted context’s register state, effectively saving GR16 - GR31 of the interrupted context. (This assumes PSR.bn is 1 at the time of interruption; see Section 3.4.3 for how to deal with the case where PSR.bn is 0 at the time of interruption.)

As specified in Section 3.3.7, “Banked General Registers”, GR24 - GR31 **of bank 0** should not be used while PSR.ic is 1. By firmware convention, PAL-based interruption handlers may use these registers without preserving their values when PSR.ic is 1. When PSR.ic is 0, software may safely use GR24 - GR31 of bank 0 as scratch register.

Several other PSR bits and the RSE.CFLE are modified by the hardware when an interruption is delivered. Table 3-1 summarizes the execution environment that interruption handlers operate in, and what each PSR bit and the RSE.CFLE values mean for the interruption handler.

Table 3-1. Interruption Handler Execution Environment (PSR and RSE.CFLE Settings)

PSR Bit	New Value	Effect on Low-level Interruption Handle
be	DCR.be	Byte order used by handler is determined by be-bit in DCR register.
ic & i	0	Disables interruption collection and external interrupts. Bank 0 is made active bank. This is discussed above
bn	0	
dt, rt, it, pk	unchanged	Instruction/Data/RSE address translation and protection key setting remain unchanged.
dfl & dfh	0	Floating-point registers are made accessible. This allows handlers to spill FP registers without having to toggle FP disable bits first. Modified bits indicate which registers were touched. See Section 4.2.2, “Preservation of Floating-point State in the OS” on page 2:419 for details.
mfl, mfh	unchanged	
pp	DCR.pp	Privileged Monitoring is determined by pp-bit in DCR register. By default, user counters are enabled and performance monitors are unsecured in handlers. See Chapter 12, “Performance Monitoring Support” for details.
up	unchanged	
sp	0	
di	0	Instruction set transitions are not intercepted.
si	0	Interval timer is unsecured.

Table 3-1. Interruption Handler Execution Environment (PSR and RSE.CFLE Settings)

PSR Bit	New Value	Effect on Low-level Interruption Handle
ac	0	No alignment checks are performed.
db, lp, tb, ss	0	Debug breakpoints, lower-privilege interception, taken branch and single step trapping are disabled.
cpl	0	Current privilege level becomes most privileged.
is	0	Intel® Itanium™ Instruction set. Handlers execute Intel® Itanium™ instructions.
id, da, ia, dd, ed	0	Instruction/data debug, access bit and speculation deferral bits are disabled. For details, refer to Section 5.5.4, “Single Instruction Fault Suppression” and Section 5.5.5, “Deferral of Speculative Load Faults”.
ri	0	Interrupt handler starts at first instruction is bundle.
mc	unchanged	Software can mask delivery of some machine check conditions by setting PSR.mc to 1, but the processor hardware does not set this bit upon delivery of an IVA-based interruption. Delivery of resets and BINITs cannot be masked.
RSE.CFLE (not a PSR bit)	0	Allows interruption handler to service faults in presence of an incomplete current register stack frame. This can happen when a mandatory RSE load takes an exception during when RSE is servicing a register stack underflow. For details refer to Section 6.6, “RSE Interruptions”.

3.3.2 Interruption Register State

The Itanium architecture provides a set of hardware registers which, if interruption collection is enabled, capture relevant interruption state when an interruption occurs. The state of the PSR.ic bit at the time of an interruption controls whether collection is enabled. In this section, it is assumed that interruption collection is enabled (PSR.ic is 1); see Section 3.4.3 for details on handling interruptions when collection is disabled (PSR.ic is 0). For details on collection of interruption resources for each interruption vector refer to Chapter 8, “Interruption Vector Descriptions” and Chapter 9, “IA-32 Interruption Vector Descriptions”.

A processor based on the Itanium architecture provides the following interruption registers for collecting information about the latest interruption or the state of the machine at the time of the interruption:

- IPSR – A copy of the processor status register (PSR) at the moment the interruption occurred. The OS can use the IPSR to determine the value of any PSR bit when the interruption occurred. The contents of IPSR are restored into the PSR when the OS executes an `rfi` instruction. If the OS wishes to change the PSR state of the interrupted process (e.g. to step over an instruction debug fault), it can do so by modifying the IPSR contents before executing the `rfi`. When an interruption occurs, the processor sets IPSR.ri to the slot number (0, 1, or 2) of the instruction that was interrupted.
- IIP – A copy of the instruction pointer (IP) where the interruption occurred. The instruction bundle address contained in IIP, along with the IPSR.ri field, defines the instruction whose execution was interrupted. This instruction has not completed (i.e. it has not retired), so when the OS returns to the interrupted context, typically this is the instruction at which execution of the interrupted context resumes[1]. When the OS executes an `rfi` instruction, the contents of IIP are copied into the IP register and the processor begins fetching instructions from this address.

- ISR – Contains extra information about the specific interruption that occurred. This register is useful for determining exactly which interruption occurred for interruptions which share the same IVT vector.
- IFA – Faults related to addressing (e.g. Data TLB fault) materialize the faulting address in this register.
- ITIR – Faults related to addressing materialize the default page size and permission key for the region to which the faulting address belongs in this register.
- IIPA – Contains the instruction bundle address of the last instruction to retire successfully while PSR.ic was 1. In conjunction with ISR.ei, IIPA can be used by software to locate the instruction that caused a trap or that was executed successfully prior to a fault or interrupt.
- IIM – Instructions that take a Speculation fault (e.g. `chk`) or a Break Instruction fault (e.g. `break.i`) write this register with their immediate field when taking these faults. For these cases, the IIM register can be used to emulate the instruction, or to pass information to the fault handler; for example, software can use a particular immediate field value in a break instruction to indicate to the operating system that a system call is being performed.
- IHA – Faults related to the VHPT place the VHPT hash address in this register. See Section 5.3, “Virtual Hash Page Table” on page 2:434 for details.
- IFS – This register can be used by software to save a copy of the interrupted context’s PFS register, but an interruption handler must do this explicitly; hardware only clears the valid bit (IFS.v) upon interruption. See below for details.

No other architectural state is modified when an interruption occurs. Note that only IIP, IPSR, ISR, and IFS are written by all interruptions (assuming PSR.ic is 1 at the time of interruption); the other interruption control registers are only written by certain interruptions, and their values are undefined otherwise. For details on which faults update which interruption resources refer to Chapter 8, “Interruption Vector Descriptions” and Chapter 9, “IA-32 Interruption Vector Descriptions”.

3.3.3 Resource Serialization of Interrupted State

As defined in Section 3.2, “Serialization”, Itanium control register updates do not take effect until software explicitly serializes the processor’s data or instruction stream with a `srlz.d` or a `srlz.i` instruction, respectively. Control register updates that change a control register’s value and that have not yet been serialized are termed “in-flight”. Refer to Section 3.2.3, “Definition of In-flight Resources” for a precise definition.

When an interruption is delivered and before execution begins in the interruption handler, the processor hardware automatically performs an instruction and data serialization on all “in-flight” control registers, except for 4 resources: the IVA control register, DCR.be, DCR.pp, and PSR.ic.

[1] When an instruction faults because it requires emulation by the OS, the OS will normally skip the emulated instruction by returning to the instruction bundle address and slot number that follows IIP in program order. It does so by writing the next in-order bundle address and slot number into IIP and IPSR.ri, respectively, before executing an `rfi` instruction. Details on emulation handlers is in Chapter 7, “Instruction Emulation and Other Fault Handlers”.

As described in Section 3.3.1 above, these four resources determine the execution environment of the interruption handler. As a result, to update these four resources, software must ensure that external interrupts are disabled and that no instruction or data references will take an exception until the resource update has been appropriately serialized. Typically, the code toggling these four resources is mapped by an instruction translation register to avoid TLB related faults.

For example, assume that GR2 contains the new value for IVA and that PSR.i is 1. To modify the IVA register, software would perform the following code sequence, where the code page is mapped by an instruction translation register or instruction translation is disabled:

```
rsm psr.i          // external interrupts disabled upon next instruction
mov cr[iva] = r2
;;
srlz.i             // writing IVA requires instruction serialization
;;
ssm psr.i          // external interrupts will be re-enabled after next srlz
```

3.3.4 Resource Serialization upon rfi

An `rfi` instruction also performs an instruction and a data serialization operation when it is executed. Any values that were written to processor register resources by instructions in an earlier instruction group than the `rfi` will be observed by the returned-to instruction, except for those register resources which are also written by the `rfi` itself, in which case the value written by the `rfi` will be observed. This makes the interruption handler more efficient by avoiding additional data and instruction serialization operations before returning to the interrupted context.

3.4 Interruption Handling

The Itanium-based operating systems need to distinguish the following interruption handler types:

- Lightweight interruptions: Lightweight interruption handlers are allocated 1024 bytes (192 instructions) per handler in the IVT. These are discussed in Section 3.4.1.
- Heavyweight interruptions: Heavyweight interruption handlers are allocated only 256 bytes (48 instructions) per handler in the IVT. These are discussed in Section 3.4.2.
- Nested interruptions: If an interruption is taken when PSR.ic was 0 or was in-flight, a nested interruption occurs. Nested interruptions are discussed in Section 3.4.3.

3.4.1 Lightweight Interruptions

Lightweight interruption handlers are allocated 1024 bytes (192 instructions) per handler in the IVT. Typically, lightweight handlers are written in Itanium-based assembly code, and run in their entirety with interruption collection turned off (PSR.ic = 0) and external interrupts disabled (PSR.i = 0). Because these lightweight handlers are usually very short and performance-critical, they are intended to fit entirely in the space allocated to them in the IVT. An example of a lightweight interruption handler is the Data TLB vector (offset 0x0800). The first 20 vectors in the IVT, offsets 0x0000 (VHPT Translation vector) through 0x4c00 (reserved), are lightweight vectors. Typical lightweight handlers deal with instruction, data or VHPT TLB Misses, protection key miss handling, and page table dirty or access bit updates.

A typical lightweight interruption handler can operate completely out of register bank 0. If the bank 0 registers provide sufficient storage for the handler, none of the interrupted context's register state need be saved to memory, and the handler does not need to use stacked registers. Assuming no stacked registers are needed, the lightweight interruption handler can operate with an incomplete current register stack frame, obviating the need for `cover` and `alloc` instructions in the handler. This also allows the TLB related handlers to service TLB misses that result from mandatory RSE loads to the current frame.

3.4.2 Heavyweight Interruptions

Heavyweight interruption handlers are allocated only 256 bytes (48 instructions) per handler in the IVT. This stub provides enough space to save minimal processor state, re-enable interruption collection and external interrupts, and branch to another routine to handle the interruption. Unlike a lightweight interruption handlers described above, heavyweight interruption handlers use general register bank 0 only until they can establish a safe memory context for spilling the interrupted context's state. This allows heavyweight handlers to be interruptible and to take exceptions.

A heavyweight handler stub (i.e. the portion of the handler that is located in the IVT) should determine exactly which type of interruption has occurred based on its offset in the IVT and the contents of the ISR control register. It can then branch out of the IVT to the actual interruption handler. For some heavyweight interruptions (e.g. Data Debug fault), these handlers are typically written in a high-level programming language; for others (e.g. emulation handlers) the interruption can be handled efficiently in Itanium-based assembly code.

The sequence given below illustrates the steps that an Itanium-based heavyweight handler needs to perform to save the interrupted context's state to memory and to create an interruptible execution environment. These steps assume that the low-level kernel code, the kernel backing store, and the kernel memory stack are pinned in the TLB (using a translation register), so that no TLB misses arise from referencing those memory pages. The ordering of the steps below is approximate and other operating system strategies are possible.

1. Copy the interruption resources (IIP, IPSR, IIPA, ISR, IFA) into bank 0 of the banked registers. To avoid conflicts with processor firmware, use registers GR24-31 for this purpose. Both register bank 0 and the interruption control registers are accessible, since, as described in Section 3.3.1, the processor hardware, upon an interruption always switches to register bank 0, and clears PSR.ic and PSR.i.
2. Preserve the interrupted the predicate registers into bank 0 of the banked registers.
3. Determine whether interruption occurred in the operating system kernel or in user space by inspecting both IPSR.cpl and the memory stack pointer (GR12).
    a. If IPSR.cpl is zero and the interrupted context was already executing on a kernel stack, then no memory stack switch is required.
    b. Otherwise, software needs to switch to a kernel memory stack by preserving the interrupted memory stack pointer to a banked register in bank 0, and setting up a new kernel memory stack pointer in GR12.
4. Allocate a "trap frame" to store the interrupted context's state on the kernel memory stack, and move the interruption state (IIP, IPSR, IIPA, ISR, IFA, IFS), the interrupted memory stack pointer and the interrupted predicate registers from the banked registers to the trap frame.
5. Save register stack and RSE state by following the steps outlined in Section 6.11.1, "Switch from Interrupted Context".

a. If IPSR.cpl is zero and the interrupted context was not executing on a kernel backing store (determined by inspecting BSPSTORE), then the new kernel BSPSTORE needs to be allocated such that enough space is provided for the RSE to spill all stacked registers. The architectural required maximum RSE spill area is 16KBytes. As a result, BSPSTORE should be offset from the base of the kernel backing store base by at least 16KBytes. This offset can be reduced if the kernel queries PAL for the actual implementation specific number of stacked physical registers (RSE.N_STACK_PHYS). Based on RSE.N_STACK_PHYS, the required minimum offset in bytes is:

```
8 * (RSE.N_STACK_PHYS + 1 + truncate((RSE.N_STACK_PHYS + 62)/63))
```

Otherwise, the interrupted context was already executing on the kernel backing store. In this case, no new BSPSTORE pointer needs to be setup. The sequence in Section 6.11.1, "Switch from Interrupted Context", is still required, however, step 6 in that sequence can be omitted.

In either case, the interrupted register stack and RSE state (RSC, PFS, IFS, BSPSTORE, RNAT, and BSP) needs to be preserved, and should be saved either to the trap frame on the kernel memory stack, or to a newly allocated register stack frame.

6. Switch banked register to bank one and re-enable interruption collection as follows:

```
ssm 0x2000// Set PSR.ic
bsw.1;;// Switch to register bank 1
srlz.d// Serialize PSR.ic update
```

   With interruptions collection re-enabled, the kernel may now branch to paged code and may reference paged data structures.

7. Preserve branch register and application register state according to operating system conventions.

8. Preserve general and floating-point register state. If this is an involuntary interruption, e.g. an external interrupt or an exception, then software must save the interrupted context's volatile general register state (scratch registers) to the "trap frame" on the kernel memory stack, or to the newly allocated register stack frame. If this is a voluntary system call then there is no volatile register state. Preserved registers may or may not be spilled depending on operating system conventions. Additionally, the Itanium architecture provides mechanisms to reduce the amount of floating-point register spills and fills. More details on preservation of register context are given in Section 4.2, "Preserving Register State in the OS" on page 2:417.

9. At this point enough context has been saved to allow complete restoration of the interrupted context. Re-enable taking of external interrupts using the ssm instruction as follows:

```
ssm 0x4000 ;; // Set PSR.i
```

   There is no need to explicitly serialize the PSR.i update, unless there is a requirement to force sampling of external interrupts right away. Without the serialization, the PSR.i update will occur at the very latest when the next exception causes an implicit instruction serialization to occur.

10. Dispatch interruption service routine (can be high-level programming language routine).

11. Return from interruption service routine.

12. Disable external interrupts as follows:

```
rsm 0x4000 ;; // Clear PSR.i
```

   There is no need to explicitly serialize the PSR.i update, since clearing of the PSR.i bit with the `rsm` instruction takes effect at the next instruction group. For details refer to the `rsm` instruction page in Chapter 2, "Instruction Reference" of Volume 3.

13. Restore general and floating-point register state saved in step 8 above.

14. Restore branch register and application register state saved in step 7 above.
15. Disable collection of interruption resources and switch banked register to bank zero as follows:

```
rsm 0x2000// Clear PSR.ic
bsw.0;;// Switch to register bank 0
srlz.d// Serialize PSR update
```

16. Restore register stack and RSE state by following the steps outlined in Section 6.11.2, “Return to Interrupted Context”.
17. Restore interrupted context’s interruption state (IIP, IPSR, IIPA, ISR, IFA, IFS) from the “trap frame” on the kernel memory stack.
18. Restore interrupted context’s memory stack pointer and predicate registers from the trap frame on the kernel memory stack. This step essentially deallocates the trap frame from the kernel memory stack.
19. Return from interruption using the `rfi` instruction.

Many of the steps shown above are identical for different heavyweight interruptions, so unless there is a specific need to create a different handler for a particular interruption, a common handler can be used. Because external interrupt handlers use the Itanium external interrupt control registers to determine the specific external interrupt vector that needs servicing and to mask off other external interrupt vectors, an external interrupt handler looks somewhat different. Refer to Section 10.4, “External Interrupt Delivery” on page 2:465 for details on writing external interrupt handlers.

3.4.3 Nested Interruptions

The Itanium architecture provides a single set of interruption registers whose updates are controlled by PSR.ic. When an IVA-based interruption is delivered and PSR.ic is 0 or in-flight (e.g. during a lightweight interruption handler, or at the beginning of a heavyweight interruption handler), we say that a nested interruption has occurred. On a nested interruption (other than a Data Nested TLB fault) only ISR is updated by the hardware. All other interruption registers preserve their pre-interruption contents.

With the exception of the Data Nested TLB fault, the Itanium architecture does not support nested interruptions. Data Nested TLB faults are special and are discussed in Section 5.4.4, “Data Nested TLB Vector” on page 2:439. The remainder of this section does not apply to Data Nested TLB faults.

When a nested interruption occurs, the processor will update ISR as defined in Chapter 8, “Interruption Vector Descriptions” and it will set the ISR.ni bit to 1. A value of 1 in ISR.ni is the only indication to an interruption handler that a nested interruption has occurred. Since all other interruption registers are not updated, there is generally no way for the OS to recover from nested interruptions; the handler for the nested interruption has no context other than ISR for handling the nested interruption. If a nested interruption is detected, it is often useful for the handler to call some function in the OS that logs the state of ISR, IIP, and any other relevant register state to aid in debugging the problem.

Context Management 4

This chapter discusses specific context management considerations in the Itanium architecture. With 128 general registers and 128 floating-point registers, the architecture provides a comparatively large amount of state. This chapter discusses various context management and state preservation rules. This chapter introduces some architectural features that help an operating system limit the amount of register spill/fill and gives recommendations to system programmers as to how to use some of the instruction set features.

4.1 Preserving Register State across Procedure Calls

The Itanium Software and Runtime Architecture Conventions [SWC] define a contract on register preservation between procedures as follows:

- Scratch Registers (Caller Saves): GR2-3, GR8-11, GR14-GR15, and GR16-31 in register bank 1, FR6-15, and FR32-127. Code that expects scratch registers to hold their value across procedure calls is required to save and restore them.
- Preserved Registers (Callee Saves): GR4-7, FR2-5, and FR16-31. Procedures using these registers are required to preserve them for their callers.
- Stacked Registers: GR32-127, when allocated, are preserved by the RSE.
- Constant Register: GR0 is always 0. FR0 is always +0.0. FR1 is always +1.0.
- Special Use Registers: GR1, GR12, and GR13 have special uses.

Additional architectural register usage conventions apply to GR16-31 in register bank 0 which are used by low-level interrupt handlers and by processor firmware. For details refer to Section 3.3.1.

Itanium general registers and floating-point registers contain three state components: their register value, their control speculative (NaT/NaTVal) state, and their data speculative (ALAT) state. When software saves and restores these registers, all three state components need to be preserved. As described in Table 4-1, software is required to use different state preservation methods depending on the type of register. More details on register preservation are provided in the next two sections.

Table 4-1. Preserving Intel® Itanium™ General and Floating-point Registers

State Components	General Registers		Floating-point Registers
	GR1-31 (static)	GR32-127 (stacked)	FR2-127
Register Value	`st8.spill` & `ld8.fill` preserve register value.	RSE automatically preserves register value.	`stf.spill` & `ldf.fill` preserve register value.
Control Speculative State (NaT/NaTVal)	`st8.spill` & `ld8.fill` preserve register NaT.	RSE automatically preserves register NaT.	`stf.spill` & `ldf.fill` preserve NaTVal.
Data Speculative State (ALAT)	Software must `invala.e` a register's ALAT state when restoring the register.	RSE and ALAT manage stacked register's ALAT state automatically.	Software must `invala.e` a register's ALAT state when restoring the register.

4.1.1 Preserving General Registers

The Itanium general register file is partitioned into two register sets: GR0-31 are termed the **static general registers** and GR32-127 are termed the **stacked general registers**. Typically, `st8.spill` and `ld8.fill` instructions are used to preserve the static GRs, and the processor's register stack engine (RSE) automatically preserves the stacked GRs.

Using the `st8.spill` and `ld8.fill` instructions, the general register value and its NaT bit are always preserved and restored in unison. However, these instructions do not save and restore a register's data speculative state in the Advanced Load Address Table (ALAT). To maintain the correct ALAT state, software is therefore required to explicitly invalidate a register's ALAT entry using the `invala.e` instruction when restoring a general register. The Itanium calling conventions avoid such explicit ALAT invalidations by disallowing data speculation to preserved registers (GR4-7) across procedure calls.

Spills and fills of general registers using `st8.spill` and `ld8.fill` cause implicit collection and restoration of the accompanying NaT bits to/from the User NaT collection application register (UNAT). The UNAT register needs to be preserved by software explicitly. The spill and fill instructions derive the UNAT bit index of a spilled/filled NaT bit from the spill/fill memory address and not from the spilled/filled register index. As a result, software needs to ensure that the 512-byte alignment offset[1] of the spill/fill memory address is preserved when a general register is restored. This can be an issue particularly for user context data structures that may be moved around in memory (e.g. a `setjmp()` jump buffer).

Unlike the `st8.spill` and `ld8.fill` instructions, the register stack engine (RSE) preserves not only register values and register NaT bits, but it also manages the stacked register's ALAT state by invalidating ALAT that could be reused by software when the physical register stack wraps. This automatic management of ALAT state across procedure calls permits compilers to use speculative advanced loads (`ld.sa`) to perform cross-procedure call control and data speculation in stacked general registers (GR32-127). Whenever software changes the virtual to physical register mapping of the stacked registers, the ALAT needs to be invalidated explicitly using the `invala` instruction. Typically this happens during process/thread context switches or in `longjmp()` when the register stack is reloaded with a new BSPSTORE. Refer to Section 4.5.1.1, "Non-local Control Transfers (setjmp/longjmp)" on page 2:422.

The RSE collects the NaT bits of the stacked general registers within the RNAT application register and automatically saves and restores accumulated RNAT collections to/from fixed locations within the register stack backing store. RNAT collections are placed on the backing store whenever BSPSTORE bits{8:3} are all one, which results in one RNAT collection for every 63 registers. When software copies a backing store to a new location, it is required to maintain the backing store's 512-byte alignment offset[2] to ensure that the RNAT collections get placed at the proper offset.

4.1.2 Preserving Floating-point Registers

The Itanium architecture encodes a floating-point register's control speculative state as a special unnormalized floating-point number called NaTVal. As a result, Itanium floating-point registers do not have a NaT bit. The architecture provides the `stf.spill` and `ldf.fill` instructions to save

[1] The specific requirement is that (fill_address mod 512) must be equal to (spill_address mod 512).

[2] The specific requirement is that (old_bspstore mod 512) must be equal to (new_bspstore mod 512).

and restore floating-point register values and control speculative state. These instructions always generate a 16-byte memory image regardless of the precision of the floating-point number contained in the register.

Preservation of data speculative state associated with floating-point registers needs to be managed by software. As with the general registers, software is required to explicitly invalidate a register's ALAT entry using the `invala.e` instruction when restoring a floating-point register. The Itanium calling conventions avoid such explicit ALAT invalidations by disallowing data speculation to preserved floating-point registers (FR2-5, FR16-31) across procedure calls.

4.2 Preserving Register State in the OS

The software calling conventions described in the previous section apply to state preservation across procedure call boundaries. When entering the operating system kernel either voluntarily (for a system call) or involuntarily (for handling an exception or an external interrupt) additional concerns arise because the interrupted user's context needs to be preserved in its entirety.

The Itanium architecture defines a large register set: 128 general registers and 128 floating-point registers account for approximately 1 KByte and 2 KBytes of state, respectively. The architecture provides a variety of mechanisms to reduce the amount of state preservation that is needed on commonly executed code paths such as system calls and high frequency exceptions such as TLB miss handlers.

Additionally, Itanium-based operating systems have opportunities to reduce the amount of context they need to save by distinguishing various kernel entry and exit points. For instance, when entering the kernel on behalf of a voluntary system call, the kernel need only preserve registers as outlined by the calling conventions. Furthermore, the operating system can be sensitive to whether the preserved context is coming from the IA-32 or Itanium instruction set, especially since the IA-32 register context is substantially smaller than the full Itanium register set. Ideally, an Itanium-based operating system should use a single state storage structure which contains a field that indicates the amount of populated state.

Table 4-2 summarizes several key operating system points at which state preservation is needed.

Scratch GRs and FRs, the bulk of all state, only need to be preserved at involuntary interruptions resulting from unexpected external interrupts or from exceptions that need to call code written in a high-level programming language. The demarcation of floating-point registers FR32-127 as "scratch" along with architectural support for lazy state save/restore of the floating-point register file allows software to substantially reduce the overhead of preserving the scratch FRs. See Section 4.2.2 for details.

In principal, preserved GRs and FRs need not be spilled/filled when entering the kernel. Whatever function is called from the low-level interruption handler or the system call entry point will itself observe the calling conventions and preserve the registers. The only occasion when preserved registers need to be spilled/filled is on a process or thread context switch. However, many operating systems provide `get_context()` functions that provide user context upon demand. Although such functions are called infrequently, many operating systems prefer to pay the penalty of spilling preserved registers at system call and at interruption entry points to avoid the complexity of piecing together user state from various potentially unknown kernel stack locations on demand.

Fortunately, the amount of preserved Itanium general register state is relatively small, and the Itanium architecture provides additional mechanisms for lazy floating-point state management. See Section 4.2.2 for details.

Table 4-2. Register State Preservation at Different Points in the OS

Register Type	Number of Registers	System Call (Voluntary)	Lightweight Interruptions[a] (Involuntary)	Heavyweight Interruptions[b] (Involuntary)	Process/Thread Context Switch (Voluntary)
Scratch GRs	23	no spill/fill required	Untouched (use banked registers)	spill/fill required	no spill/fill required (done at interruption)
Preserved GRs	4	no spill/fill required	Untouched (use banked registers)	no spill/fill required	spill/fill required
Stacked GRs	96	Backing Store Switch	Untouched	Backing Store Switch	Synchronous Backing Store Switch using flushrs[c]
Scratch FRs	106	no spill/fill required	Untouched	spill/fill required	no spill/fill required (done at interruption)
Preserved FRs	20	no spill/fill required	Untouched	no spill/fill required	spill/fill required

a. For details on lightweight interruption handlers refer to Section 3.4.1, “Lightweight Interruptions” on page 2:410.

b. For details on heavyweight interruption handlers refer to Section 3.4.2, “Heavyweight Interruptions” on page 2:411.

c. Refer to Section 6.11.3, “Synchronous Backing Store Switch” on page 1:131 for details.

Stacked GRs are managed by the register stack engine (RSE). On process/thread context switches the operating system is required to completely flush the register stack to its backing store in memory (using the `flushrs` instruction). In cases where the operating system knows that it will return to the user process along the same path, e.g. in system calls and exception handling code, the Itanium architecture allows operating systems to switch the register stack backing store without having to flush all stacked registers to memory. This allows such kernel entry points to switch from the user’s to the kernel’s backing store without causing any memory traffic, as described in the next section.

4.2.1 Preservation of Stacked Registers in the OS

A switch from a thread of execution into the operating system kernel, whether on behalf of an involuntary interruption or a voluntary system call, requires preservation of the stacked registers. Instead of flushing all dirty stacked register’s to memory, the RSE can be used to automatically preserve the stacked registers of the interrupted context. Automatic preservation offers performance benefits: the register stack may contain only a handful of dirty registers, system call parameters can be passed on the register stack, and, upon return to the interrupted context the `loadrs` instruction only needs to restore registers that were actually spilled to memory. Since system call rates scale with processor performance, the RSE offers a key method for reducing the kernel’s execution time of a system call.

To ensure operating system integrity the RSE requires a valid backing store (i.e. one with a valid page mapping). The validity of the current backing store depends on the interrupted context. If the interrupted context is itself a kernel thread, then its backing store is in a known state, and no backing store switch is required (assuming that kernel interruptions are nested). If the interrupted context is a user process, then the backing store could be pointing at an invalid region of memory, and software is required to redirect the RSE at a kernel backing store. Section 6.11.1, "Switch from Interrupted Context" on page 1:131 describes the code sequence to switch the RSE backing store without causing memory traffic.

If the kernel redirects the backing store to a kernel memory region, then the kernel must restore the backing store of the interrupted context prior to resumption of the interrupted context. The kernel must also restore the register stack to its interrupted state by manually pulling the spilled registers from the backing store. The kernel uses the `loadrs` instruction to restore stacked registers from the backing store. The `loadrs` instruction requires the backing store pointer to align with any registers spilled from the interrupted context. Thus the kernel should have paired all function calls (`br.call` instructions) with function returns (`br.ret` instructions), or manually manipulated the kernel backing store pointer, so that all kernel contents have been removed from the kernel backing store prior to the `loadrs`. After loading the stacked registers, the kernel can switch to the backing store of the interrupted frame. This code sequence is described in Section 6.11.1, "Switch from Interrupted Context" on page 1:131.

The kernel may occasionally gather the complete interrupted user context, such as to satisfy a debugger request or to provide extended information to a user signal handler. To provide the preserved register stack contents, including NaT values, the kernel must extract the user context values from its backing store.

4.2.2 Preservation of Floating-point State in the OS

A full preservation of Itanium floating-point register file requires approximately 2 KBytes of memory. To reduce the frequency of such large register spills and fills, the Itanium architecture offers additional mechanisms for lazy floating-point state management. These features allow the system programmer to eliminate many unnecessary floating-point state spills and fills especially around voluntary and involuntary entries into the kernel, e.g. around system calls, external interrupts and exceptions. Lazy state preservation can provide a significant reduction of memory traffic and hence faster interrupt handlers and system calls, especially since most interrupt handlers and much system code rarely perform floating-point computations.

The 126 non-constant floating-point registers are architecturally divided into the lower set (FR2-31) and the higher set (FR32-127). The Itanium architecture provides two floating-point register set "modified" bits, PSR.mfl and PSR.mfh, which are set by hardware upon a write to any register in the lower and higher sets, respectively. The "modified" bits are accessible to a user process through the user mask. Additionally, two "disabled" bits, PSR.dfl and PSR.dfh, are accessible to the privileged software alone. Setting a "disabled" bit causes a fault into the disabled-fp vector upon first use (read or write) of the corresponding register set.

As mentioned earlier, an involuntary kernel entry (e.g. interruption) needs to preserve all scratch floating-point registers. Instead of blindly always spilling all registers, state spills can be conditionalized upon the "modified" bits in the PSR. Additionally, the "disabled" bits allow a deferred, or lazy, approach to both spills and fills. This is particularly useful for "on demand" state motion in an involuntary interruption handler that does not use many floating-point registers. To perform deferred spills on the high set, the handler sets PSR.dfh immediately upon entry. Any reference to a floating-point register in the high set will then fault into the disabled-fp vector which spills the corresponding state to a prearranged store before allowing use within the handler. Lazy state restoration is performed in a similar manner: the handler sets the "disabled" bit just before exit, causing the first reference by the interrupted context to the disabled set to fault into the kernel's disabled floating-point vector which can then restore the appropriate state. Note the importance of agreeing upon prearranged stores for deferred spill/fill policies and the need for a mechanism to communicate a past fill or spill.

At process or thread context switches all preserved floating-point registers need to be context switched. The higher (scratch) set is also managed here if the context-switch was occasioned by an involuntary interruption (e.g. timer interrupt) which did not already spill the higher set. Use of the "modified" bits by the OS to determine if the appropriate register set is "dirty" with previously unsaved data can help avoid needless spills and fills.

The "modified" bits are intentionally accessible through the user mask so that a user process can provide hints to the OS code about its register liveness requirements. Clearing PSR.mfh, for instance, suggests that the user process does not see the higher register set as containing useful data anymore.

4.3 Preserving ALAT Coherency

As described in Section 4.4.5.3, "Detailed Functionality of the ALAT and Related Instructions" on page 1:58 of Volume 1, software is required to explicitly invalidate the entire ALAT using the `invala` instruction whenever the virtual to physical register mapping is changed. Typically this occurs when the `clrrb` instruction is used, when a synchronous backing store switch is performed (e.g. in a user-level or kernel thread context switch), or when software "discontinuously" remaps the register to backing store mapping by resetting BSPSTORE (e.g. by calling `longjmp()`).

When returning to a user-process after servicing an involuntary interruptions, an Itanium-based operating system is required to invalidate the entire ALAT using the `invala` instruction. This is required because the operating system may have targeted advanced loads at scratch registers, and thereby altered the user-visible ALAT state.

When returning from a system call, however, full ALAT invalidations can be avoided by using `invala.e` instructions to selectively invalidate ALAT entries of all preserved registers (GR4-7, FR2-5, and FR16-31), or by ensuring that these registers where never accessible to software during the system call (see Section 4.2.2 for details). This works, because at the system call entry user-code may not have any dependencies on the state of the scratch registers.

4.4 System Calls

Reducing the overhead associated with system calls becomes more important as processor efficiency increases. As processor frequencies and pipeline lengths increase, the typical overhead associated with flushing the processor pipeline to effect privilege domain crossings is increased. To reduce system call overhead, the Itanium architecture provides an efficient "enter privileged code" (`epc`) instruction (page 3:48 of Volume 3) that can be paired with the demoting branch return. Additionally, the Itanium architecture provides the traditional `break` instruction (page 3:26 of Volume 3) to enter privileged mode, that is typically paired with the `rfi` instruction (page 3:208 of Volume 3) to return to user mode.

The epc instruction offers higher efficiency than the break instruction for invoking a kernel system call. Whereas a break instruction will always cause a pipeline flush to change privilege level, the epc is designed not to. The break instruction also passes the system call number as a parameter, and requires a table lookup with an indirect branch to the system call. With the epc instruction, the user application can directly branch to the system call code.

More information about epc based system calls is provided in Section 4.4.1. More information about break based system calls is provided in Section 4.4.2. Regardless of whether the epc or break instruction are used, an Itanium-based operating system needs to check the integrity of system call parameters. In addition to traditional integrity checking of the passed parameter values, the system call handler should inspect system call parameters for set NaT bits as described in Section 4.4.3.

4.4.1 epc/Demoting Branch Return

To execute a system call with epc, a user system call stub branches to an execute-only kernel page containing the system call, using the br.call instruction. The kernel page executes an epc to raise the privilege level. The privilege level is raised to the privilege level of the page mapping corresponding to the instruction address of the epc instruction. The page mapping must be execute-only (see Section 4.1.1.6, "Page Access Rights" on page 1:46 for details).

After the kernel completes its system call, it returns to the user system call stub with a br.ret instruction. The br.ret demotes the privilege level, by restoring the privilege level contained within the PFS application register (PFS.ppl). To ensure operating system integrity epc checks that the PFS.ppl field is no greater than the PSR.cpl at the time the epc is executed.

As described in Section 4.2.1, interruptions and system calls in a typical Itanium-based operating system need to switch to the kernel register stack backing store upon kernel entry. The epc instruction does not disable interrupts nor does it switch the processor to the kernel backing store. As a result, code directly following the epc instruction that runs at increased privilege level is still running on the caller's backing store. It is recommended that software disable external interrupts right after the epc until the switch to the kernel backing store has been completed. Additionally, low-level operating system handlers should not only use IPSR.cpl, but should also check BSPSTORE, to determine whether they are running on the kernel backing store (imagine an external interrupt being delivered on the first instruction after the epc).

4.4.2 break/rfi

The break instruction, when issued in the *i*, *f*, and *m* syllables, specifies an arbitrary 21-bit immediate value. The kernel can choose a specific break immediate value to differentiate system calls from other usage of the break instruction (such as debug). The break instruction jumps to the break fault handler, which should be a valid address mapping for each user application, and raises the privilege mode to the most privileged level.

The system call number is an additional parameter passed to the kernel when invoking a system call via the break instruction. The system call number must reside in a fixed location. If stored within GR32, then the system call stub must rearrange its input parameters to map to the register stack starting at GR33. This register jostling can be avoided by passing the system call number through a scratch static general register or by using the break immediate itself. Additionally, the system call can utilize all eight input registers of the register stack for system call parameters.

4.4.3 NaT Checking for NaTs in System Calls

In addition to regular range/value checking on system call arguments, Itanium-based operating systems need to additionally ensure that system call arguments passed in by a user application do not have any NaT bits set. The following code fragment can be used:

```
        mov mask = 0xff
        clrrrb
        ;;
// create register stack frame with only output registers for system call args
        alloc tmp = ar.pfs, 0, 0, 8, 0
        shl mask = mask, syscall_arg_count
        ;;
        mov pr = mask, 0xff00         // define p8 .. p15
        ;;
        cmp.eq p7 = r0, r0            // set p7 to true
        ;;
// test for NaT bits in the input arguments
(p8)    cmp.eq.and p7 = r32, r32      // and type compare clears p7 if r32 is NaT
(p9)    cmp.eq.and p7 = r33, r33
(p10)   cmp.eq.and p7 = r34, r34
(p11)   cmp.eq.and p7 = r35, r35
(p12)   cmp.eq.and p7 = r36, r36
(p13)   cmp.eq.and p7 = r37, r37
(p14)   cmp.eq.and p7 = r38, r38
(p15)   cmp.eq.and p7 = r39, r39
(p7)    br.cond.sptk ok_arguments     // No NaTs found
;;
// p7 was cleared by at least one NaT argument
```

4.5 Context Switching

This section discusses context switching at the user and kernel levels.

4.5.1 User-level Context Switching

4.5.1.1 Non-local Control Transfers (setjmp/longjmp)

A non-local control transfer such as the C language `setjmp()`/`longjmp()` pair requires software to correctly handle the register stack and the RSE. The register stack provides the BSP application register which always contains the backing store address of the current GR32. This permits execution of a `setjmp()` without having to manipulate any register stack or RSE state. All register stack and RSE manipulation is postponed to the much less frequent `longjmp()`.

In `setjmp()` only the RSC, PFS and BSP application registers have to be preserved. This can be accomplished by reading these registers, and without having to disable the RSE. The preserved values will be referred to as `setjmp_rsc`, `setjmp_pfs`, and `setjmp_bsp` further on.

In `longjmp()` restoration of the appropriate register stack and RSE state is more involved, and software needs to take the following steps:

1. Stop RSE by setting RSC.mode bits to zero.
2. Read current BSPSTORE (referred to as `current_bspstore` further down).
3. Find `setjmp()`'s RNAT collection (`rnat_value`).
    a. Compute the backing store location of `setjmp()`'s RNAT collection as follows:

    ```
    rnat_collection_address{63:0} = setjmp_bsp{63:0} | 0x1F8
    ```

    The RNAT location is computed by setting bits{8:3} of `setjmp()`'s BSP to all ones. This is where `setjmp()`'s RNAT collection will have been spilled to memory.

    b. If (`current_bspstore > rnat_collection_address`), then the required RNAT collection has already been spilled to the backing store.

    c. Otherwise if (`current_bspstore <= rnat_collection_address`), the required RNAT collection is incomplete and is still contained in the register stack. To materialize the complete RNAT collection, flush the register stack to the backing store using a `flushrs` instruction.

    d. Finally, load `rnat_value` from `rnat_collection_address` in memory.
4. Invalidate the contents of the register stack as follows:
    a. Allocate a zero size register stack frame using the `alloc` instruction.

    b. Write RSC.loadrs field with all zeros and execute a `loadrs` instruction.

    c. Invalidate the ALAT using the `invala` instruction.
5. Restore `setjmp()`'s register stack and RSE state as follows:
    a. Write BSPSTORE with `setjmp_bsp`.

    b. Write RNAT with `rnat_value`.

    c. Write RSC with `setjmp_rsc`.

    d. Write PFS with `setjmp_bsp`.
6. Restore `setjmp()`'s return IP into BR7.
7. Return from `longjmp()` into `setjmp()`'s caller using `br.ret` instruction.

4.5.1.2 User-level Co-routines

The following steps need to be taken to execute a voluntary user-level thread switch.

1. Save all preserved register state of outgoing thread to memory stack. Refer to Section 4.1 for details on preservation of general and floating-point registers.
2. Preserve predicate, branch, and application registers.
3. Flush outgoing register stack to backing store, and switch to incoming thread's backing store as described in Section 6.11.3, "Synchronous Backing Store Switch" on page 1:131. This code sequence includes ALAT invalidation.
4. Switch thread memory stack pointers.
5. Restore incoming thread's predicate, branch, and application registers.
6. Restore incoming thread's preserved register state.

4.5.2 Context Switching in an Operating System Kernel

4.5.2.1 Thread Switch within the Same Address Space

To switch between different threads in the same address space the following steps are required:

1. Application architecture state associated with each thread (GRs, FRs, PRs, BRs, ARs) are saved and restores as if this were a user-level coroutine. This is described in Section 4.5.1.2.
2. Memory Ordering: to preserve correct memory ordering semantics the context switch routine needs to fence `fc` and other memory references by performing a `sync.i` and an `mf` instruction. More details on memory ordering are given in Section 2.3.

4.5.2.2 Address Space Switching

When an operating system switches address spaces it needs to perform the same steps as a same address space thread switch (described in the previous section). Additionally, however between the saves of the outgoing and the restores of the incoming process, the operating system context switch handler is required to:

1. Save the contents of the protection key registers associated with the outbound context, and then invalidate the protection key registers.
2. Save the default control register (DCR) of the outbound context (if the DCR is maintained on a per-process basis).
3. Save the region registers of the outbound address space.
4. Restore the region registers of the inbound address space.
5. Restore the default control register (DCR) of the inbound context (if the DCR is maintained on a per-process basis).
6. Restore the contents of the protection key registers associated with the inbound context.

Memory Management 5

This chapter introduces various memory management mechanisms of the Itanium architecture: region register model, protection keys, and the virtual hash page table usage models are described. This chapter also discusses usage of the architecture translation registers and translation caches. Outlines are provided for common TLB and VHPT miss handlers.

5.1 Address Space Model

The Itanium architecture provides a byte-addressable 64-bit virtual address space. The address space is divided into 8 equally-sized sections called regions. Each region is 2^{61} bytes in size and is tagged with a unique region identifier (RID). As a result, the processor TLBs can hold translations from many different address spaces concurrently, and need not be flushed on address space switches. The regions provide the basic virtual memory architecture to support multiple address space (MAS) operating systems.

Additionally, each translation in the TLB contains a protection key that is matched against a set of software maintained protection key registers. The protection keys are orthogonal to the region model and allow efficient object sharing between different address spaces. The protection key registers provide the basic virtual memory architecture to support single address space (SAS) operating systems.

5.1.1 Regions

For each of the eight regions, there is a corresponding region register (RR), which contains a RID for that region. The operating system is responsible for managing the contents of the region registers. RIDs are between 18 and 24 bits wide, depending on the processor implementation. This allows an Itanium-based operating system to uniquely address up to 2^{24} address spaces each of which can be up to 2^{61} bytes in virtual size. An address space is made accessible to software by loading its RID into one of the eight region registers.

Address Translation: The upper 3 bits of a 64-bit virtual address (bits 63:61) identify the region to which the address belongs; these are called the virtual region number (VRN) bits. When a virtual address is translated to a physical address, the VRN bits select a region register which provides the RID used for this translation. Each TLB entry contains the RID tag bits for the translation it maps; these are matched against the RID bits from the selected region register when the TLB is looked up during address translation. Address translation only succeeds if the RID and VPN bits from the virtual address match the RID and VPN bits from the TLB entry. Note that the VRN bits are used only to select the region register, are not matched against the TLB entries.

Inserting/Purging of Translations: When a translation is inserted into the processor TLBs (either by software, or by the processor’s hardware page walker), the VRN bits of the virtual address translation being inserted are used only to index the corresponding region register; they are not inserted into the TLB. Likewise, when software purges a translation from the processor’s TLBs, the

VRN bits of the address used for the purge are used only to index the corresponding region register and are not used to find a matching translation. Only the RID and VPN bits are used to find overlapping translations in the TLBs.

The fact that the VRN bits are not contained in the processor TLB allows the same address space (identified by a RID) to be referenced through any of the eight region registers. In other words, the combination of RID and VPN establishes a unique 85-bit virtual address, regardless of which VRN (and region register) was used to form the pair. Independence of VRN allows easy creation of temporary virtual mappings of an address space and can accelerate cross-address space copying as described in Section 5.1.1.3.

5.1.1.1 RID Management

Before a RID that has been used for one address space can be reused for another address space, all TLB entries relating to the first address space have to be purged. In general, this will require a complete flush of the TLBs of all processors in the system. This can be accomplished by performing an IPI to all processors and executing the `ptc.e` loop described in Section 5.2.2.2.2 on each processor in the TLB coherence domain.

A more efficient alternative, depending on the size of the defunct address space, might be to perform a series of `ptc.ga` operations on one processor to tear down just the translations used by the recycled RID. Some processor implementations support an efficient region-wide purge page size such that this can be accomplished with a single `ptc.ga` operation.

The frequency of these global TLB flushes can be reduced by using a RID allocation strategy that maximizes the time between use and reuse of a RID. For example, RIDs could be assigned by using a counter that is as wide as the number of implemented RID bits and that is incremented after every assignment. Only when the RID counter wraps around it is necessary to do a global TLB flush. After the flush the operating system can either remember the in-use RIDs or it can re-assign new RIDs to all currently active address spaces.

5.1.1.2 Multiple Address Space Operating Systems

Multiple address space (MAS) operating systems provide a separate address space for each process. Typically, only when a process is running is its address space visible to software.

The application view of the virtual address space in the MAS OS model is a contiguous 64-bit address space, though normally not all of this virtual address space is accessible by the application. At least one of the 8 regions must be used to map the OS itself so that the OS can handle interruptions and system services invoked by the application.

The OS chooses a region ID and a region (e.g. region 7) into which to map itself during the boot process and usually does not change this mapping after enabling address translation. The other seven regions may be used to map process-private code and data; code and data that are shared amongst multiple processes; to map large files; temporary mappings to allow efficient cross-address space copies (see Section 5.1.1.3); and, for operating systems which use it, the long format VHPT.

In a MAS OS, the RID bits act as an address space identifier or tag. For each process-private region, a unique RID is assigned to that process by the OS. If a process needs multiple process-private regions (e.g. the process requires a private 64-bit address space), the OS assigns multiple unique RIDs for each such region. Because each translation in the processor’s TLBs is tagged with its RID, the TLBs may contain translations from many different address spaces (RIDs) concurrently. This obviates the need for the OS to purge the processor's TLBs upon an address space switch. When the OS performs a context switch from process A to process B, the OS need only remove process A's private RIDs from the CPU's region registers and replace them with process B's private RIDs.

5.1.1.3 Cross-address Space Copies in a MAS OS

The use of regions, region registers, and RIDs provides a mechanism for efficient address space-to-address space copies. Because translations are tied to RIDs and not to a particular static region, a MAS OS can easily copy a memory range from one address space to another by temporarily remapping the target memory location to another region. This remapping is accomplished simply by placing the RID to which the target location belongs into a different region register and then performing the copy from source to target directly.

For example, assume a MAS OS wishes to copy and 8-byte buffer from virtual address 0x0000000000A00000 of the currently executing process (process A) to virtual address 0x0000000000A00000 of another process (process B):

```
        mov r2 = 2
        mov r3 = process_b_rid
        movl r4 = 0x0000000000A00000
        movl r5 = 0x4000000000A00000;;;   // reference process B through RR[2]
        mov rr[r2] = r3 ;;                // put process B RID into RR[2]
        srlz.d                            // serialize RR write
copyloop:
        ld8 r6 = [r4] ;;                  // read buffer from process A addr space
        st8 [r5] = r6                     // store buffer into process B addr
space
    (p4)br copyloop                       // loop until done
        mov r3 = original_rr2_rid ;;
        mov rr[r2] = r3 ;;                // restore RR[2] RID
        srlz.d                            // serialize RR write
```

When the OS switches to process B and places process B's RID into RR[0] and resumes execution of process B, the process can reference the message via virtual address 0x0000000000A00000. Note that no new translations need to be created to make the sequence shown above work; because translations are tagged by RID and not by region, all existing translations for process B's address space are visible regardless of which region the reference is made to, as long as the region register for that region contains the correct process B RID. Note that the sequence shown above is intended for illustrative purposes only; the OS may need to perform other steps as well to perform a cross-address space copy.

5.1.2 Protection Keys

The Itanium architecture provides two mechanisms for applying protection to pages. The first mechanism is the access rights bits associated with each translation. These bits provide privilege level-granular access to a page. The second mechanism is the protection keys. Protection keys permit domain-granular access to a page. These are especially useful for mapping shared code and data segments in a globally shared region, and for implementing domains in a single address space (SAS) operating system.

Protection key checking is enabled via the PSR.pk bit. When PSR.pk is 1, instruction, data, and RSE references go through protection key access checks during the virtual-to-physical address translation process.

All processors based on the Itanium architecture implement at least 16 protection key registers (PKRs) in a protection key register cache. The OS is responsible for maintaining this cache and keeping track of which protection keys are present in the cache at any given time.

Each protection key register contains the following fields:

- v – valid bit. When 1, this register contains a valid key, and is checked during address translation whenever protection keys are enabled (PSR.pk is 1).
- wd – write disable. When 1, write permission is denied to translations which match this protection key, even if the data TLB access rights permit the write.
- rd – read disable. When 1, read permission is denied to translations which match this protection key, even if the data TLB access rights permit the read.
- xd – execute disable. When 1, execute permission is denied to translations which match this protection key, even if the instruction TLB access rights give execute permission.
- key – protection key. An 18- to 24-bit (depending on the processor implementation) unique key which tags a translation to a particular protection domain.

When protection key checking is enabled, the protection key tagged to a referenced translation is checked against all protection keys found in the protection key register cache. If a match is found, the protection rights specified by that key are applied to the translation. If the access being performed is allowed by the matching key, the access succeeds. If the access being performed is not allowed by the matching key (e.g. instruction fetch to a translation tagged with a key marked 'xd'), a Protection Key Permission fault is raised by the processor. The OS may then decide whether to terminate the offending program or grant it the requested access.

If no match is found, a Protection Key Miss fault is raised by the processor, and the OS must insert the correct protection key into the PKRs and retry the access.

Protection keys can be used to provide different access rights to shared translations to each process. For example, assume a shared data page is tagged with a protection key number of 0xA. Two processes share this data page: one is the producer of the data on this page, and the other is only a consumer. When the producer process is running, the OS will insert a valid PKR with the protection key 0xA and the 'wd' and 'rd' bits cleared, to allow this process to both read and write this page. When the consumer process is running, the OS will insert a valid PKR with the protection key 0xA and the 'rd' bit cleared, to allow this process to read from the page. However, the 'wd' bit for this PKR will be set when the consumer process is running to prevent it from writing the page.

The processor hardware has no notion of which protection keys belong to which process. The only check the hardware performs is to compare the protection key from the translation to any valid protection keys in the PKR cache. On a context switch, the OS must purge any valid protection keys from the PKRs which would provide access rights to the switched-to context that are not allowed. The OS may purge an existing PKR by performing a move to PKR instruction with the same key as the existing PKR, but with the PKR valid bit set to 0.

Protection keys can be read from the processor's data TLBs via the `tak` instruction. However, instruction TLB key values cannot be read directly. Software must keep track of these values in its own data structures.

5.1.2.1 Single Address Space Operating Systems

Processes in a single address space (SAS) OS all cohabit a global address space. SAS operating systems running on a processor based on the Itanium architecture can view the RID bits as effectively extending the single virtual address space to between 79 and 85 bits (depending on the number of RID bits implemented by the processor). This address space is then divided into between 2^{18} and 2^{24} 61-bit regions, up to eight of which may be accessed concurrently.

Note that there is no “SAS OS” or “MAS OS” mode in the Itanium architecture. The processor behavior is the same, regardless of the address space model used by the OS. The difference between a SAS OS and a MAS OS is one of OS policy: specifically how the RIDs and protection keys are managed by the OS, and whether different processes are permitted to share RIDs for their private code and data. Multiple, unrelated processes in a SAS OS may share the same RID for their private pages; it is the responsibility of the OS to use protection keys and the protection key registers (PKRs) to enforce protection. In a MAS OS, the unique per-process RIDs enforce this protection.

Hybrid SAS/MAS models that combine unique RIDs for process-private regions and shared RIDs with protection keys for per-page memory protection in shared regions are also possible.

5.2 Translation Lookaside Buffers (TLBs)

All processors based on the Itanium architecture implement one or more translation lookaside buffers (TLBs) for fast virtual-to-physical address translation. The architecture provides instructions for managing instruction and data TLBs as separate structures.

Both the instruction and data TLBs are further divided into a set of translation registers (TRs), which are managed exclusively by software and are “locked down” to pin critical address translations (e.g. kernel memory); and a set of translation cache entries (TCs), which can be managed by both software and the processor hardware. The TRs are divided into slots, each of which are individually addressable on insertion by software. The TCs are treated as a set associative cache and are not addressable by software. The TC replacement policy is determined by software. All processor models implement at least 8 instruction and 8 data TRs, and at least one instruction and 1 data TC entry.

Software inserts translations into the TLBs via insertion instructions. There are four variants of insertion instructions. `itr.i` and `itr.d` insert a translation into the specified instruction or data TR slot, respectively. `itc.i` and `itc.d` insert a translation into a hardware-selected instruction or data TC entry, respectively.

Software TR purge instructions also distinguish between the instruction and data TRs (`ptr.i`, `ptr.d`). TC purge instructions do not.

5.2.1 Translation Registers (TRs)

Once a translation is inserted by software into a TR, it remains in that TR until either the translation is overwritten by software, or the translation is purged. TRs are used by the OS to pin critical address translations; all memory references made to a TR translation will always hit the TLB and will never cause the processor's hardware page walker to walk the VHPT or raise a fault. Examples of memory areas that the OS might cover with one or more TRs are the Interruption Vector Table, critical interruption handlers not contained completely in the Interruption Vector Table, the root-level page table entries, the long format VHPT, and any other non-pageable kernel memory areas.

Two address translations are said to overlap when one or more virtual addresses are mapped by both translations. Software must ensure that translations in an instruction TR never overlap other instruction TR or TC translations; likewise, software must ensure that translations in a data TR never overlap other data TR or TC translations. If an overlap is created, the processor will raise a Machine Check Abort.

The processor hardware will never overwrite or purge a valid TR. TRs that are currently unused may be used by the processor hardware as extra TC entries, but if software subsequently inserts a translation into an unused a TR, the TC translation will be purged when the insertion is executed.

5.2.1.1 TR Insertion

To insert a translation into a TR, software performs the following steps:

1. If PSR.ic is 1, clear it and execute a `srlz.d` instruction to ensure the new value of PSR.ic is observed.
2. Place the base virtual address of the translation into the IFA control register.[1]
3. Place the page size of the translation into the ps field of the ITIR control register. If protection key checking is enabled, also place the appropriate translation key into the key field of the ITIR control register. See below for an explanation of protection keys.
4. Place the slot number of the instruction or data TR into which the translation is be inserted into a general register.
5. Place the base physical address of the translation into another general register.
6. Using the general registers from steps 4 and 5, execute the `itr.i` or `itr.d` instruction.

A data or instruction serialization operation must be performed after the insert (for `itr.d` or `itr.i`, respectively) before the inserted translation can be referenced.

Software may insert a new translation into a TR slot already occupied by another valid translation. However, software must perform a TR purge to ensure that the overwritten translation is no longer present in any of the processor's TLB structures.

Instruction TR inserts will purge any instruction TC entries which overlap the inserted translation, and may purge any data TC entries which overlap it. Data TR inserts will purge any data TC entries which overlap the inserted translation and may purge any instruction TC entries which overlap it.

Software may insert the same (or overlapping) translation into both the instruction TRs and the data TRs. This may be desirable for locked pages which contain both code and data, for example.

5.2.1.2 TR Purge

To purge a TR from the TLBs, software performs the following steps:

1. Place the base virtual address of the translation to be purged into a general register.[2]

[1] The upper 3 bits (VRN) of this address specify a region register whose contents are inserted along with the rest of the translation. See Section 5.1.1 for details.

2. Place the address range in bytes of the purge into bits {7:2} of a second general register.
3. Using these two GRs, execute the `ptr.d` or `ptr.i` instruction.

A data or instruction serialization operation must be performed after the purge (for `ptr.d` or `ptr.i`, respectively) before the translation is guaranteed to be purged from the processor's TLBs.

Note: The TR purge instruction operates independently of the slot into which the translation was originally inserted.

A `ptr.d` instruction will never purge an overlapping translation in an instruction TR, but may purge an overlapping translation in an instruction TC; likewise, a `ptr.i` instruction will never purge an overlapping translation in a data TR, but may purge an overlapping translation in a data TC.

A TR purge does not modify the page tables nor any other memory location, nor does it affect the TLB state of any processor other than the one on which it is executed.

5.2.2 Translation Caches (TCs)

The TC array acts as a cache of the dynamic working set for data and instruction translations. It is managed by software (via `itc` and `ptc` instructions) and, optionally by hardware, if the processor provides a hardware page walker (HPW) and the walker is enabled. See Section 5.3 below.

The size, associativity, and replacement policy of the TC array are implementation-dependent. With the exception of the forward progress rules defined in Section 4.1.1.2, "Translation Cache (TC)", software cannot depend on the existence or life-span of a TC translation, as a TC entry may be replaced or invalidated by the hardware at any time.

5.2.2.1 TC Insertion

To insert a TC entry, software performs the following steps:

1. If PSR.ic is 1, clear it and execute a `srlz.d` instruction to ensure the new value of PSR.ic is observed.
2. Place the base virtual address of the translation into the IFA control register.[3]
3. Place the page size of the translation into the ps field of the ITIR control register. If protection key checking is enabled, also place the appropriate translation key into the key field of the ITIR control register. See below for an explanation of protection keys.
4. Place the base physical address of the translation into a general register.
5. Using the general register from step 4, execute the `itc.i` or `itc.d` instruction.

A data or instruction serialization operation must be performed after the insert (for `itc.d` or `itc.i`, respectively) before the inserted translation can be referenced.

[2] The upper 3 bits (VRN) of this address specify a region register whose contents are used as part of the translation to be purged. See Section 5.1.1 for details.

[3] The upper 3 bits (VRN) of this address specify a region register whose contents are inserted along with the rest of the translation. See Section 5.1.1 for details.

Instruction TC inserts always purge overlapping instruction TCs and may purge overlapping data TCs. Likewise, data TC inserts always purge overlapping data TCs and may purge overlapping instruction TCs.

5.2.2.2 TC Purge

There are several types of TC purge instructions. Unlike the other TLB management instructions, the TC purge instructions do not distinguish between instruction and data translations; they will purge any matching translations in either the data or instruction TC arrays.

5.2.2.2.1 ptc.l

The most basic TC purge is the local TC purge instruction (`ptc.l`). To purge a TC from the local processor TLBs, software performs the following steps:

1. Place the base virtual address of the translation to be purged into a general register.[4]
2. Place the address range in bytes of the purge into bits {7:2} of a second general register.
3. Using these two GRs, execute the `ptc.l` instruction.

A data or instruction serialization operation must be performed after the `ptc.l` before the translation is guaranteed to be no longer visible to the local data or instruction stream, respectively.

The `ptc.l` instruction does not modify the page tables nor any other memory location, nor does it affect the TLB state of any processor other than the one on which it is executed.

5.2.2.2.2 ptc.e

To purge all TC entries from the local processor's TLBs, software uses a series of `ptc.e` instructions. Software must call the PAL_PTCE_INFO PAL routine at boot time to determine the parameters needed to use the `ptc.e` instruction. Specifically, PAL_PTCE_INFO returns:

- tc_base – an unsigned 64-bit integer denoting the beginning address to be used by the first `ptc.e` instruction in the purge loop.
- tc_counts – two unsigned 32-bit integers packed into a 64-bit parameter denoting the loop counts of the outer and inner purge loops. count1 (outer loop) is contained in bits {63:32} of the parameter, and count2 (inner loop) is contained in bits {31:0} of the parameter.
- tc_strides – two unsigned 32-bit integers packed into a 64-bit parameter denoting the loop stride of the outer and inner purge loops. stride1 (outer loop) is contained in bits {63:32} of the parameter, and stride2 (inner loop) is contained in bits {31:0} of the parameter.

Software then executes the following sequence:

```
disable_interrupts();
addr = tc_base;
for (i = 0; i < count1; i++) {
    for (j = 0; j < count2; j++) {
        ptc.e addr;
        addr += stride2;
    }
    addr += stride1;
}
enable_interrupts();
```

[4] The upper 3 bits (VRN) of this address specify a region register whose contents are used as part of the translation to be purged. See Section 5.1.1 for details.

A data or instruction serialization operation must be performed after the sequence shown above before the translations are guaranteed to be no longer visible to the local data or instruction stream, respectively.

The `ptc.e` instruction does not modify the page tables nor any other memory location, nor does it affect the TLB state of any processor other than the one on which it is executed.

5.2.2.2.3 ptc.g, ptc.ga

The Itanium architecture supports efficient global TLB shootdowns via the `ptc.g` and `ptc.ga` instructions. These instructions obviate the need for performing inter-processor interrupts to maintain TLB coherence in a multi-processor system. A TLB coherence domain is defined as a group of processors in a multiprocessor system which maintain TLB coherence via hardware.

For the remainder of this section, **`ptc.g`** refers to both the `ptc.g` and `ptc.ga` instructions, except where otherwise noted.

Only one `ptc.g` operation can be in progress at any time, otherwise one or more of the processors in the system may raise a Machine Check Abort. To guarantee that only one `ptc.g` operation is in progress at a time, software should create a shootdown lock variable which must be acquired before issuing a `ptc.g`, and released after the `ptc.g` has completed.

A `ptc.g` instruction is a release operation; all memory references that precede a `ptc.g` in program order are made visible to all other processors before the `ptc.g` is made visible. To guarantee visibility of the `ptc.g` prior to a particular point in program execution, software must use another release operation or a memory fence.

To purge a translation from all TLBs in the coherence domain, software performs the following steps:

1. Acquire the shootdown lock variable.
2. Place the base virtual address of the translation to be purged into a general register.
3. Place the address range in bytes of the purge into bits {7:2} of a second general register.
4. Using these two GRs, execute the ptc.g instruction. Note that the `ptc.g` instruction must be followed by a stop.
5. Release the shootdown lock variable.

Global purges can be batched together by performing multiple `ptc.g` instructions prior to releasing the lock.

A data or instruction serialization operation must be performed after the sequence shown above before the translations are guaranteed to be no longer visible to the local data or instruction stream, respectively. To guarantee the translations are no longer visible on remote processors, a release operation or memory fence instruction is required after the `ptc.g` instruction.

The `ptc.g` instruction does not modify the page tables nor any other memory location. It affects both the local and all remote TC entries in the TLB coherence domain. It does not remove translations from either local or remote TR entries, and if a `ptc.g` overlaps a translation contained in a TR on either the local processor or on any remote processor in the coherence domain, the processor containing the overlapping translation will raise a Machine Check Abort.

The `ptc.ga` variant of the global purge instruction behaves just like the `ptc.g` variant, but it also removes any ALAT entries which fall into the address range specified by the global shootdown from all remote processors' ALATs. The `ptc.ga` variant is intended to be used whenever a translation is remapped to a different physical address to ensure that any stale ALAT entries are invalidated. Note that the `ptc.ga` does not affect the issuing processor's ALAT; software must perform a local ALAT purge via the invala instruction on the processor issuing the `ptc.ga` to ensure the local ALAT is coherent.

Note that processors based on the Itanium architecture may support one or more implementation-dependent purge sizes; some implementations may include a region-wide purge. The PAL_VM_PAGE_SIZE firmware call returns the supported page sizes for purges for a particular processor implementation. Refer to Section 11.9.1, "PAL Procedure Summary" for details. When software wishes to purge an address range that is much larger than the largest supported purge size from all TCs in the coherence domain, performance may be enhanced by issuing inter-processor interrupts to all processors and using the `ptc.e` loop described in Section 5.2.2.2.2 on each processor, instead of issuing many `ptc.g` instructions from one processor.

`ptc.g` instructions do not apply to processors outside the coherence domain of the processor issuing the `ptc.g` instruction. Systems with multiple coherence domains must use a platform-specific method for maintaining TLB coherence across coherence domains.

5.3 Virtual Hash Page Table

The Itanium architecture defines a data structure that allows for the insertion of TLB entries by a hardware mechanism. The data structure is called the "virtual hash page table" (VHPT) and the hardware mechanism is called the VHPT "walker".

Unlike the IA-32 page tables, the Itanium VHPT itself is virtually mapped, i.e. VHPT walker references can take TLB faults themselves. Virtual mapping of the page tables is needed because the page tables for 2^{64} address space are quite large and typically do not fit into physical memory.

The Itanium architecture prescribes the format of a leaf-node page table entry (PTE) seen by the VHPT walker, but does not impose an OS page table data structure itself. As summarized in Table 5-1, the architecture support two different VHPT formats:

- **Short** format uses 8-byte PTEs, and is a linear page table. The short format VHPT cannot use protection keys (there are not enough PTE bits for that). Short format is a per-region linear page table, i.e. the PTEs and hash function are independent of the RID. The short format prefers use of a self-mapped page table. The short format VHPT is an efficient representation for address spaces that contain only a few large clusters of pages, like the text, data, and stack segments of applications running on a MAS operating system.
- **Long** format uses 32-byte PTEs, and is a hashed page table. The hash function embedded in hardware. The long format supports protection keys and the use of multiple page sizes in a region. The long format hash and tag functions incorporate the RID, and allows multiple address space translations to be present in the same VHPT. The long format is expected to be used either as a cache of the real OS page tables, or as a primary page table with collision chains. The long format VHPT is a much better representation for address spaces that are sparsely populated, since the short format VHPT has a linear layout and would consume a

large amount of memory. Single address space operating systems may prefer the long format VHPT for this reason.

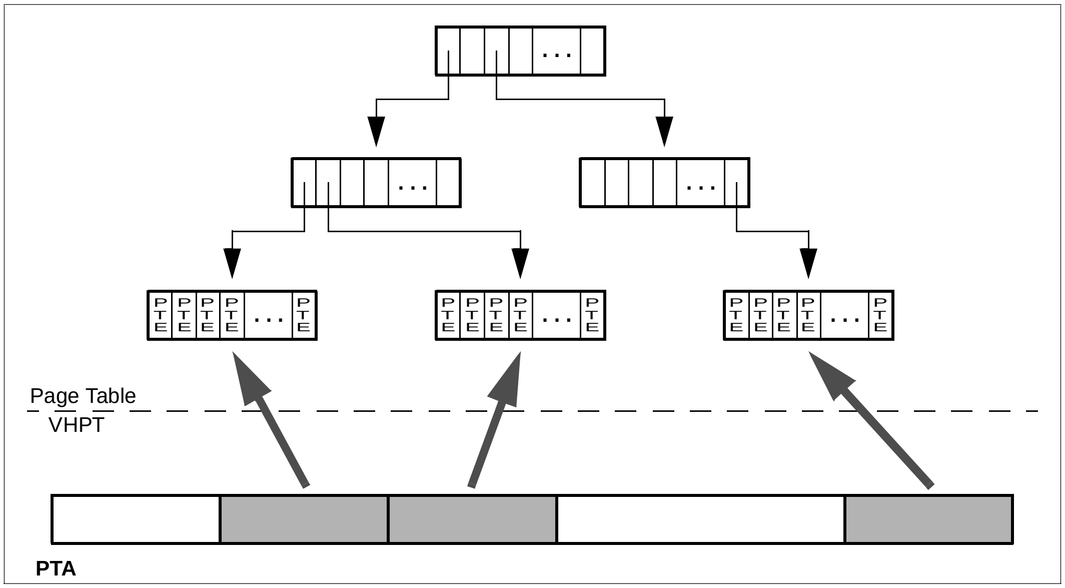
Table 5-1. Comparison of VHPT Formats

Attribute	Short Format	Long Format
Entry Size	8 Byte	32 Byte
Lookup	Linear	Hashed
Protection Keys	No	Yes
Page Size	per region	per entry

5.3.1 Short Format

The short format VHPT is a per-region linear table that contains translation entries for every page in the region's virtual address space. This makes the VHPT very large, but since the VHPT itself lives in virtual address space only those parts of the VHPT that actually contain valid translation entries have to be present in physical memory. If the operating system's page table is a hierarchical data structure and the last level of the hierarchy is a linear list of translations, the VHPT can be mapped directly onto the page table as shown in Figure 5-1.

Figure 5-1. Self-mapped Page Table



If the VHPT walker tries to access a location in the VHPT for which no translation is present in the TLB, a VHPT Translation fault is raised. The original address for which the VHPT walker was trying to find an entry in the VHPT is supplied to the fault handler in the IFA register. The fault handler can use this address to traverse the page table and insert a translation into the TLB that maps the address the VHPT walker tried to access (in IHA) to the page that contains the corresponding leaf page table.

5.3.2 Long Format

The long format VHPT is organized as a hash table which contains a subset of all translation entries. The long format VHPT entries contain a 8-byte field that is ignored by the VHPT walker and can be used by the operating system to link VHPT entries to software-walkable hash collision chains if it uses the VHPT as its primary page table. The size of the long format VHPT is usually kept small enough to keep a mapping for it in one of the translation registers (TRs), so it is not necessary to handle VHPT translation faults.

The long format hash algorithm is based on the per-region preferred page size, but a translation for a larger page can still be entered into the VHPT by subdividing the large page into multiple smaller pages with the preferred page size and placing an entry for the large page at all VHPT locations that correspond to the smaller pages.

5.3.3 VHPT Updates

The VHPT walker uses unordered load semantics to access the in-memory VHPT. Visibility of VHPT updates to a VHPT walker on another processor follows the rules outlined in Section 4.1.7, "VHPT Environment". Since a global TLB purge has release semantics, prior modifications to the VHPT will be visible to operations that occur after the TLB purge operation.

Atomic updates to short format VHPT entries can easily be done through 8-byte stores. For atomic updates of long format VHPT entries, the "ti" flag in bit 63 of the tag field can be utilized as follows:

- Set the "ti" bit to 1.
- Issue a memory fence.
- Update the entry.
- Clear the "ti" bit through a store with release semantics.

5.4 TLB Miss Handlers

The Itanium architecture enables lightweight TLB fault handlers by providing individual entry points for different excepting conditions and by pre-setting the translation insertion registers for the various types of TLB faults. The following subsections list the typical steps for resolving each kind of fault.

5.4.1 Data/Instruction TLB Miss Vectors

These faults occur when the data or instruction TLB required for a data access or instruction fetch is not found in the processor TLBs, the VHPT walker is enabled, and:

- Either the VHPT walker aborted the walk (for any reason and at any time), or
- The VHPT walker found the translation but the insert failed (due to tag mismatch in the long format or badly formed PTE), or
- The walker is not implemented on this processor.

There is a separate vector for each fault type (data and instruction).

Since the VHPT walker may abort a walk at any time and raise these faults, software must always be able to handle all TLB faults, even when the VHPT walker is enabled. Upon entry to these fault handlers, the IHA, ITIR, and IFA control registers are initialized by the hardware as follows:

- IHA – contains the virtual address of the hashed page table address corresponding to the reference which raised the fault.
- ITIR – contains the default translation information for the reference which raised the fault (i.e. for the virtual address contained in IFA). The access key field is set to the region ID from the RR corresponding to the faulting address. The page size field is set to the preferred page size (RR.ps) from the RR corresponding to the faulting address.
- IFA – the virtual address of the bundle (for instruction faults) or data reference (for data faults) which missed the TLB.

The fault handler for a short format VHPT performs the following steps, at a minimum, to handle the fault:

1. Move IHA into a general register, chosen by convention to match the register expected by the nested TLB fault handler.
2. Perform an 8-byte load into another general register from the address contained in this general register to grab the VHPT entry. Note that the format of these first 8 bytes is identical to the format required for TLB insertion. If the VHPT is not mapped by a TR, software must be prepared to handle a nested TLB fault when performing this load.
3. Using the general register from step 2 that holds the contents of the VHPT entry, perform a TC insert (`itc.i` for instruction faults, `itc.d` for data faults).
4. In an MP environment, reload the VHPT entry from step 2 into a third general register and compare the value to the one loaded in step 2. If the values are not the same, then the VHPT has been modified by another processor between steps 2 and 3, and the entry will have to be re-inserted. In this case, purge the entry just inserted using a `ptc.l` instruction. The fault will re-occur after the `rfi` in step 5 (unless the VHPT walker succeeds on the next TLB miss) and the fault handler will re-attempt the insertion. (Uniprocessor environments may skip this step.)
5. `rfi`.

For a long format VHPT, additional steps are required to load bytes 16-23 of the VHPT entry and check for the correct tag (the correct tag for the reference can be generated using the ttag instruction). If the tags do not match, this indicates a VHPT collision, and the handler must proceed to walk the operating system’s collision chain manually to find the correct entry. The handler may then choose to swap places between the correct entry and the VHPT entry. Note that the pointers for a collision chain can be stored in bytes 24-31 of the VHPT entry format since these bytes are ignored by the VHPT walker.

If the default page size and key are not sufficient, the handler must also perform additional steps to load the correct page size and key into the ITIR register before performing the TC insert in step 3 of the sequence shown above.

5.4.2 VHPT Translation Vector

Processors based on the Itanium architecture does not perform recursive TLB hardware page walks. Since the VHPT is itself a virtually addressed structure, each reference performed by the walker itself goes through the TLBs and may miss. These faults are raised when the VHPT walker is enabled, but the walker misses the TLBs when attempting to service a TLB miss caused by the program.

There is a separate vector for each fault type (data and instruction).

Upon entry to this fault handler, the IHA, IFA, and ITIR control registers are initialized by the hardware as follows:

- IHA – contains the virtual address of the hashed page table address corresponding to the reference which raised the fault.
- ITIR – contains the default translation information for the VHPT address which missed the TLBs (i.e. for the virtual address contained in IHA). The access key field is set to the region ID from the RR corresponding to the VHPT address. The page size field is set to the preferred page size (RR.ps) from the RR corresponding to the VHPT address.
- IFA – contains the original faulting address that the VHPT walker was attempting to resolve.

The fault handler for a short format VHPT performs the following steps, at a minimum, to handle the fault:

1. Move the IHA register into a general register.
2. Perform a thash instruction using the general register from step 1 This will produce, in the target register, the VHPT address of the VHPT entry that maps the VHPT entry corresponding to the original faulting address (i.e. the address in IFA).
3. Using the target general register of the thash from step 2 as the load address, perform an 8-byte load from the VHPT. Note that the format of these first 8 bytes is identical to the format required for TLB insertion. Software must be prepared to take a nested TLB fault if this load misses the TLBs.
4. Move the IHA value from the general register written in step 1 into the IFA register.
5. Using the general register from step 3 that holds the contents of the VHPT entry, perform a data TC insert using the `itc.d` instruction. (VHPT references always go through the data TLBs.)
6. In an MP environment, reload the VHPT entry from step 3 into a different general register and compare the value to the one loaded in step 3. If the values are not the same, then the VHPT has been modified by another processor between steps 3 and 4, and the entry will have to be re-inserted. In this case, purge the entry just inserted using a `ptc.l` instruction. The fault will re-occur after the `rfi` in step 7 (unless the VHPT walker succeeds on the next TLB miss) and the fault handler will re-attempt the insertion. (Uniprocessor environments may skip this step.)
7. `rfi`.

For a long format VHPT, additional steps are required to load bytes 16-23 of the VHPT entry and check for the correct tag; see Section 5.4.1 for more details.

A separate structure other than the VHPT may be used to back VHPT translations, in which case the handler would not use the thash instruction to generate the address of the translation mapping the VHPT entry corresponding to the original faulting address. Instead, the handler would use the operating system's own mechanism for finding VHPT back-mappings. Other schemes for handling VHPT misses are also possible, but are beyond the scope of this document.

5.4.3 Alternate Data/Instruction TLB Miss Vectors

These faults are raised when an instruction or data reference misses the processor's TLBs and the VHPT walker is not enabled for the faulting address, i.e. TLB misses are handled entirely in software. Operating systems which do not wish to use the VHPT walker can disable the walker and use these fault vectors for software TLB fill handlers. The OS may also choose to enable the walker on a per-region basis and use these vectors to handle misses in regions where the walker is disabled.

Upon entry to these fault handlers, the IFA and ITIR registers are initialized by the hardware as follows:

- ITIR – contains the default translation information for the reference which raised the fault (i.e. for the virtual address contained in IFA). The access key field is set to the region ID from the RR corresponding to the faulting address. The page size field is set to the preferred page size (RR.ps) from the RR corresponding to the faulting address.
- IFA – the virtual address of the bundle (for instruction faults) or data reference (for data faults) which missed the TLB.

The OS needs to lookup the PTE for the faulting address in the OS page table, convert it to the architected insertion format (see Section 4.1.1.5, "Translation Insertion Format"), and insert it into the TLB. The mechanism used to handle these faults is OS-specific and is beyond the scope of this document.

5.4.4 Data Nested TLB Vector

To enable efficient handling of software TLB fills, the Itanium architecture provides a dedicated Data Nested TLB fault vector. The Data Nested TLB fault handler is intended to be used by the Data TLB fault handler, which allows the OS to page the page tables themselves. When PSR.ic is 0, any data reference that misses the TLB and would normally raise a Data TLB Miss fault (e.g. a load performed by the Data TLB fault handler to the page tables) will vector to the Data Nested TLB fault handler instead. Because IFA is not updated when PSR.ic is 0, the Data Nested TLB fault handler must get the faulting address from the general register used as the load address in the Data TLB fault handler[5]. Unlike other nested interruptions, the hardware does *not* update ISR when a Data Nested TLB fault is delivered.

The processor will not deliver a Data Nested TLB fault when PSR.ic is in-flight; Data Nested TLB faults are only delivered when PSR.ic is 0. If PSR.ic is in-flight, any data references which miss the TLB and trigger a fault will raise a Data TLB fault, and the processor will set ISR.ni to 1.

[5] This requires a register usage convention between all TLB miss handlers and the Data Nested TLB miss handler.

5.4.5 Dirty Bit Vector

The operating system is expected to lookup the PTE for the faulting address in the OS page table and load the PTE into a general register r_x. It can then set the “dirty” bit in r_x and write the updated PTE back to the page table. To continue execution, the OS must insert the updated PTE into the data TLB or update the PTE memory image and let the VHPT walker perform the insertion.

5.4.6 Data/Instruction Access Bit Vector

The operating system is expected to lookup the PTE for the faulting address in the OS page table and load the PTE into a general register r_x. It can then set the “access” bit in r_x and to continue execution, the OS must either:

- Write the updated PTE back to the page table, and have the VHPT walker pick it up, or
- Insert the updated PTE into the TLB using `itc.i` r_x for instruction pages, and `itc.d` r_x for data pages, or
- Step over the instruction/data access bit fault by setting the IPSR.ia or IPSR.da bits prior to performing an `rfi`.

5.4.7 Page Not Present Vector

Forward the fault to the operating system’s virtual memory subsystem.

5.4.8 Data/Instruction Access Rights Vector

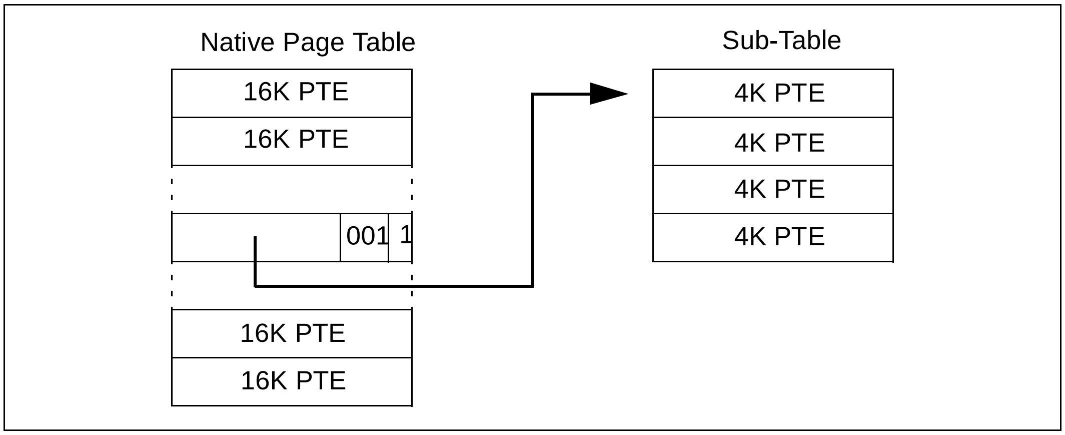
Forward the fault to the operating system’s virtual memory subsystem.

5.5 Subpaging

The native page size an Itanium-based operating system will choose for its page tables is likely be larger than the architectural minimum page size of 4 KB. Some legacy IA-32 applications, however, expect a page protection granularity of 4 KB. The following technique allows support for these applications with minimal impact on the native, larger page size paging mechanism.

A special type of entry is used in the native page table to mark pages that are subdivided into smaller 4 KByte units. The entry must have its memory attribute field set to the architecturally “software reserved” encoding (binary 001), and it carries a pointer to an array of 4 KB subentries in its most significant 59 bits. An example using a native page size of 16 KB is shown in Figure 5-2. The use of the “software reserved” memory attribute prevents the VHPT walker from attempting to insert the entry into the TLB.

Figure 5-2. Subpaging



When one of the subdivided pages is referenced and does not have a translation in the TLB, a TLB miss will occur. The handler for this fault can then use the faulting address to calculate the appropriate offset into the sub-table and insert the corresponding 4KByte PTE into the TLB.

Some care is required to ensure forward progress for IA-32 instructions. Each IA-32 instruction can reference up to 8 distinct memory pages during its execution (see also Section 10.6.3, "IA-32 TLB Forward Progress Requirements"). This means that the fault handler not only has to insert the PTE for the current fault into the TLB, but also the PTEs for up to seven faults that occurred before, if these faults originate from the same IA-32 instruction. This can be accomplished by maintaining a buffer for the most recent faulting IIP and for the parameters of up to 7 TLB insertions. If a TLB fault occurs while executing in IA-32 mode and the IIP matches the most recent IIP, all TLB insertions in the buffer have to be repeated and the parameters for the new TLB fault must be added to the buffer. Otherwise, the buffer can be cleared out and the most recent IIP can be updated. The buffer also has to be cleared out when a TLB purge occurs.

Runtime Support for Control and Data Speculation 6

An Itanium-based operating system needs to handle exceptions generated by control speculative loads (`ld.s` or `ld.sa`), data speculative loads (`ld.a`) and architectural loads (`ld`) in different ways.

Software does not have to worry about control or data speculative loads potentially hitting uncacheable memory with side-effects, since `ld.s`, `ld.sa`, and `ld.a` instructions to non-speculative memory are always deferred by the processor for details refer to Section 4.4.6, "Speculation Attributes". As a result, compilers can freely use control and data speculation to all program variables.

Control speculative loads require special exception handling and the Itanium architecture provides a variety of deferral mechanisms for handling of control speculative exception handling. This is discussed in Section 6.1.

The Itanium architecture supports different control speculation recovery models. These are discussed in Section 6.2.

Handling of exceptions caused by architectural and data speculative loads is the same, except for emulation of unaligned data speculative references, which require special unaligned emulation handling. This is discussed in Section 6.3.1.

6.1 Exception Deferral of Control Speculative Loads

Exceptions that occur on control speculative loads (`ld.s` or `ld.sa`) can be handled by the operating system in different ways. The operating system can configure a processor based on the Itanium architecture in three ways:

- Hardware-Only Deferral: automatic hardware deferral of all control speculative exceptions. In this case, the processor hardware will always defer excepting control speculative loads without invoking the operating system.
- Combined Hardware/Software Deferral: automatic deferral of some control speculative exceptions, but deliver others to software. In this case, some exceptions will result in hardware deferral as described above, other exceptions will be reported to the operating system. The operating system fault handlers can identify that an exception has been caused by a control speculative load (ISR.sp will be 1). Furthermore, OS handlers can software-defer an exception on a control speculative load by setting IPSR.ed to 1 prior to `rfi`-ing back to the `ld.s` or `ld.sa`. This allows an operating system to service "cheap" non-fatal exceptions (e.g. simple TLB misses), while software-deferring both "expensive" non-fatal (e.g. page faults) as well as fatal exceptions (e.g. non-recovery protection violation).
- Software-Only Deferral: processor is configured to deliver all control speculative exceptions to software. In this case, operating system software handles all non-fatal control speculative exceptions, and software-defers all fatal control speculative exceptions.

Details on these three models are discussed in the next three sections as well as in Section 5.5.5, "Deferral of Speculative Load Faults".

6.1.1 Hardware-only Deferral

Hardware only deferral is configured by setting all speculation deferral bits in the DCR register (dd, da, dr, dx, dk, dp and dm) to 1. All excepting control speculative loads are automatically deferred by the processor. As a result, all excepting control speculative loads that hit non-fatal exceptions, e.g. a TLB miss or a page fault, will be deferred by the processor hardware, and will cause speculation recovery code to be invoked. This can cause speculation recovery code to be invoked more often than strictly necessary.

6.1.2 Combined Hardware/Software Deferral

Setting of a DCR deferral bit to 1 results in hardware deferral by the processor, whereas clearing of a deferral bit causes exceptions to be delivered to software. The operating system may want to configure the processor to deliver control speculative exceptions to its handlers for certain non-fatal faults such as TLB misses or protection key misses. Early handling of these exceptions avoids unnecessary invocation of speculation recovery code, and the associated performance penalty. This is especially useful for exceptions handlers whose overhead is small. Note that handlers will also be invoked for excepting control speculative loads that have been hoisted from not taken paths, and therefore are not needed. As a result, software handling of control speculative exceptions is recommended only for statistically infrequent light weight fault handlers such as TLB miss or protection key miss handlers. If, while handling the exception, the operating system determines that this instance of the exception may require too much effort, e.g. a TLB miss turns out to be a page fault, the handler still has the choice of software-deferring the exception.

6.1.3 Software-only Deferral

Software only deferral is configured by clearing all speculation deferral bits in the DCR register (dd, da, dr, dx, dk, dp and dm) to 0. Control speculative loads that hit any Debug, Access Bit, Access Rights, Key Permissions, Key Miss, or Not Present fault, or that suffer a TLB miss or a VHPT Translation fault will be delivered to software.

6.2 Speculation Recovery Code Requirements

As described by Table 6-1, code generators for the Itanium architecture are not always required to generate speculation recovery code for all forms of speculation. Compilers and operating systems can collaborate to provide two models for handling of recovery from failed control speculation:

- ITLB.ed=1 (application with recovery code - the default): The compiler generates appropriate recovery code for all `ld.s` instructions, as well as for `ld.sa` and `ld.a` instructions that have speculatively executed uses. Speculation failure of `ld.sa` and `ld.a` instructions that have no speculatively executed uses can be recovered by a `ld.c` instruction, and hence do not require recovery code. The operating system may defer non-fatal exceptions.

- ITLB.ed=0 (no control speculative recovery code): The compiler generates recovery code only for ld.sa and ld.a instructions that have speculatively executed uses. Speculation failure of `ld.sa` and `ld.a` instructions that have no speculatively executed uses can be recovered by a `ld.c` instruction, and hence do not require recovery code. Speculation failure of `ld.s` instructions does not require recovery code, because, in this model, the operating system must guarantee that only fatal exceptions will be deferred. This requires software-only deferral of all potential non-fatal exceptions. The motivation for this model is that the absence of `chk.s` instructions and their associated recovery code may make for shorter and more compact in-line code, especially in loops with tight instruction schedules.

Table 6-1. Speculation Recovery Code Requirements

Usage Model	OS May Defer Non-fatal Exceptions on Control Speculative Loads (ITLB.ed=1)	OS Must Not Defer Non-fatal Exceptions on Control Speculative Loads (ITLB.ed=0)
No Speculative Load Uses		
`ld.s`	Recovery code required; Invoked by `chk.s` or non-speculative use of speculative value recovers from failed control speculation.	No recovery code required; OS handles all non-fatal exceptions speculatively.
`ld.sa, ld.a`	No recovery code required; `ld.c` recovers from failed data speculation.	
With Speculative Load Uses		
`ld.s`	Recovery code required; invoked by `chk.s` or non-speculative use of speculative value recovers from failed control speculation.	No recovery code required; OS handles all non-fatal exceptions speculatively.
`ld.sa, ld.a`	Recovery code required; `chk.a` recovers from failed data speculation.	

Presence or lack of control speculation recovery code is communicated from the compiler and the run-time system to the operating system by marking the code page's page table entry ed-bit appropriately (this bit is referred to as ITLB.ed). When ITLB.ed is 1, the operating system will expect recovery code to be present; when ITLB.ed is 0 no recovery code is expected. When a control speculative load takes an exception, the code page's ITLB.ed bit is copied into ISR.ed and is made available to the operating system exception handler. Furthermore, a set ISR.sp bit indicates that an exception was caused by a control speculative load.

6.3 Speculation Related Exception Handlers

6.3.1 Unaligned Handler

Misaligned control and data speculative loads, as well as architectural loads, are not required to be handled by the processor. As a result, the operating system's unaligned reference handler has to be prepared to emulate such misaligned memory references, especially in cases where the application has not provided any recovery code (see Section 6.2 for details). Furthermore, misaligned data speculative loads (`ld.sa` or `ld.a`) must be forced failed by the unaligned emulation handler, because the ALAT cannot track all sizes of misalignment for store conflict detection.

The following pseudo code outlines the basic steps for an unaligned reference handler:

1. Ensure that only ISR.r is 1, and that ISR.w, ISR.x, and ISR.na are 0.
2. Inspect the ISR.sp and ISR.ed. If both are 1, then defer this control speculative load by setting IPSR.ed and `rfi`-ing.
3. Crack the instruction opcode to determine:
   a. Size of the load: 1, 2, 4, 8, 10 bytes
   b. Type of the load: `ld.sa`, `ld.s`, `ld.a`, `ld.c.clr`, `ld.c.nc` or `ld`
   c. Target, source and post-increment registers of the load
4. If this is a data speculative load (`ld.sa`, or `ld.a`), invalidate the target register's ALAT entry using an `invala.e` instruction, and `rfi`.
5. If this is a `ld.c.clr` instruction invalidate the target register's ALAT entry using an `invala.e` instruction.
6. Emulate the memory read of the load instruction by updating the target register as follows:
   a. Validate that emulated code has the access rights to the target memory location at the privilege level that it was running prior to taking the alignment fault. The `probe` instruction can be used on the first and the last byte of the unaligned memory reference. If both probes succeed the memory reference may proceed.
   b. Using architectural `ld` instructions if the emulated operation is a `ld` or a `ld.c` (either clear or no clear flavor).
   c. Using `ld.s` instructions if the emulated operation is a `ld.s`. The result in the target register may end up with its NaT bit or NaTVal set, if one of the parts of emulation causes an exception. If ITLB.ed is 0 (no control speculation recovery code), then the misaligned `ld.s` may only be deferred if a fatal exception occurred on either half or the `ld.s` emulation.
7. If this is a post-increment load, compute the new value for the source register.

Instruction Emulation and Other Fault Handlers 7

This chapter introduces several common emulation handlers that an Itanium-based operating system must support. A general overview is given for:

- Unaligned Reference Handler – emulation of misaligned memory references that the processor hardware cannot handle, or has been configured to fault on.
- Unsupported Data Reference Handler – emulation of memory operations that the processor hardware does not support. Examples are semaphore, `ldfe` or `stfe` operations to uncacheable memory.
- Illegal Dependency Fault Handler – this is a fatal condition that operating system needs to provide error logging functionality for.
- Long Branch Handler – the Itanium processor does not implement the long branch instruction. When encountered on the Itanium processor, long branches must be emulated by the operating system.

Floating-point software assist emulation handlers are not discussed here, but are presented in Chapter 8, “Floating-point System Software”. Additionally, Section 5.5.1, “Efficient Interruption Handling” discusses more details about emulation code in the Itanium architecture.

7.1 Unaligned Reference Handler

Misaligned memory references that are not supported by the processor cause Unaligned Reference Faults. This behavior is implementation-specific but typically occurs in cases where the access crosses a cache line or page boundary. In cases where the operating system chooses to emulate misaligned operations, some special cases need to be considered:

- Emulation of control and data speculative loads as well as advanced check and “regular” loads requires special attention. For details consult Section 6.3.1, “Unaligned Handler” on page 2:445.
- Emulation of unaligned semaphores, especially when interacting with IA-32 code require special attention. For details consult Section 2.1.3.2, “Behavior of Uncacheable and Misaligned Semaphores” on page 2:377.

IA-32 programs do not use the Itanium-based handler to support unaligned references. The hardware that supports IA-32 execution provides the appropriate behavior if alignment checking is disabled through EFLAGS.ac. If an unaligned reference occurs in IA-32 code when EFLAGS.ac is set to enable alignment checking, alignment faults are delivered to a different vector from the unaligned reference handler. Specifically they are delivered to the IA_32_Exception(AlignmentCheck) vector; see Section 9, “IA-32 Interruption Vector Descriptions” for details.

7.2 Unsupported Data Reference Handler

Processors based on the Itanium architecture do not support all types of memory references to all memory attributes. In particular:

- Semaphore operations to uncacheable memory are not supported. For details consult Section 2.1.3.2, "Behavior of Uncacheable and Misaligned Semaphores" on page 2:377.
- A 10-byte memory access, e.g. `ldfe` or `stfe`, to uncacheable memory are not supported by all implementations.

The handler for 10-byte memory accesses must go through the following steps to emulate the `ldfe` or `stfe` instructions:

- Determine that the opcode at the faulting address is an `ldfe` or `stfe`. On control-speculative flavors of these instructions (`ldfe.s` or `ldfe.sa`) processor hardware always defers the unsupported data reference fault. In other words, software does not have to emulate control-speculative fault deferral.
- If the instruction is an advanced load `ldfe.a` then the emulation handler should invalidate the ALAT entry of the appropriate floating-point target register using the `invala.e` instruction. Furthermore, a zero should be returned in the floating-point target register.
- If the instruction is a regular `ldfe` or `stfe`, then software must emulate the load or store behavior of the instruction taking the appropriate faults if necessary.
- If the instruction is the base register update form, update the appropriate base register.

A number of these steps may require the use of self-modifying code to patch instructions with the appropriate operands (for example, the target register of the `inval.e` must be patched to the destination register of the `ldfe` or `stfe`). See Section 2.5, "Updating Code Images" on page 2:399 for more information.

7.3 Illegal Dependency Fault

The Itanium instruction sequencing rules specify that, generally speaking, instructions within an instruction group are free of dependencies as described in Section 3.4, "Instruction Sequencing Considerations" in Volume 1. A dependency violation occurs anytime a program violates read-after-write (RAW), write-after-write (WAW) or write-after-read (WAR) resource dependency rules within an instruction group.

As Section 3.4.4, "Processor Behavior on Dependency Violations" in Volume 1 describes, an implementation may provide hardware to detect and report dependency violations. It is important to note that the presence and capabilities of such hardware is implementation-specific. A processor based on the Itanium architecture reports dependency violations through the General Exception Vector with an ISR.code of 8.

It is recommended that operating systems log the dependency violation and then terminate the offending application, as hardware behavior is undefined when a dependency violation occurs.

7.4 Long Branch

The Itanium architecture supports “long” branches with a 64-bit offset. This provides IP-relative conditional- and call-type branches that can reach any address in a 64-bit address space. These instructions use the MLX template, and similar to the move long instruction (`movl`), they encode their immediate in the L and the X slot of the bundle.

The Intel Itanium processor does not support the long branch instruction, `brl`, and requires the operating system to emulate its behavior. When an Itanium processor encounters a `brl` instruction, it vectors to the Illegal Operation Fault handler, regardless of the branches’ qualifying predicate. This handler is expected to emulate the long branch instruction in software. A general outline of the long branch emulation handler is as follows:

- The emulation handler reads the IIP, IPSR, and predicates at the time of the fault.
- If the fault occurred in IA-32 code or if the fault did not occur in slot 2 of a bundle (IPSR.ri is not 2), the handler passes the fault to regular illegal operation fault handler.
- Two floating-point registers are spilled into the integer register file to get ready to load the bundle.
- The emulation handler speculatively loads the 128-bit bundle at the faulting IP using the integer form of the floating-point load pair instruction. This instruction is chosen because it operates atomically (see Section 4.5, “Memory Datum Alignment and Atomicity”). Using two 64-bit integer loads would require the handler to ensure that another agent does not update the bundle between the two reads.
- If the speculation fails, the recovery code re-issues the load. Before re-issuing an architectural load, the processor must first re-enable PSR.ic to be able to handle potential TLB misses when reading the opcode from memory. In other words, this becomes a heavyweight handler. For details see Section 3.4.2, “Heavyweight Interruptions” on page 2:411. Once the opcode has been read from memory successfully flow of the emulation continues at the next step.
- The 128-bit bundle is moved from the FP register file into two integer registers and the FP registers are restored to their contents at the time of the fault.
- The handler extracts the fields necessary to decode the instruction (specifically, the qp, template, major opcode, and btype or b_1 fields of slot 2). It also determines the value of the qualifying predicate of the instruction in slot 2 from the contents of the predicate register at the time of the fault. Itanium instruction are always stored in memory in little-endian memory format. When extracting bit fields from the loaded opcode current processor endianness (PSR.be) must be taken into account.
- The emulation handler passes the fault off to the regular illegal operation fault handler if the bundle is not an MLX or if the faulting instruction is not a `brl.cond` or `brl.call`.
- If the faulting instruction is a not-taken `brl.cond` or `brl.call`, the code prepares to change the IIP to the address of the sequential successor of the faulting branch (i.e. IIP + 16) and jumps ahead to the trap detection code mentioned below.
- If the faulting instruction is a taken `brl.call`, the handler emulates the appropriate behavior of the call. The code uses a `br.call` to move the appropriate values into CFM and AR[PFS]. There are several details, however. First, the branch register update from the call must be backed out (as it is not the correct update for the `brl.call`). Second, AR[PFS].ppl must be set based on the cpl at the time of the fault (which is given by IPSR.cpl). Finally, the code must update the branch register specified in the `brl.call` instruction with the IP of the successor of the `brl.call` (predication helps here as the Itanium instruction set does not provide an indirect move to branch register instruction).

- The handler forms the 60-bit immediate IP-offset for the `brl` target from the `i` and `imm20` fields from the X syllable of the bundle (the `brl` instruction) and the `imm39` field from the L syllable of the bundle.
- The handler checks to see if there are any traps to be taken. Specifically, it verifies that the next IP is at an implemented address (the specific test depends on whether the processor was in virtual or physical mode at the time of the fault as IPSR.it indicates), that taken branch traps are not enabled if the branch is taken, and that single stepping is not enabled.
- If a trap condition is detected, the ISR.code and ISR.vector fields are set up as appropriate and the handler jumps to the appropriate operating system entry point after restoring the predicates at the time of the fault and setting the IIP to the appropriate address.
- If no trap occurs, the handler restores the predicates and returns to the faulting code at the appropriate IP.

A processor based on the Itanium architecture typically does not fault on instructions with false qualifying predicates. However, an implementation may take an Illegal Operation Fault on an MLX instruction with a false predicate; the Itanium processor is such an implementation. This implies that the `brl` emulation handler must also provide the means to skip the faulting instruction when its qualifying predicate is false.

Floating-point System Software 8

This chapter details the way floating-point exceptions are handled in the Itanium architecture and how the architecture can be used to implement the ANSI/IEEE Std. 754-1985 for Binary Floating-point Arithmetic (IEEE-754). It is useful in creating and maintaining floating-point exception handling software by operating system writers.

8.1 Floating-point Exceptions in the Intel® Itanium™ Architecture

Floating-point exception handling in the Itanium architecture has two major responsibilities. The first responsibility is to assist a hardware implementation to conform to the Itanium floating-point architecture specification. The Floating-point Software Assistance (FP SWA) Exception handler supports this conformance and is included as a driver in the Extensible Firmware Interface (EFI). The second responsibility is to provide conformance to the IEEE-754 standard. The IEEE Floating-point Exception Filter (IEEE Filter) supports providing this conformance.

When a floating-point exception occurs, a minimal amount of processor state information is saved in interruption control registers. Additional information is contained in the Floating-point Status Register (FPSR), i.e. application register (AR40). This register contains the IEEE exception enable controls, the IEEE rounding controls, the IEEE status flags, and information to determine the dynamic precision and range of the result to be produced.

When a floating-point exception occurs, execution is transferred to the appropriate interruption vector, either the Floating-point Fault Vector (at vector address 0x5c00) or the Floating-point Trap Vector (at vector address 0x5d00.) There the operating system may handle the exception or save additional processor information and arrange for handling of the exception elsewhere in the operating system. Floating-point exception faults must be handled differently than other faults. Correcting the condition that caused the fault (e.g. a page not present is brought into memory) and re-executing the instruction is how most other faults are handled. For floating-point faults, software is required to emulate the operation and continue execution at the next instruction as is normally done for traps. Part of this emulation needs to include a check for any lower priority traps that would have been raised if the instruction hadn’t faulted, e.g. a single-step trap.

8.1.1 The Software Assistance Exceptions (Faults and Traps)

There are three categories of Software Assistance (SWA) exceptions that must handled by the operating system. The first two categories, SWA Faults and SWA Traps, are implementation dependent and could be generated by any Itanium floating-point arithmetic instruction that contains a status field specifier in the instruction's encoding. An implementation may choose to raise a SWA Fault as needed. The SWA Trap can only be raised under special circumstances. The third category, architecturally mandated SWA Faults, is limited to the scalar reciprocal and scalar reciprocal square-root approximation instructions and is not implementation dependent. It is required for the correctness of the divide and square root algorithms.

8.1.1.1 SWA Faults

The Itanium architecture allows an implementation to raise SWA faults as required. Therefore an implementation-independent operating system must be able to emulate the architectural behavior of all FP instructions that can raise a floating-point exception. However, hardware implementations will limit the cases that raise SWA Faults for performance reasons. The most likely cases would be for the consumption of denormalized or unnormalized operands and production of denormalized results.

The general flow of the SWA Fault handler is as follows:

1. From the interruption instruction bundle pointer (IIP) and faulting instruction index (IPSR.ri), determine the FP instruction that faulted.
2. From the instruction, decode the opcode, static precision, status field and input/output register specifiers.
3. Read the data from the input registers.
4. From the opcode and the FPSR’s status field, decode the result range and precision.
5. From the ISR.code, determine that a SWA Fault has occurred, if not go to the last step.
6. From the FPSR, determine if the trap disabled or trap enabled result is wanted.
7. Emulate the Itanium instruction to produce the Itanium architecture specified result.
8. Place the result(s) in the correct FR and/or PR registers, if required.
9. Update the flags in the appropriate status field of the FPSR, if required.
10. Update the ISR.code if required. (This is required if the SWA fault has been translated into an IEEE fault or trap.)
11. Check to see if an IEEE fault or trap needs to be raised. If so, then queue it to the IEEE Filter, otherwise continue checking for lower priority traps that may need to be raised and if required invoke their handler. When finished, continue execution at the next instruction.

8.1.1.2 SWA Traps

SWA traps are allowed in the Itanium architecture as an optimization for cases when the hardware implementation has produced the result of the first (exponent unbounded) IEEE rounding[1] and can't continue with the second (exponent bounded) IEEE rounding to produce the final result. One option for the implementation would be to throw away the first IEEE rounding result and raise the SWA Fault. The SWA Fault handler would then have to redo the computation of the first IEEE rounding. A potentially more efficient option would be for the implementation to return the first IEEE rounding result and raise a SWA trap. Returning the first IEEE rounded result is the same as what is done when the IEEE Overflow or Underflow exceptions are enabled. However, hardware implementations will limit the cases that raise SWA Traps for performance reasons. The most likely case would be for the production of denormalized results.

For tiny[2] results, the SWA Trap handler has the simpler task of taking the intermediate result of the first IEEE rounding, the ISR.fpa and ISR.i status bits and producing the correctly rounded and signed minimum normal, denormal or zero. For huge[3] results, the SWA Trap handler has the even

[1] ANSI/IEEE Std 754-1985 sections 7.3 Overflow and 7.4 Underflow.

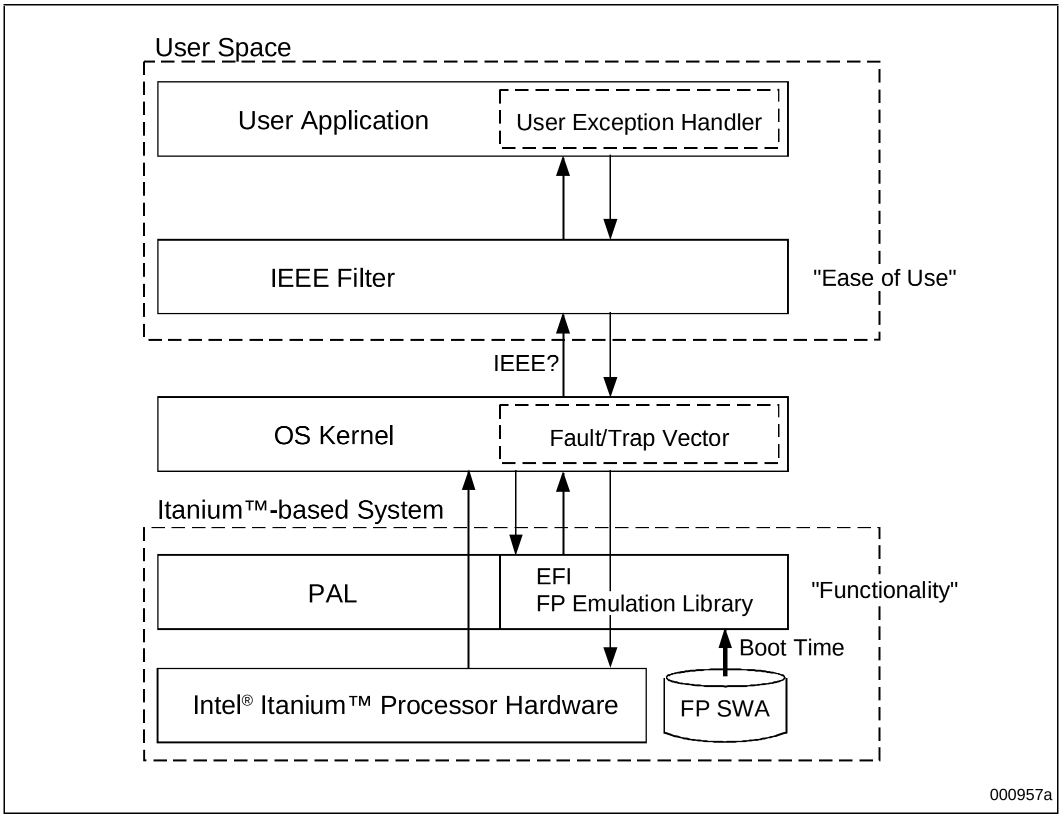
[2] Tiny numbers are non-zero values with a magnitude smaller than the smallest normal floating-point number.

[3] Huge numbers have values larger in magnitude than the largest normal floating-point number.

simpler task of taking the intermediate result of the first rounding and producing the correctly signed maximum representable normal or infinity, based on the sign of the result, the rounding direction, and the result precision and range.

Note: The Itanium architecture also allows for SWA Traps to be raised when the result is just Inexact. This is a trivial case for the SWA Trap handler, since result of the second IEEE rounding is identical to the first IEEE rounding.

Figure 8-1. Overview of Floating-point Exception Handling in the Intel® Itanium™ Architecture



The general flow of the SWA Trap handler is as follows:

1. From the interruption instruction previous address (IIPA) and exception instruction index (ISR.ei), determine the FP instruction that trapped.
2. From the instruction, decode the opcode, static precision, status field and input/output register specifiers.
3. From the ISR.code and FPSR trap enable controls, determine if a SWA Trap has occurred, if not go to the last step.
4. Read the first IEEE rounded result from the FR output register.
5. From the opcode and the status field, decode the result range and precision.
6. From the ISR.code's FPA, O, U, and I status bits and the intermediate result, produce the Itanium architecture specified result.
7. Place the result in the output FR register.
8. Update the flags in the appropriate status field of the FPSR, if required.

9. Update the ISR.code if required. (This is required if the SWA trap has been translated into an IEEE trap.)
10. Check to see if an IEEE trap needs to be raised. If so, then queue it to the IEEE Filter, otherwise continue checking for lower priority traps that may need to be raised and if required invoke their handler. When finished, continue execution at the next instruction.

8.1.1.3 Approximation Instructions and Architecturally Mandated SWA Faults

The scalar approximation instructions, `frcpa` and `frsqrta`, can raise architecturally mandated SWA Faults. This occurs when their input operands are such that they are potentially prevented from generating the correct result by the usual software algorithms that are employed for divide and square root. The reasons for this are that these algorithms may suffer from underflow, overflow, or loss of precision, because the inputs or result are at the extremes of their range. For these special cases, the SWA Fault handler must use alternate algorithms to provide the correct quotient or square root and place that result in the floating-point destination register. The predicate destination register is also cleared to indicate the result is not an approximation that needs to be improved via the iterative algorithm.

The parallel approximation instructions `fprcpa` and `fprsqrta` have situations similar to the scalar approximation instruction’s architecturally mandated SWA Faults. This occurs when their input operands are such that they are potentially prevented from generating the correct result by the usual software algorithms that are employed for divide and square root. For these special cases, instead of generating a SWA Fault, the parallel approximation instructions indicate that software must use alternate algorithms to provide the correct reciprocal or square-root reciprocal by clearing the destination predicate register. The cleared predicate is the indication to the parallel IEEE-754 divide and square root software algorithms that alternative algorithms are required to produce the correct IEEE-754 quotient or square root.

8.1.2 The IEEE Floating-point Exception Filter

The Itanium architecture supports the reporting of the five IEEE-754 standard floating-point exceptions and the IA-32 Denormal Operand exception. In the Itanium architecture the Denormal Operand exception is expanded to the Denormal/Unnormal Operand exception. When referring to the IEEE-754 exceptions in the Itanium architecture the Denormal/Unnormal Operand exception is included.

At the application level, a user floating-point exception handler could handle the Itanium floating-point exception directly. This is the traditional operating system approach of providing a signal handler with a pointer to a machine-dependent data structure. It would be more convenient for the application developer if the operating system were to first transform the results to make them IEEE-754 conforming and then present the exception to the user in an abstracted manner. It is recommended that the operating system include such a software layer to enable application developers that want to handle floating-point exceptions in their application. The IEEE Floating-point Exception Filter provides this convenience to the developer through three functions.

- The first function of the IEEE Filter is to map the Itanium architecture's result to the IEEE-754 conforming result. This includes the wrapping of the exponent for Overflow and Underflow exceptions. The Itanium architecture keeps the exponent in the 17-bit format, which is not wrapped (i.e. scaled) with the appropriate value for the destination precision.
- The second function of an IEEE Filter is to transform the interruption information to a format that is easier to interpret and to invoke a user handler for the exception. The user's handler may

then provide a value to be substituted for the IEEE default result, based on the operation, exception and inputs.

- The third function of the filter is to hide the complexities of the parallel instructions from the user. If a floating-point fault occurs in the high half of a parallel floating-point instruction and there is a user handler provided, the parallel instruction is split into two scalar instructions. The result for the high half comes from the user handler, while the low half is emulated by the IEEE Filter. The two results are combined back into a parallel result and execution is continued. More complicated cases can also occur with multiple faults and/or traps occurring in the same instruction.

Note: Usage of the IEEE Filter should not be compulsory - the user should be able to choose to handle enabled floating-point exceptions directly. The IEEE filter just hides the details of the instruction set and frees the user handler from having to emulate instructions directly and potentially incorrectly.

8.1.2.1 Invalid Operation Exception (Fault)

The exception-enabled response of an Itanium floating-point arithmetic instruction to an Invalid Operation exception is to leave the operands unchanged and to set the V bit in the ISR.code field of the ISR register. The operating system kernel, reached via the floating-point fault vector, will then invoke the user floating-point exception handler, if one has been registered.

8.1.2.2 Divide by Zero Exception (Fault)

The exception-enabled response of an Itanium floating-point arithmetic instruction to a Divide-by-Zero exception is to leave the operands unchanged and to set the Z bit in the ISR.code field of the ISR register. The operating system kernel, reached via the floating-point fault vector, will then invoke the user floating-point exception handler, if one has been registered.

8.1.2.3 Denormal/Unnormal Operand Exception (Fault)

The exception-enabled response of the Itanium arithmetic instruction to a Denormal/Unnormal Operand exception is to leave the operands unchanged and to set the D bit in the ISR.code field of the ISR register. The operating system kernel, reached via the floating-point fault vector, will then invoke the user floating-point exception handler, if one has been registered.

8.1.2.4 Overflow Exception (Trap)

The exception-enabled response of an Itanium floating-point arithmetic instruction to an Overflow exception is to deliver the first (exponent unbounded) IEEE rounded result, and to set the O bit (and possibly the I and FPA bits) in the ISR.code field of the ISR register and the Overflow flags (and possibly the Inexact flag) in the appropriate status field of the FPSR register.

The IEEE-754 standard requires that, when raising an overflow exception, the user handler should be provided with the result rounded to the destination precision with the exponent range unbounded. For the huge result to fit in the destination's range, it must be scaled down by a factor equal to 2.0^{a} (with a equal to $3*2^{n-2}$, where n is the number of bits in the exponent of the floating-point format used to represent the result.) This scaling down will bring the result close to the middle of the range covered by the particular format. The exponent adjustment factors to do the scaling for the various formats are determined as follows:

- 8-bit (single) exponents are adjusted by $3*2^{6}$ = 0xc0 = 192.

- 11-bit (double) exponents are adjusted by $3*2^9$ = 0x600 = 1536.
- 15-bit (double-extended) exponents are adjusted by $3*2^{13}$ = 0x6000 = 24576.
- 17-bit (register) exponents are adjusted by $3*2^{15}$ = 0x18000 = 98304.

The actual scaling of the result is not performed by the Itanium architecture. The IEEE filter that is invoked before calling the user floating-point exception handler typically performs the scaling.

8.1.2.5 Underflow Exception (Trap)

The exception-enabled response of an Itanium floating-point arithmetic instruction to an Underflow exception is to deliver the first (exponent unbounded) IEEE rounded result, and to set the U bit (and possibly the I and FPA bits) in the ISR.code field of the ISR register and the Underflow flag (and possibly the Inexact flag) in the appropriate status field of the FPSR register.

The IEEE-754 standard requires that, when raising an underflow exception, the user handler should be provided with the result rounded to the destination precision with the exponent range unbounded. For the tiny result to fit in the destination's range, it must be scaled up by a factor equal to 2.0^a (with a equal to $3*2^{n-2}$, where n is the number of bits in the exponent of the floating-point format used to represent the result.). The scaling up will bring result close to the middle of the range covered by the particular format. The exponent adjustment factors to do this scaling for the various formats are the same as those for enabled overflow exceptions, listed above.

Just as for overflow, the actual scaling of the result is not performed by the Itanium architecture. It is typically performed by the IEEE Filter, which is invoked before calling the user floating-point exception handler.

8.1.2.6 Inexact Exception (Trap)

The exception-enabled response of an Itanium arithmetic instruction to an Inexact exception is to set the I bit (and possibly the FPA bit) in the ISR.code field of the ISR register and the Inexact flag in the appropriate status field of the FPSR register. The operating system kernel, reached via the floating-point fault vector, will then invoke the user floating-point exception handler, if one has been registered.

8.2 IA-32 Floating-point Exceptions

IA-32 floating-point exceptions may occur when executing code in IA-32 mode. When this happens, execution is transferred to the Itanium interruption vector for IA-32 Exceptions (at vector address 0x6900.) For classic IA-32 floating-point instructions, they are raised via the "IA_32_Exception(FPError) - Pending Floating-point Error". For Streaming SIMD Extension (SSE) instructions, they are raised via the "IA_32_Exception(StreamingSIMD) - Streaming SIMD Extension Numeric Error Fault". The operating system may schedule Itanium-based and/or IA-32 exception handlers for these exceptions.

IA-32 Application Support 9

The Itanium architecture enables Itanium-based operating systems to host IA-32 applications, Itanium-based applications, as well as mixed IA-32/Itanium-based applications. Unless the operating system explicitly intercepts ISA transfers (using the PSR.di), user-level code can transition between the two instruction sets without operating system intervention. This allows IA-32 programs to call Itanium-based subroutines or vice-versa. Itanium-based and IA-32 code can share data through registers and/or memory. Multi-threaded IA-32 and Itanium-based applications can easily communicate with each other or the Itanium-based operating system using shared memory. The Itanium architecture does not support execution of Itanium-based programs on an IA-32 operating system. While the architecture does not prevent IA-32 code from executing as part of an Itanium-based operating system, it is strongly recommended that Itanium-based operating systems do **not** contain IA-32 code.

One of the most compelling motivations for executing IA-32 code on an Itanium-based operating system is the ability to run existing unmodified IA-32 application binaries. Because IA-32 performs 32-bit instruction/memory references that are zero-extended into 64-bit virtual addresses, Itanium-based operating systems must ensure that all IA-32 code and data is located in the lower 4GBytes of the virtual address space. Compute intensive IA-32 applications can improve their performance substantially by migrating compute kernels from IA-32 to Itanium-based code while preserving the bulk of the application's IA-32 binary code. If mixed IA-32/Itanium-based applications are supported, care has to be taken that the data accessible to IA-32 portions of the application is located in the lower 4GBytes of the virtual address space.

While processors based on the Itanium architecture are capable of supporting a wide range of Itanium-based/IA-32 code mixing, Itanium-based operating systems need to provide a software support infrastructure to enable full interoperability between the IA-32 and Itanium instruction set. Most Itanium-based operating systems are expected to support user-level IA-32 applications, and, as a result, must be able to provide the full range of operating system services through a 32-bit system call interface. However, different operating systems and run-time conventions may reduce the set of interoperability modes as desired by the operating system vendor.

While it is an interesting topic, this chapter does not discuss 32-bit application binary interfaces provided by specific operating systems. Instead, this chapter focusses on what services are required from an Itanium-based operating system by a processor based on the Itanium architecture that is executing IA-32 code. In other words, the focus of this chapter is the low-level processor / operating system interface rather than the IA-32 software / operating system (application binary) interface.

9.1 Transitioning between Intel® Itanium™ and IA-32 Instruction Sets

As mentioned earlier, user-level code can transition from Itanium to IA-32 (or back) instruction sets without operating system intervention. As described in Chapter 6, "IA-32 Application Execution Model in an Intel® Itanium™ System Environment" in Volume 1, two instructions are provided for this purpose: `br.ia` (an Itanium unconditional branch), and JMPE (an IA-32 register indirect and absolute jump). Prior to executing any IA-32 instructions, however, the Itanium-based operating system needs to setup an execution environment for executing IA-32 code.

9.1.1 IA-32 Code Execution Environments

Processors based on the Itanium architecture are capable of executing IA-32 code in real mode, VM86 mode or protected mode. When segmentation is enabled both 16 and 32-bit code are supported. Prior to transferring control to IA-32 code, an Itanium-based application and/or operating system is expected to setup the complete IA-32 execution environment in Itanium registers.

In particular, Itanium-based software must setup IA-32 segment descriptor and selector registers in Itanium application registers, and must ensure that code and stack segment descriptors (CSD, SSD) are pointing at valid and correctly aligned memory areas. It is also worth noting that the IA-32 GDT and LDT descriptors are maintained in GR30 and GR31, and are unprotected from Itanium-based user-level code. For more details on the IA-32 execution environment please refer to Section 6.2 "IA-32 Application Register State Model" in Volume 1.

Some IA-32 execution environments may need support from an Itanium-based operating system. Which IA-32 software environments are supported by an Itanium-based operating system is determined by the operating system vendor. Itanium-based platform firmware (SAL) provides a run-time environment that allows execution of real-mode IA-32 code found in PCI configuration option ROMs.

9.1.2 br.ia

`br.ia` is an unconditional indirect branch that transitions from Itanium to IA-32 instruction set. Prior to entering IA-32 code with `br.ia`, software is also required to flush the register stack. `br.ia` sets the size of the current register stack frame to zero. The register stack is disabled during IA-32 code execution. Because IA-32 code execution uses Itanium registers, much of the Itanium register state is overwritten and left in an undefined state when IA-32 code is run. As a result, software can not rely on the value of such registers across an instruction set transition. Execution of IA-32 code also invalidates the ALAT. For more details refer to Table 6-2 in Volume 1.

For best performance, the following code sequence is recommended for transitioning from Itanium to IA-32 instruction set:

```
{.mii
    flushrs           // flush register stack
    mov b7 = rTarget  // Setup IA-32 target address
    nop.i             // nop.i or other instruction
    ;;
{.mib
    nop.m             // nop.m or other instruction
    nop.i             // nop.i or other instruction
    br.ia.sptk b7     // branch to IA-32 target defined by
                      // lower 32-bits of branch register b7
    ;;
```

Key to performance is that the register stack flush (`flushrs`) and the `br.ia` instruction are separated by a single cycle, and that the `br.ia` instruction is the first B-slot in the bundle directly following the `flushrs`. The `nop` instruction slots in the code example may be used for other instructions.

9.1.3 JMPE

JMPE is an IA-32 instruction that comes in a register indirect and absolute branch flavors. The code segment descriptor base is held in the CSD application register (ar.csd).

- JMPE reg16/32 computes the target of the Itanium instruction set as
  `IP = ([reg16/32] + CSD.base) & 0xfffffff0`
- JMPE disp16/32 computes the target of the Itanium instruction set as
  `IP = (disp16/32 + CSD.base) & 0xfffffff0`

Targets of the IA-32 JMPE instruction are forced to be 16-byte aligned, and are constrained to the lower 4Gbytes of the 64-bit virtual address space. The JMPE instruction leaves the IA-32 return address (address of the IA-32 instruction following the JMPE itself) in IA_64 register GR1.

9.1.4 Procedure Calls between Intel® Itanium™ and IA-32 Instruction Sets

If procedure call linkage is required between Itanium-based and IA-32 subroutines, software needs to perform additional work as described in the next two sections.

9.1.4.1 Intel® Itanium™-based Caller to IA-32 Callee

This section outlines what steps an Itanium-based caller of an IA-32 procedure needs to perform. The ordering of the steps is approximate and need not be executed exactly in the order presented.

1. Setup IA-32 execution environment, if not already done (see Section 9.1.2 for details). Ensure that no NaTed registers are used to setup IA-32 environment nor that they are passed as procedure call arguments to IA-32 code.
2. Marshall arguments from the register stack to memory stack according to IA-32 software conventions.
3. Set up exception handle unwind data structures according to OS convention.
4. Make sure JMPE knows where to return to, e.g. deposit return address for the JMPE on memory stack or pass it in an IA-32 visible register.
5. Setup IA-32 branch target in branch register.
6. Flush register stack, but no other RSE updates.
7. `br.ia` is an indirect branch to IA-32 code. There is no need to preserve Itanium only application registers, since IA-32 code execution leaves them unmodified.
8. Run in the IA-32 callee until it executes a JMPE instruction.
9. JMPE instruction is an unconditional jump to Itanium-based code. JMPE should use the return address specified in step 4.
10. Move return values from memory stack to static Itanium register used for procedure return value according to Itanium calling conventions.
11. Ensure that IA-32 code correctly unwound memory stack, and that memory stack pointer is correctly aligned.
12. Update exception handle unwind data structures according to OS convention.
13. `br.ret` returns to Itanium-based caller.

9.1.4.2 IA-32 Caller to Intel® Itanium™-based Callee

This section outlines what steps an IA-32 caller of an Itanium-based procedure needs to perform. The ordering of the steps is approximate and need not be executed exactly in the order presented.

1. Caller deposits arguments on memory stack, and calls Itanium-based transition stub using the JMPE instruction.
2. Execute JMPE instruction as an unconditional branch to Itanium-based code. The JMPE instruction will leave the address of the IA-32 instruction following the JMPE itself in Itanium register GR1. This address may be used as a return address later.
3. Allocate a register stack frame with the `alloc` instruction.
4. Load procedure arguments from memory stack into Itanium stacked registers. Preserve IA-32 return address in memory or register stack.
5. Set up exception handle unwind data structures according to OS convention.
6. `br.call` to target Itanium-based callee.
7. Execute Itanium-based code until it returns using `br.ret`.
8. Move return value from static Itanium register to memory stack.
9. Load IA-32 return address from step 4 into branch register.
10. Instead of flushing the register stack to memory, the contents of the register stack can be discarded at this point since IA-32 code execution will overwrite it anyway. Invalidate register stack by:
    a. Allocating a zero-size stack frame using the `alloc` instruction.
    b. Writing zero into RSC application register, and executing a `loadrs` instruction.
    c. Restore RSC application register to its original value in preparation for the next call from IA-32 to Itanium instruction set.
11. Ensure memory stack pointer is correctly aligned prior to returning to IA-32 code.
12. `br.ia` returns to IA-32 caller.

9.2 IA-32 Architecture Handlers

An Itanium-based operating system needs to be prepared to handle exceptions from Itanium-based and IA-32 code. Depending on the exception cause, exception vectors can be:

- Shared Itanium/IA-32 Exception Vectors: all virtual memory related instruction and data reference faults share a common exception vector, regardless of whether they were caused by Itanium-based or IA-32 code.
- Unique Itanium Exception vectors: these are conditions that only Itanium-based code can cause. Examples are: Instruction Breakpoint fault, Illegal Operation fault, Illegal Dependency fault, Unimplemented Data Address fault, etc.
- Unique IA-32 Exception Vectors: these conditions can occur only from IA-32 instructions.

A detailed break-down of which exceptions occur on which interruption vector and from which instruction set is given in Table 5-5. Table 9-1 shown below summarizes all IA-32 related exceptions that an Itanium-based operating system needs to be ready to handle. These IA-32 specific interrupts are grouped into three vectors: the IA-32 Exception vector, the IA-32 Intercept, and the IA-32 Interrupt vector. Within each of these vectors the interrupt status register (ISR) provides detailed codes as to the origin of this exception. Details on the IA-32 vectors is provided in Chapter 9, "IA-32 Interruption Vector Descriptions". More details on debug related IA-32 exceptions is given in the following section of this document.

Table 9-1. IA-32 Vectors that Need Itanium™-based OS Support

Vector (IVA offset)	Exception Name	Exception Related To	Expected OS Behavior
IA-32 Exception vector (0x6900)	IA-32 Instruction Debug fault	Debug	Relay to debugger.
	IA-32 Code Fetch fault	Segmentation	Signal application.
	IA-32 Instruction Length > 15 bytes fault	Bad Opcode	Signal application.
	IA-32 Device Not Available fault	Numeric	Signal application.
	IA-32 FP Error fault	Numeric	Signal application.
	IA-32 Segment Not Present fault	Segmentation	Signal application.
	IA-32 Stack Exception fault	Segmentation	Signal application.
	IA-32 General Protection fault	Segmentation	Signal application.
	IA-32 Divide by Zero fault	Numeric	Signal application.
	IA-32 Alignment Check fault	Misaligned IA-32 Memory Reference with alignment checking enabled.	Depends on convention.
	IA-32 Bound fault	Segmentation	Signal application.
	IA-32 Streaming SIMD Extension Numeric Error Fault	Numeric	Signal application.
	IA-32 INTO Overflow trap	Numeric	Signal application.
	IA-32 Breakpoint (INT 3) trap	Software Breakpoint	Depends on convention.
	IA-32 Data Breakpoint trap	Debug	Relay to debugger.
	IA-32 Taken Branch trap	Debug	Relay to debugger.
	IA-32 Single Step trap	Debug	Relay to debugger.
	IA-32 Invalid Opcode fault	Bad Opcode	Signal application.
IA-32 Intercept vector (0x6a00)	IA-32 Instruction Intercept fault	Attempted to access IA-32 paging, MTRRs, IDT, IA-32 control registers, IA-32 debug registers or attempted to execute IA-32 privileged instructions.	This is not supported on an Itanium™-based OS. Signal application.
	IA-32 Locked Data Reference fault	Attempt to reference misaligned or uncacheable semaphore.	Emulation handler if needed. Refer to Section 2.1.3.2, "Behavior of Uncacheable and Misaligned Semaphores" on page 2:377.
	IA-32 System Flag Intercept trap	System Flag intercept	Depends on convention.
	IA-32 Gate Intercept trap	Gate/Task transfer intercept	Depends on convention.
IA-32 Interrupt vector (0x6b00)	IA-32 Software Interrupt (INT) trap	Software Interrupt	Depends on convention.
Cannot happen in Itanium™-based operating system	IA-32 Double Fault IA-32 Invalid TSS Fault, IA-32 Page Fault, IA-32 Machine Check	N/A	Don't worry,

9.3 Debugging IA-32 and Intel® Itanium™-based Code

Itanium-based operating systems that want to provide debug support for both IA-32 and Itanium-based applications, need to be aware of the differences between taking instruction and data breakpoint exceptions as well as single step or taken branch traps on Itanium and IA-32 instructions.

9.3.1 Instruction Breakpoints

If an Itanium instruction matches an instruction breakpoint register (IBR) then an Instruction Debug Fault is delivered on the Itanium Debug vector. To step across a single Itanium instruction, IPSR.id must be set to one. An IA-32 instruction, however, that matches an IBR causes an IA-32 Instruction Breakpoint fault which is delivered to the IA-32 Exception vector (Debug). To step across a single IA-32 instruction, either IPSR.id or EFLAGS.rf must be set to one.

9.3.2 Data Breakpoints

If an Itanium memory reference matches a data breakpoint register (DBR) then a Data Debug Fault is delivered on the Itanium Debug vector. To step across a single data breakpoint, IPSR.dd must be set to one. An IA-32 instruction, however, that matches a DBR causes an IA-32 Data Breakpoint *trap* which is delivered to the IA-32 Exception vector (Debug). In other words, the debugger only gets control after the instruction making the reference has completed. Since IA-32 instruction can make multiple memory references, a single IA-32 instruction may cause multiple data break points to trigger. Details on how this is communicated to software in the interrupt status register (ISR) is given in Section 9.1 "IA-32 Trap Code". Since IA-32 data breakpoints are traps, there is no need to step over them.

9.3.3 Single Step Traps

When PSR.ss enables single stepping of Itanium-based applications, each instruction that is stepped will stop at the Single Step trap handler. When PSR.ss or EFLAG.tf enable single stepping of IA-32 applications, an IA-32_Exception(Debug) trap is taken after each IA-32 instruction. For more details refer to Section 9.1 "IA-32 Trap Code".

9.3.4 Taken Branch Traps

When PSR.tb enables taken branch trapping on Itanium-based applications, each taken branch will transfer control to the Taken Branch Trap handler. When PSR.tb is set, taken IA-32 branches transfer control to the IA-32_Exception(Debug) trap handler taken after each IA-32 instruction. For more details refer to Section 9.1 "IA-32 Trap Code".

External Interrupt Architecture 10

The Itanium architecture provides a high performance external interrupt architecture. While IA-32 processors commonly use a three wire shared APIC bus, processors based on the Itanium architecture utilize a high performance message based point-to-point protocol between processors and multiple I/O interrupt controllers. To ensure that processors based on the Itanium architecture can fully leverage the large set of existing platform infrastructure and I/O devices, compatibility with existing platform infrastructure is provided in the form of direct support for Intel® 8259A compatible interrupt controllers and limited support for level sensitive interrupts.

This chapter introduces the basic external interrupt mechanism provided by the architecture, while Section 5.8 "Interrupts" provides the complete architectural definition for the Itanium external interrupt architecture.

10.1 External Interrupt Basics

Interrupts are identified by their vector number. The vector number implies interrupt priority, and also determines whether the interrupt is delivered to processor firmware as a “PAL-based” interrupt, or whether it is delivered to the operating system as an “IVA-based” external interrupt.

This chapter discusses asynchronous external interrupts only. PAL-based platform management interrupts (PMI) are not discussed here. External interrupts are IVA-based and are delivered to the operating system by transferring control to code located at address CR[IVA]+0x3000. This code location is also known as the external interrupt vector and is described on page 2:166.

Software can distinguish interrupts based on their vector number. Vector numbers range from 0 to 255. Vector numbers also establish interrupt priorities as follows:

- Vector numbers below 16 are special, and are architecturally defined in Section 5.8.1 "Interrupt Vectors and Priorities". The non-maskable interrupt (NMI) is always vector 2 and is higher priority than all in-service external interrupts. ExtINT, Intel 8259A compatible external interrupt controller interrupt, is always vector 0. Vector numbers below 16 have higher priority than vectors above 16. Vector 15 is used to indicate that the highest priority pending interrupt in the processor is at a priority level that is currently masked or there are no pending external interrupts.
- For vector numbers between 16 and 255, higher vector numbers imply higher priority. In this range, vectors are freely assignable by software. This is achieved by programming of interrupt controllers and the processor internal interrupt configuration registers.

10.2 Configuration of External Interrupt Vectors

As defined in Section 5.8 "Interrupts", external interrupts originate from one of four sources:

- From external sources, e.g. external interrupt controllers or intelligent external I/O devices, or
- From the processor's LINT0 or LINT1 pins (typically connected to an Intel 8259A compatible interrupt controller), or
- From internal processor sources, e.g. timers or performance monitors, or
- From other processors, e.g. inter-processor interrupts (IPIs).

All interrupts are point-to-point communications. There is no facility for broadcasting of interrupts. The interrupt message protocol used by the processor-to-processor and the external source-to-processor is not defined architecturally, and is not visible to software.

A number of external interrupt control registers (LID,TPR, ITV, PMV, CMCV, LRR0 and LRR1) allow software to directly configure the processor interrupt resources. The Local ID register (LID) establishes a processor's unique physical interrupt identifier. The Task Priority Register (TPR) allows masking of external interrupts based on vector priority classes. The ITV, PMV, CMCV, LRR0 and LRR1 external interrupt control registers configure the vector number for the processor's local interrupt sources. Configuration of the external controllers and devices is controller/device specific, and is beyond the scope of this document.

10.3 External Interrupt Masking

The Itanium architecture provides four mechanisms to prevent external interrupts from being delivered to a processor: a bit in the processor status register (PSR.i), the interrupt vector register (IVR) and the end-of-interrupt (EOI) register, the task priority register (TPR), and the external task priority register (XTPR). The next four sections discuss these mechanisms.

10.3.1 PSR.i

When PSR.i is zero, the processor does not accept any external interrupts. However, interrupts continue to be pended by the processor. Software can use PSR.i to temporarily disable taking of external interrupts, e.g. to ensure uninterruptable execution of critical code sections. Since clearing of PSR.i takes effect immediately (refer to the rsm instruction page), software is not necessarily required to explicitly serialize clearing of PSR.i (unless another processor resource requires serialization). On the way out of an uninterruptable code section software is not required to serialize the setting of PSR.i either, unless it is of interest to software to be able to take interrupts in the very next instruction group. A code example for this case is given below:

```
rsm i ;;
// rsm of PSR.i takes effect on the next instruction

// uninterruptable code sequence here

ssm i ;;
// ssm of PSR.i does require data serialization, if we need to ensure
// that external interrupts are enabled at the very next instruction. If
// data serialization is omitted, PSR.i is set to 1 at the latest when
// the next exception is taken.
```

By avoiding the serialization operations on PSR.i the performance of such uninterruptable code sections is improved.

10.3.2 IVR Reads and EOI Writes

As described in Section 10.4, IVR reads return the highest priority, pending, unmasked vector, and places this vector "in-service". Additionally, IVR reads have the side-effect of masking all vectors that have equal or lower priority than one that is returned by the IVR read. Correspondingly, writes to the EOI register unmask all vectors with equal or lower priority than the highest priority "in-service" vector. Due to nesting of higher priority interrupts, it is possible to have multiple vectors in the "in-service" state.

10.3.3 Task Priority Register (TPR)

The Task Priority Register (TPR) provides an additional interrupt masking capability. It allows software to mask interrupt "priority classes" of 16 vectors each by specifying the mask priority class in the TPR.mic field. The TPR.mmi field allows masking of all maskable external interrupts (essentially all but NMI).

An example of TPR use is shown in Section 10.5.2, "TPR and XPTR Usage Example" on page 2:467.

10.3.4 External Task Priority Register (XTPR)

The External Task Priority Register (XTPR) is a per-processor resource that can be provided by external bus logic in some Itanium-based platforms. If supported by the platform, XTPR can be used by the operating system to redirect external interrupts to other processors in a multi-processor system.

The XTPR is updated by performing a 1-byte store to the XTP byte which is located at an offset of 0x1e0008 in the Processor Interrupt Block (see Section 5.8.4 "Processor Interrupt Block" for details). Since the timing of the modification of the XTP register is not time critical there is no serialization required. Effects of the one byte store operation are platform specific. Typically, it will generate a transaction on the system bus identifying it as an XTP register update transaction, and will indicate which processor generated the transaction as well as the stored data.

An example of XTPR use is included in Section 10.5.2, "TPR and XPTR Usage Example" on page 2:467.

10.4 External Interrupt Delivery

The architectural interrupt model in Section 5.8 defines how each interrupt vector cycles through one of four states:

- *Inactive*: there is no interrupt *pending* on this vector.
- *Pending*: an interrupt has been received by the processor on this vector, but has not been *accepted* by the processor and has not been *acquired* by software. The processor hardware will *accept* the interrupt when this vector's priority level is higher than the highest currently

in-service vector, PSR.i is one, and TPR settings do not mask the interrupt. This will cause the processor to transfer control flow to the external interrupt handler. Software can then *acquire* the highest priority, pending, unmasked vector by reading the IVR control register. The IVR read returns the 8-bit vector number in a register and masks all vectors that have equal or lower priority. This vector now enters the In-Service/None Pending state.

- *In-Service/None Pending*: an interrupt has been received by the processor on this vector, and has been acquired by software (by reading the IVR control register), but software has not *completed servicing* this interrupt. In this state, the processor masks all vectors that have equal or lower priority. In this state, the processor can receive and remember a second interrupt on this vector. If this happens, the processor transitions this vector to the “In-Service/One Pending” state. If software *completes the interrupt* service routine (indicated to the processor by writing the EOI register) before another interrupt is received on this vector, then the processor returns this vector to the Inactive state, and all vectors with equal or lower priority are unmasked.
- *In-Service/One Pending*: an interrupt has been received by the processor on this vector, and has been acquired by software (by reading the IVR control register), and software has not completed servicing this interrupt. Additionally, the processor received a second interrupt on this vector, which is now held pending. If additional interrupts on this vector are received by the processor while this vector is in the “In-Service/One Pending” state, those additional interrupts are not distinguishable by the processor hardware. When software completes the interrupt service routine for the original interrupt on this vector (indicated to the processor by writing the EOI register), then the processor returns this interrupt vector to the Pending state for the second interrupt that was received on this vector. Additionally, all vectors with equal or lower priority are unmasked.

It is recommended the following structure for an Itanium-based external interrupt handler:

1. Read and Save TPR, i.e. save Old Task Priority variable (optional).
2. External Interrupt Harvest Loop:
    a. Read the IVR control register to determine which vector is being delivered. If the returned IVR value is 15, then this is a spurious interrupt and it can be can ignored; software can now clear PSR.ic, restore IPSR and IIP and then `rfi` to the interrupted context. If the returned IVR value is not 15, continue with step 2b.

    b. Raise TPR register to the interrupt class to which the level read out of IVR belongs (optional).

    c. Software must preserve IIP and IPSR prior to re-enabling PSR.ic and PSR.i which will re-enable taking of exceptions and higher priority external interrupts.

    d. Issue a `srlz.d` instruction. This ensures that updated PSR.ic and PSR.i settings are visible, and it also makes sure that the IVR read side effect of masking lower or equal priority interrupts is visible when PSR.i becomes 1.

    e. Dispatch the appropriate interrupt service routine.

    f. Disable external interrupts by clearing PSR.i with an `rsm 0x4000` instruction.This ensures that external interrupts are disabled prior to the EOI write in the next step.

    g. Notify the processor that interrupt handling for this vector is completed by writing to the EOI register. This will unmask any pending lower priority interrupts. If this was a level triggered interrupt, write to the I/O SAPIC EOI register.

    h. Lower TPR register to Old Task Priority (optional).

i. Issue a `srlz.d` instruction. This ensures that ensure the EOI write from step 2g is reflected in the future IVR read (in step 2a). It also ensures that the TPR update from step 2h unmasks any interrupts in the priority classes (including the current task priority level) that were masked by the previous value of TPR.

j. Return to top of loop (step 2a).

These steps assume that the routine's caller already performed the required state preservation of interruption resources. Therefore the focus of the steps above is to check the IVR to acquire the vector so the operating system can determine what device the interrupt is associated with. The code is setup to loop, servicing interrupts until the spurious interrupt vector (15) is returned. Looping and harvesting outstanding interrupts reduces the time wasted by returning to the previous state just to get interrupted again. The benefit of interrupt harvesting is that the processor pipeline is not unnecessarily flushed and that the interrupted context is only saved/restored once for a sequence of external interrupts. Once the vector is obtained the specific interrupt service routine is called to service the device request. Upon return from the interrupt service routine, an EOI is written and the IVR is checked once again.

If the operating system does not implement priority levels then there is no need to save and restore the task priority level (steps 1, 2b, and 2h are optional). As described in Section 10.3 above, an IVR read automatically masks interrupts at the current in-service level and below until the corresponding EOI is issued. For level triggered interrupts, the programmer must not only inform the processor, but the external interrupt controller that the level triggered interrupt has been serviced.

10.5 Interrupt Control Register Usage Examples

The examples in this section provide an overview of using the Itanium external interrupt control registers. Actual and pseudo code fragments are listed to aid in the development of OS code which will utilize these registers. It is up to the operating system and its writer to determine what minimum set of control registers are required to be used.

10.5.1 Notation

Preprocessor macros for function ENTRY and END are used in the examples to reduce duplication of code and reduce document space requirements.

```
#define ENTRY(label) \
     .text; \
     .align 32;; \
     .global label; \
     .proc label; \
label::

#define END(label) .endp
```

10.5.2 TPR and XPTR Usage Example

This code will allow certain interrupts to be masked by increasing/decreasing the task priority register. If you don't want to mask all external interrupts, you can raise the priority level to mask out only the interrupts that have higher priority (and no effect on your current critical section).

We also take the expensive route here by updating not only the processor TPR, but the External Task Priority Register used by the chipset (if supported) as a hint to what processor should receive the next external interrupt.

```
//
// routine to set the task priority register to mask
// interrupts at the specific level or below
//
// INPUT: SPL level
//

TPR_MIC=4
TPR_MIC_LEN=4

.global external_task_pri_reg// address points to Interrupt Delivery block

ENTRY(set_spl)
    alloc r18=ar.pfs,1,0,0,0
    dep.z r22=r32,TPR_MIC,TPR_MIC_LEN
    movl r19=external_task_pri_reg
    ;;
    mov cr.tpr=r22
    ld8 r20=[r19]  // get address of EXt. TASK Priority Register
    ;;
    srlz.d         // srlz.d only required if want TPR update effective
immediately
    st1 [r20]=r32  // if supported by platform: update eXternal Task Priority
(XTP)
    br.ret.sptk b0
    ;;
END(set_spl)
```

10.5.3 EOI Usage Example

This example is a typical return from an interrupt service routine to the generic interrupt handler. Interrupts are disabled before returning to the main trap handler in preparation for returning from kernel space.

```
return_from_interrupt:
// disable interrupts here

    rsm 0x4000            // make sure interrupts disabled


// interrupt_eoi# clear the sapic/pic interrupt
sapic_eoi:
    mov cr.eoi=r0         // issue and eoi
    ;;
    srlz.d                // make sure it takes effect


// issue the appropriate EOI sequence to the external interrupt
// controller here.
```

For level trigger interrupts, the OS is required to issue an EOI not only to the processor, but also the external interrupt controller where the interrupt originated. This forces the OS to keep track of whether the vector is associated with a level or an edge trigger interrupt line.

10.5.4 IRR Usage Example

Waiting on an interrupt with interrupts disabled.

```
my_interrupt_loop::
//
// check for vector 192 (0xc0) via irr3
//

        mov     r3=cr.irr3
        ;;
        and     r3=0x1,r3
        ;;
        cmp.eq p6,p7=0x1,r3
    (p7)br.cond.sptk.few   my_interrupt_loop
        ;;
        mov     r4=cr.ivr       // read the vector
        ;;
        mov     cr.eoi=r0       // clear it
        ;;
```

10.5.5 Interval Timer Usage Example

The Itanium architecture provides a 64 bit interval timer for time stamps and elapsed time notification interrupts. It is equivalent to the IA-32 Time Stamp Counter (TSC). Programming the Itanium interval timer consists of initializing the ITV (CR 72), ITM (CR 1), and ITC (AR 44).

The Interval Timer Vector (ITV) specifies the external interrupt vector number for the Interval Timer Interrupts. The code examples below show how to clear and initialize the timers vector, match register, and count registers.

The Interval Time Counter (ITC) gets updated at a fixed relation to the processor clock. The ITM, Interval Timer Match, is used to determine when a interval timer interrupt is generated. When the ITC matches the ITM and the timer is unmasked via ITV then an interrupt will be generated.

```
//
// routine to reset the interval timer to zero..
//

ENTRY(em_timer_reinit)
    mov    ar.itc=r0              // reset itimer counter
    br.ret.spnt.few rp
END(em_timer_reinit)


//
// routine to setup the interval timer.
//
//  1) setup the interval timer vector
//  2) initialize the time counter to zero
//  3) initialize the match register
//
//  INPUTS: timermatch -- value to initialize ITM register with.
//          vector number -- vector to interrupt with
//  OUTPUTS: none
//
ENTRY(enable_minterval)
    alloc   r14=ar.pfs,0x2,0,0,0   // get ready for input parameters
    mov     ar.itc=r0              // initialize counter to zero
    ;;
    mov     cr.itm=r32             // set match register
    ;;
```

```
srlz.d
mov     cr.itv=r33            // set interval timer vector
;;
srlz.d                        // make sure it goes through
br.ret.sptk.few rp            // return
.endp
```

Since the ITC gets updated at a fixed relation to the processor clock, in order to find out the frequency at run time, one can use a firmware call to obtain the input frequency information to the interval time. Using this frequency information the ITM can be set to deliver an interrupt at a specific time interval (i.e. for operating system scheduling purposes). Assuming the frequency information returned by the firmware is in ticks per second, the programmer could use a time-out delta for delivering a timer interrupt every 10 milliseconds as follows:

```
timeout_delta=ticks_per_second/100;
```

where `ticks_per_second` is the frequency value returned by the firmware and `timeout_delta` will be the value added to the ITC for setting the next ITM. Therefore, the ITC is left free running, but the ITM must be updated upon every timer interrupt with its next time out match value, i.e. ITM = ITC + `timeout_delta`.

The only issue with this setup is if the timer interrupt delivery is delayed beyond the point of the original intended delivery time (i.e. ITC > ITM). This could happen if interrupts were disabled or blocked by the operating system/device driver longer than the time-out value. In this case the ITM has to be adjusted in order for the next ITM to be accurate. The following algorithm could be used to adjust the next ITM before returning from the timer interrupt handler.

```
for (;;) {
    itm_next = itm_next + timeout_delta + (read current ITC - read current ITM);
    if (itm_next < current ITC) {
        /* we missed the next interrupt already, continue */
    } else {
        set_itm(itm_next);
        break;
    }
}
```

where `itm_next` was initialized to current ITC + `timeout_delta`, and `set_itm` in Itanium-based assembly would look like:

```
.global set_itm
.proc set_itm
set_itm:
    alloc r18=ar.pfs,1,0,0,0
    mov cr.itm=r32
    ;;
    srlz.d
    br.ret.sptk b0
    ;;
.endp set_itm
```

10.5.6 Local Redirection Example

The Local Redirection Registers (LRR0-1) serves to steer external signal based interrupts that are directly connected to the processor. LRR0 and LRR1 control the external interrupt signals (pins) referred to as Local Interrupt 0 (LINT0) and Local Interrupt 1 (LINT1) respectively. The example below shows how to mask interrupt delivery on LINT0.

```
movl r18=(1<<16)
;;
mov cr.lrr0=r18
;;
srlz.d   // srlz.d is required after LRR write to ensure write effect
```

10.5.7 Inter-processor Interrupts Layout and Example

A processor generates an inter-processor interrupt (IPI) by storing a 64-bit interrupt command to an 8-byte aligned address in the Interrupt delivery region of the Processor Interrupt block. The address being stored to determines what target processor receives the IPI. The example below is an example of sending an interrupt to a specific processor based on the destination ID passed in. The destination ID consists of the Local interrupt ID and the Extended interrupt ID.

Writing to improperly aligned addresses in the delivery region or failure to store less than 64 bits can result in an invalid operation fault. The access must be uncacheable in order to generate an IPI.

```
//
// send_ipi_physical (dest_id, vector)
//
// inputs:     processor destination ID vector to send
//             (Local ID (8 bits << 8)| EID ( 8 bits))
//
//
//

.global ipi_block             // pointer to processor I/O block

IPI_DEST_EID=0x4

ENTRY(send_ipi_physical)
        alloc r19=ar.pfs,2,0,0,0
        movl r17=ipi_block;;
        ld8 r17=[r17]         // get pointer to processor block
        shl r21=r32,IPI_DEST_EID;;
        add r20=r21,r17;;     // point to proper processor
        st8.rel [r20]=r33     // send the IPI
        br.ret.sptk b0;;

END(send_ipi_physical)
```

10.5.8 INTA Example

External interrupt controllers, that are compatible with the Intel 8259A interrupt controller can not issue interrupt messages, so the vector number is not available at the time of the interrupt request. When an interrupt is accepted the software must check to see if it came from an external controller by the vector number (via IVR) to see if it is the ExtINT vector.

Once the software determines it is an ExtINT, it must obtain the actual vector by doing an uncached 1 byte load from the INTA byte located in the upper half of the processor interrupt block, offset 0x1e0000 from the base.

```
EXTINT=r0
INTA_PHYS_ADDRESS=0x80000000fefe0000
inta_address=r31

        movl inta_address=INTA_PHYS_ADDRESS
        ;;
        srlz.d             // make sure everything is up to date
        mov r14 = cr.ivr   // read ivr
        ;;
        srlz.d             // serialize before the EOI is written...
        ;;
        cmp.ne  p1,p2 = EXTINT,r14 ;;
    (p1)br.cond.sptk process_interrupt
        ;;

//
// A single byte load from the INTA address should cause
// the processor to emit the INTA cycle on the processor
// system bus. Any Intel 8259A compatible external interrupt
// controller must respond with the actual interrupt
// vector number as the data to be loaded.
//
//
        ld1 r17 = [inta_address]  // get the real vector..
        ;;
// vector obtained

process_interrupt:
```

I/O Architecture 11

I/O devices can be accessed from Itanium-based programs using regular loads and stores to uncacheable space. While cacheable Itanium memory references may be reordered by the processor, uncacheable I/O references are always presented to the platform in program order. This "sequentiality" of uncacheable references is discussed in Section 2.2.2, "Memory Attributes" on page 2:391 and in more detail in Section 4.4.7, "Sequentiality Attribute and Ordering".

Additionally, uncacheable memory pages are defined to be "non-speculative" which causes all data and control speculative loads to uncacheable pages to defer. Control speculative loads to uncacheable memory return a NaT/NaTVal to their target register. Data speculative loads to uncacheable memory return zero to their target register. For details, refer to Section 4.4.6, "Speculation Attributes".

When configuring chipset registers or setting up device registers, it is sometimes required to know when a memory transaction has been completed. Completion means the processor received acknowledgment that the transaction finished successfully in the platform, and that all its side-effects have occurred and will be visible to the next memory operation (issued by the same processor). To ensure completion of prior accesses on the platform, the Itanium architecture provides the `mf.a` instruction. Unlike the `mf` instruction that waits for **visibility** of prior operations, the `mf.a` waits for **completion** of prior operations on the platform. More details in Section 11.1.

To fully leverage the large set of existing platform infrastructure and I/O devices, the architecture also supports the IA-32 platform I/O port space. The Itanium instruction set does not provide IN and OUT instructions, but they can be emulated. The I/O port space can be mapped into user-space, and IA-32 applications can use IN and OUT instructions to directly communicate with the I/O port space. More details in Section 11.2.

The Itanium architecture provides a high-performance, high-bandwidth uncacheable memory attribute that supports write-coalescing. This allows the processor to burst writes to uncacheable locations at much higher bandwidth. The Itanium architecture does **not** guarantee the FIFO delivery of write-coalescing stores. More details in Section 4.4.5, "Coalescing Attribute".

11.1 Memory Acceptance Fence (mf.a)

An `mf` instruction ensures that all cache coherent agents have observed all prior memory operations made by the processor issuing the `mf`. However, it does **not** ensure that those operations have completed, in the Itanium architecture parlance it does not ensure that they have been "accepted" by the external platform. For instance, a load may have been made visible to all processors by snooping their caches, but the data return may still be in progress. Such a load would be visible, but not complete.

The `mf.a` instruction on the other hand ensures that all prior data memory references made by the processor issuing the `mf.a` have been "accepted" by the external platform. However by itself the `mf.a` does **not** guarantee that all cache coherent agents have observed all prior memory operations. For instance, an uncacheable store to a chipset register may have completed on the system bus, however, that does not entail that all prior cacheable transactions (from the processor issuing the store) have been observed by all other processors in the coherence domain.

If software needs to ensure that all prior memory operations have been accepted by the platform **and** have been observed by all cache coherent agents, both an `mf.a` and an `mf` instruction must be issued. The `mf.a` must be issued first, and the `mf` must be issued second. For more details on memory ordering between cache coherent agents please refer to Chapter 2, "MP Coherence and Synchronization".

Typically `mf`.a is used to configure a system's I/O space, e.g. to setup chipset registers that affect all subsequent memory operations. Specifically, the mf.a instruction restrains further data accesses from initiating on the external platform interface until:

1. All previous sequential (i.e. non write-coalescing uncacheable) loads have been returned data, and
2. All previous stores have been "accepted" by the platform. Typically acceptance is indicated by a bus specific signals/phase, e.g. completion of response phase on the system bus.

Architecturally, the definition of "acceptance" is platform dependent. The next section discusses the usage of the `mf`.a instruction in the context of the I/O port space.

11.2 I/O Port Space

IA-32 processors support two I/O models: memory mapped I/O and the 64KB I/O port space. To support IA-32 platforms, the Itanium architecture allows operating systems to map the 64KB I/O port space into the 64-bit virtual address space. This allows Itanium-based operating systems to see all I/O devices as a single unified memory mapped I/O model, and permits "normal" Itanium load and store instructions as well as IA-32 IN and OUT instructions to directly access the I/O port space.

As described in Section 10.7, "I/O Port Space Model", Itanium-based operating systems can map the physical 64KB I/O port space into a spread-out 64MB block of virtual address space. The virtual base address of the I/O port space (IOBase) is maintained by the operating system in kernel register KR0. When the processor issues Itanium load and stores accesses to the I/O port space, a port's virtual address is computed as:

```
port_virtual_address = IOBase | (port{15:2}<<12) | port{11:0}
```

For Itanium loads and stores, this address computation places four 1-byte ports on each 4KB page and expands the space to 64MB, with the ports being at a relative offset specified by port{11:0} within each 4KB virtual page. When executing an IA-32 IN or OUT instruction a processor based on the Itanium architecture automatically converts the IA-32 address to the appropriate expanded I/O port space address.

As a result of the spreading-out of the I/O ports into individual 4KB pages, Itanium-based operating system code can control IA-32 IN, OUT instruction and IA-32 or Itanium load/store accessibility to blocks of 4 virtual I/O ports using the TLBs. This allows Itanium-based operating systems to securely map devices that inhabit the I/O port space to different Itanium-based device drivers or to user-space Itanium-based applications.

Itanium-based operating systems must ensure that the I/O port space is always mapped as uncacheable memory, and that Itanium-based software only issues aligned 1, 2 or 4 byte references to I/O port space, otherwise device behavior is undefined.

When porting an IA-32 device driver to the Itanium architecture it can be useful to emulate the behavior of IA-32 IN and OUT instructions. The following code examples should be used for this purpose, since they enforce the strict memory ordering and platform acceptance requirements that IA-32 IN and OUT instructions are subject to. The following Itanium-based assembly code outb (out byte) and inb (in byte) examples assume that the io_port_base is the virtual address mapping pointer set up by the IA_64 operating system. An `mf.a` instruction is used to verify acceptance by the platform before returning to the calling routine. Interrupts would expected to be disabled if these routines are called from user mode. This is for possible issues with process migration after servicing an interrupt.

```
//
// void outb(unsigned char *io_port,unsigned char byte)
//
//Output a byte to an I/O port.
//
ENTRY(outb)
    base_addr = r16
    port_addr = r17
    port_offset = r18
    mask = r19

    alloc   r13 = ar.pfs, 2, 0, 0, 0        // 2 in, 0 local, 0 out, 0 rot
    movl    base_addr = io_port_base
    extr.u  port_offset = in0, 2, 14
    mov     mask = 0xfff
    ;;
    ld8     port_addr = [base_addr]
    shl     port_offset = port_offset, 12
    and     in0 = mask, in0
    ;;
    add     port_offset = port_offset, in0
    ;;
    mf
    add     port_addr = port_addr, port_offset
    ;;
    st1.rel [port_addr] = in1
    mf.a
    mf
    br.ret.spnt.few rp
END(outb)

//
// unsigned char inb(unsigned char *io_port)
//
// Input a byte from an I/O port.
//
ENTRY(inb)
    base_addr = r16
    port_addr = r17
    port_offset = r18
    mask = r19

    alloc   r13 = ar.pfs, 2, 0, 0, 0        // 2 in, 0 local, 0 out, 0 rot
    movl    base_addr = io_port_base
    extr.u  port_offset = in0, 2, 14
    mov     mask = 0xfff
    ;;
    ld8     port_addr = [base_addr]
    shl     port_offset = port_offset, 12
    and     in0 = mask, in0
    ;;
    add     port_offset = port_offset, in0
    ;;
    mf
    add     port_addr = port_addr, port_offset
    ;;
    ld1.acq r8 = [port_addr]
    mf.a
    mf
    br.ret.spnt.few rp
END(inb)
```

Performance Monitoring Support 12

Processors based on the Itanium architecture include a minimum of four performance counters which can be programmed to count processor events. These event counts can be used to analyze both hardware and software performance. Performance counters can be configured to generate a counter overflow interrupt. This interrupt can be used for event or time based profiling. For hot-spot analysis of running code, performance monitor interrupts can be used to create a profile of frequently occurring instruction pointers (IP). Another common use of event counts is to compute processor performance metrics such as cycles per instructions (CPI), the current branch, cache or TLB miss rates, etc.

The Itanium architecture provides architected support for context switching of performance monitors by an Itanium-based operating system. If supported by the operating system, this allows performance counter events to be broken down per thread or per process which is important for effective performance tuning of Itanium-based applications.

The remainder of this chapter reviews the architected performance monitoring mechanisms. It also discusses the Itanium-based operating system support needed for two monitoring usage models: per process/thread and system-wide event monitoring.

12.1 Architected Performance Monitoring Mechanisms

As defined in Section 7.2, "Performance Monitoring", processors based on the Itanium architecture provide a minimum of four generic performance counter pairs (PMC/PMD[4..7]). The performance monitor control (PMC) registers are used to select the event to be counted, and to define under what conditions the event should qualify for being counted (for details refer to Section 7.2.1, "Generic Performance Counter Registers"). The performance monitor data (PMD) registers contain the event count or data.

The PMC/PMD registers can only be written by privileged software (PSR.cpl must be zero). A counter can be configured as a “privileged” counter or a “user-level” counter by setting of the PMC[i].pm bit. Privileged counters can only read at privilege level 0, while user-level counters can by read by user mode code (unless the operating system has explicitly disabled the user-level monitor reads using PSR.sp).

Once the PMC/PMD registers have been configured, counting is enabled and disabled by setting bits in the PSR. User-level counters can be controlled at user-level using the rum and sum instructions to toggle PSR.up. Privileged counters are controlled by privileged software using the rsm, ssm, mov from/to PSR instructions to toggle PSR.pp. Counting for all counters is further controlled by the PMC[0] freeze bit. When PMC[0].fr is 0, all counters are disabled. When PMC[0].fr is 1, counting is enabled based on PMC[i].pm, PSR.pp and PSR.up. For more details on controlling of the performance monitors please refer to Section 7.2.1, "Generic Performance Counter Registers".

The PAL firmware provides information about the performance monitor registers that are implemented on the processor through the PAL_PERF_MON_INFO PAL call. Information provided by the PAL includes bit masks which indicate which PMC/PMD registers are implemented on this processor model, as well as the implemented number of generic PMC/PMD pairs, and the counter width of the generic counters.

12.2 Operating System Support

The monitoring mechanisms discussed in the previous section support two performance monitoring usage models that need support from an Itanium-based operating system.

- Per Thread/Process Event Monitoring

  To monitor processor events per thread the operating system needs to save and restore performance monitor state at thread/process context switches. This save/restore of PMC and PMD registers only needs to be done for monitored threads. The effect of the save/restore is that when a monitored thread is running, PMD reads will reflect events for the monitored thread/process only. Section 7.2.4.2, "Performance Monitor Context Switch" defines the steps required for per-thread context switch of performance monitors. It is worth noting that the PMC/PMD masks returned from PAL_PERF_MON_INFO indicate which PMC/PMD registers are implemented. The context switch routine can use the mask to save/restore implemented monitors without knowing the function of the monitors.
- System Wide Event Monitoring

  To monitor processor events system wide (across all processes and the operating system kernel itself), a monitor must be enabled continuously across all contexts. This can be achieved by configuring a privileged monitor (PMC.pm=1), and by ensuring that PSR.pp and DCR.pp remain set for the duration of the monitor session. Since the operating system typically reloads PSR and possibly DCR on context switch, this requires the operating system to set PSR.pp and DCR.pp for all contexts that are active during the monitoring session. One way to accomplish this is to have code in the context switch routine to always set PSR.pp and DCR.pp when system wide monitoring is in effect. Another technique is to set the initial state for all new threads/processes to PSR.pp=1, PSR.up=0, PSR.sp=0 and DCR.pp=1. Setting the per thread PSR and DCR in this way ensures that privileged monitors will be enabled across all contexts. When system wide monitoring is in effect, PSR.pp, DCR.pp as well as the PMC and PMD registers should not be altered by the context switch routine.

To support both per thread and system wide monitoring, the operating system needs to be aware which type of monitoring is being performed at any given moment. If per thread/process monitoring is active, then the operating system must save/restore monitor state for monitored threads. If system wide monitoring is active, then the operating system must ensure that PSR.pp and DCR.pp remain set.

The preferred approach for performance monitoring is for Itanium-based operating systems to provide a set of kernel mode services that allow performance monitoring software to be implemented in a loadable device driver. Such a loadable device driver can support various usage monitoring models, can be adapted to model-specific processor monitoring capabilities, and is a well-defined isolated and easily replaceable software component. The following operating system services allow a kernel mode device driver to take full advantage of the performance monitors:

- Allocation/Free Performance monitors – operating system should delegate management of the performance monitor resources to device driver.
- Process create/terminate notification – operating system should notify driver on process create/ terminate.
- Thread create/terminate notification – operating system should notify driver on thread create/ terminate.
- Context switch notification – operating system should notify driver on thread and process context switch. The driver will perform the required save/restore depending on the currently active usage model.
- Performance counter overflow interrupt – operating system should notify driver when a performance monitor overflow interrupt occurs.

- Get Current Process Identifier – returns a unique identifier for the current process or address space. This should be callable in any context, e.g. by an interrupt handler.
- Get Current Thread Identifier – returns a unique identifier for the current thread of execution. This should be callable in any context, e.g. by an interrupt handler.

One of the challenges when doing instruction pointer (IP) profiling is to relate the current IP to an executable binary module and to an instruction within that module. If appropriate symbol information is available, the IP can be mapped to a line of source code.

To support this IP to module mapping, it is recommended that the OS provide services to enumerate all kernel and user mode modules in memory, and to allow a kernel mode driver to be notified of each module load. The following services are recommended:

- Enumerate kernel mode modules – provides information each kernel mode module currently loaded in memory.
- Enumerate threads/processes – provides a list of current threads/processes. The list should include the unique identifier for each thread/process.
- Enumerate all user mode modules – provides information on each user mode module that is currently loaded in memory (all processes).
- Enumerate modules for a process – provides information on each user mode module that is currently loaded in memory for the selected process.
- Module load notification – OS should notify a driver when the OS loads a kernel or user mode module into memory for execution. The notification should occur before the module begins execution.

In the above services for module enumeration and load notification, the module information provided for a module should include module name, load address, size in bytes, section number (if a section of a module is loaded non-contiguously), and a process/thread identifier that identifies the process into which the module is loaded.

Firmware Overview 13

The Itanium architecture defines three firmware layers: Processor Abstraction Layer (PAL), System Abstraction Layer (SAL), and Extensible Firmware Interface (EFI).

The PAL, SAL and EFI layers work together to handle the reset abort event. The reset abort handling performs processor and system initialization for operating system (OS) boot and provides a legacy-free API to the operating system loader. The PAL and SAL firmware layers work together to handle machine check aborts (MCA), initialization events (INIT), and platform management interrupt (PMI) handling. All three firmware layers also provide runtime procedure calls to abstract processor and platform functions that may vary across implementations.

This chapter will provide an overview of the firmware layers and how the firmware layers interact with each other as well as with the operating system. For the full architecture specifications of the PAL firmware please refer to Section 11, "Processor Abstraction Layer". For full architecture specifications on SAL and EFI firmware layers please refer to Section 1.2, “Related Documents” on page 2:373.

The PAL layer is developed by Intel Corporation and delivered with the processor. The SAL and EFI firmware is developed by the platform manufacturer and provide a means of supporting value added platform features from different vendors.

The interaction of the various functional firmware blocks with the processor, platform and operating system is shown in Figure 13-1, “Firmware Model” on page 2:482.

13.1 Processor Boot Flow Overview

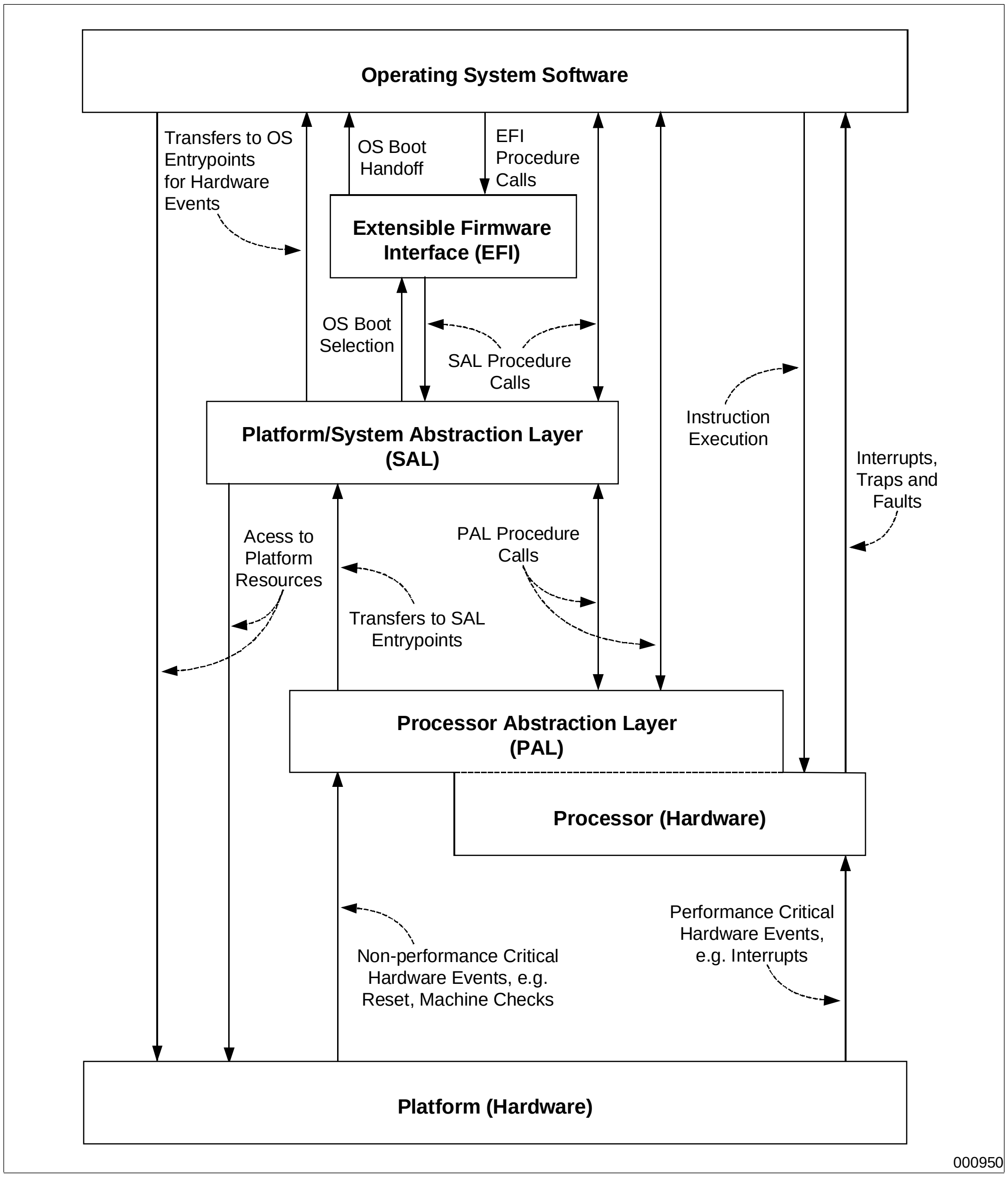
13.1.1 Firmware Boot Flow

Upon detection of a reset event on a processor based on the Itanium architecture, execution begins at an architected entry point inside of PAL. This PAL code will verify the integrity of the PAL code and may perform some basic processor testing. PAL will then branch to an entry point within the SAL firmware. This first branch to SAL is to determine if a firmware update is needed requiring re-programming of the firmware code. If no firmware update is needed SAL will branch back to PAL.

PAL now performs additional processor testing and initialization. These first processor tests are performed without platform memory. PAL indicates the outcome of the testing and branches to an entry point within SAL firmware for the second time. SAL will now begin platform testing and initialization.

The order of steps within the SAL firmware is platform implementation dependent and may vary. In general, the SAL firmware selects a Bootstrap processor (BSP) in multi-processor (MP) configurations early in the boot sequence. Next, SAL will find and initialize memory and invoke PAL procedures to conduct additional processor tests to ensure the health of the processors. SAL then initializes the system fabric and platform devices. SAL will display the progress of the boot on the video output device and permit the user to change the system configuration.

Figure 13-1. Firmware Model


Operating System Software
Transfers to OS Entrypoints for Hardware Events
OS Boot Handoff
EFI Procedure Calls
Extensible Firmware Interface (EFI)
OS Boot Selection
SAL Procedure Calls
Instruction Execution
Platform/System Abstraction Layer (SAL)
Interrupts, Traps and Faults
Acess to Platform Resources
PAL Procedure Calls
Transfers to SAL Entrypoints
Processor Abstraction Layer (PAL)
Processor (Hardware)
Performance Critical Hardware Events, e.g. Interrupts
Non-performance Critical Hardware Events, e.g. Reset, Machine Checks
Platform (Hardware)
000950

The SAL firmware layer hands off control to the EFI firmware layer which incorporates a Boot Manager. The EFI firmware specification [EFI] enables booting from a variety of mass storage devices such as hard disk, CD, DVD as well as remote boot via a network. At a minimum, one of the mass storage devices contains an EFI system partition.

The EFI Boot Manager displays the list of operating system choices and permits the user to select the operating system for booting. To support this functionality, the OS setup program stores the boot paths of the OS loaders and boot options in non-volatile storage managed by the EFI. The EFI reserves the environment variables Boot#### (#### represents values 0000 to 0xFFFF) for this purpose. The OS setup program must also store the OS loader binary images within the EFI System Partition. The EFI Boot Manager will also allow the user to add boot options, delete boot options, launch an EFI application, and set the auto-boot time out value.

The EFI System Partition also contains EFI drivers that will be loaded by the EFI firmware prior to transfer of control to an OS loader. The floating-point software assist (FPSWA) library is included in these EFI drivers. The FPSWA library may be invoked by the OS during floating-point exception faults and traps. Please see Section 8.1.1, "The Software Assistance Exceptions (Faults and Traps)" on page 2:451 for more information on the usage of this library.

If the user elects to boot an IA-32 operating system, the EFI will load 512 bytes of the first level boot code (Master Boot Record in the case of disk devices) at location 0x7C00. Next, EFI will remove its memory footprint and returns to the SAL firmware. The SAL will then invoke a PAL procedure to set up the IA-32 System environment and jump to the boot code at 0x7C00. The boot code will load an IA-32 OS loader which, in turn, loads and transfers control to the IA-32 OS kernel.

If the user elects to boot an Itanium-based operating system, the EFI loads the appropriate OS loader from the EFI System Partition and passes control to it. The OS loader will load other files including the OS kernel from an OS partition using the EFI boot services which provides a legacy free API interface to the OS loader. EFI uses SAL to access low level platform resources. The interfaces between EFI and SAL are platform firmware implementation dependent and not relevant for the OS loader developers.

The OS loader can obtain information about the memory map usage of the firmware by making the EFI procedure call GetMemoryMap(). This procedure provides information related to the size and attributes of the memory regions currently used by firmware.

The OS loader will then jump to the OS kernel that takes control of the system. Until this point, SAL retained control of key system resources such as the Interrupt Vector Table and provided the necessary interrupt, trap and fault handlers.

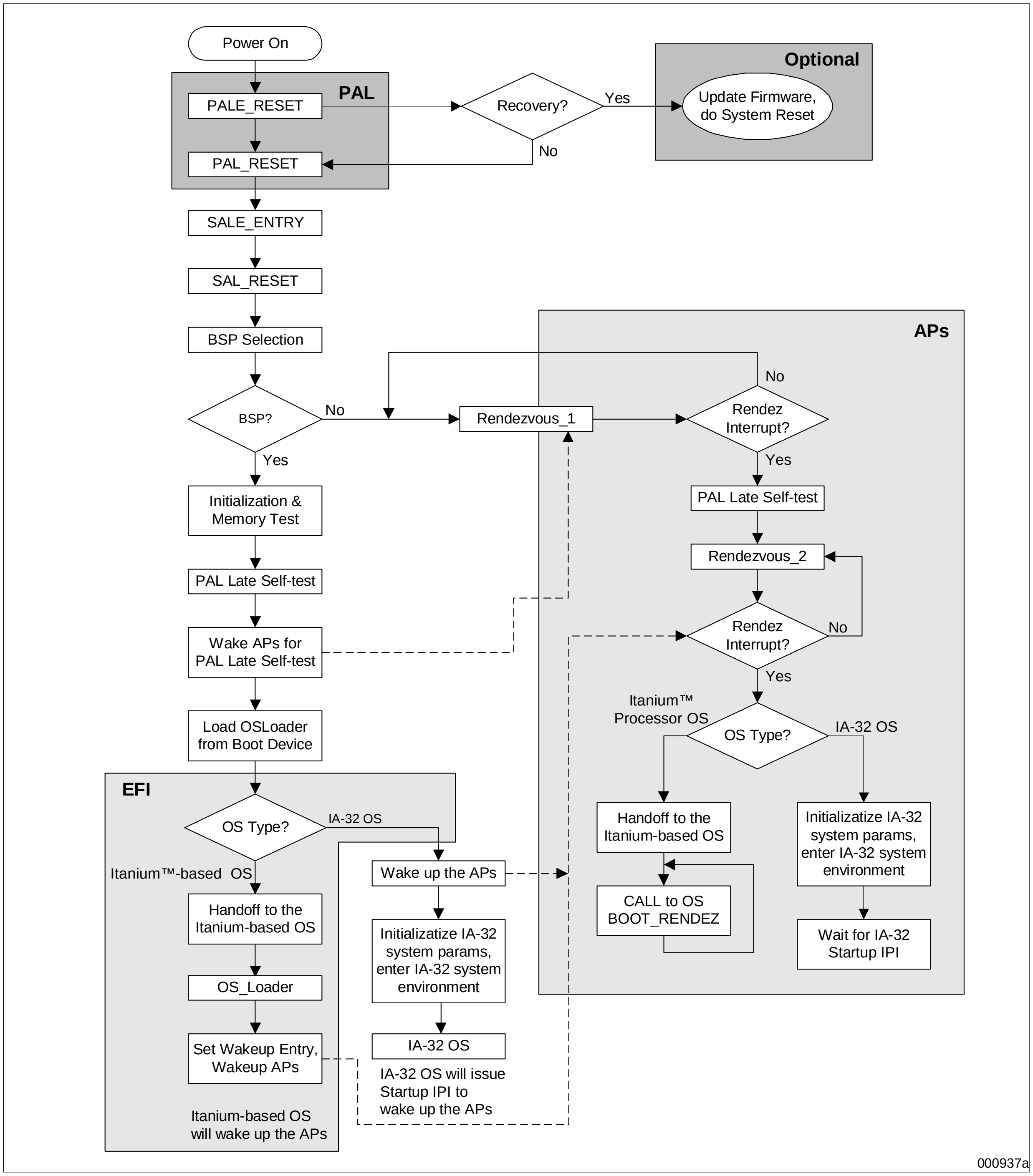
Figure 13-2, "Control Flow of Boot Process in a Multi-processor Configuration" on page 2:484 depicts the booting steps in a MP configuration.

13.1.2 Operating System Boot Steps

The firmware will initialize the processor(s) and platform to a specific state before handing off to the operating system boot loader. The boot loader is then responsible for copying the operating system from some storage medium into memory for running. Once this is done the operating system will need to initialize some key registers before entering into a higher level language code such as C. This section will describe code that an OS will need to execute in order to initialize system registers for preparing an OS to run in virtual mode and handle interrupts. Appendix A, "Code Examples" provides the Itanium-based sample assembly code described in this section.

Assuming the specific operating system boot loader hands off to the OS kernel in physical mode, the operating system should first disable interrupts and interrupt collection via the PSR. This is done to avoid taking external interrupts from timers, etc and also prepares for writing specific system registers that require PSR.ic to be 0 when written.

Figure 13-2. Control Flow of Boot Process in a Multi-processor Configuration



Power On
PAL
PALE_RESET
PAL_RESET
Recovery?
Yes
No
Optional
Update Firmware, do System Reset
SALE_ENTRY
SAL_RESET
BSP Selection
APs
BSP?
No
Yes
Rendezvous_1
No
Rendez Interrupt?
Yes
PAL Late Self-test
Initialization & Memory Test
PAL Late Self-test
Rendezvous_2
Wake APs for PAL Late Self-test
Rendez Interrupt?
No
Yes
Itanium™ Processor OS
OS Type?
IA-32 OS
Load OSLoader from Boot Device
EFI
OS Type?
IA-32 OS
Itanium™-based OS
Handoff to the Itanium-based OS
Initializatize IA-32 system params, enter IA-32 system environment
Wake up the APs
Handoff to the Itanium-based OS
CALL to OS BOOT_RENDEZ
Initializatize IA-32 system params, enter IA-32 system environment
Wait for IA-32 Startup IPI
OS_Loader
IA-32 OS
Set Wakeup Entry, Wakeup APs
IA-32 OS will issue Startup IPI to wake up the APs
Itanium-based OS will wake up the APs
000937a

Next the operating system startup code invalidates the ALAT via the `invala` instruction. The invala in complete form will invalidate all entries in the ALAT.

The register stack should be invalidated. This can be done by setting the Register Stack Configuration Register (RSC) to zero followed by a loadrs instruction. Setting the RSC to zero will put the register stack in enforced lazy mode and set the RSC.loadrs, load distance to tear point, to zero. The loadrs will invalidate all stacked registers outside current frame.

The region registers and protection key registers are then initialized with operating system implementation dependent values. For example, the OS will initialize the region register with a preferred page size. It would also disable the VHPT until it was ready for it. In the example, all region registers are initialized with an 8-KB page size.

An OS must setup a kernel stack pointer and backing store pointer for the register stack. The stack pointer (GR12) is set to the OS kernel stack area with scratch space to cover calling conventions. AR.RSC must be set to enforced lazy mode before writing to the bspstore register. Initializing the bspstore has effects on all 3 RSE pointers (BSP, BSPSTORE, and RSE.BspLoad).

In order for the operating systems to handle interruptions, the operating system interrupt vector table base address must be set up. The size of the vector table is 32K bytes and is 32K byte aligned. Setting the location of the table is accomplished by moving the address into CR.IVA.

Operating systems setup system address translations for the kernel text and data by using the translation insertion format described in Section 4.1.1.5, "Translation Insertion Format". A combination of a general register, Interruption TLB Insertion Register (ITIR), and the Interruption Faulting Address register (IFA) are used to insert entries into the TLB. To void TLB faults on specific text and data areas the operating system can lock critical virtual memory translations in the TLB by use of Translation Register (TR) section of the TLB. The entries are placed into a TR via the Insert Translation Register (itr) instruction. The translation will remain unless the software issues the Purge Translation (ptr) instruction. Other important areas might be locked also, such as entries for memory mapped I/O, etc.

After the initial translations have been entered, the OS can make final preparations for enabling virtual addressing. The OS needs to set several important bits in the IPSR, such as data address translation (dt), register stack translation (rt), instruction address translation (it), enabling interruption collection (ic), and setting the specific register bank (bn).

The Default Control Register (DCR) specifies the default parameters for PSR values on interruption, some additional global controls, and whether speculative load faults can be deferred. The example defers all speculation faults. Also, if the operating system is utilizing the performance monitors then the DCR.pp bit should be set so that on interruption the PSR.pp bit will be set.

The global pointer (GR1) should point to the global data area. It must be setup properly before using higher level languages such as C. The startup code should also set the following registers to zero, the Interruption Function State (CR.IFS, to set frame marker to zero), and AR.RNAT (to make sure no NaT bits are set before OS kernel begins using the RSE.

Before enabling virtual addressing, the Interruption Instruction Bundle Pointer (IIP) is set to point a virtual address. This is done so when the return from interruption instruction (`rfi`) is executed the instruction fetched will have a virtual address. The rfi will switch modes based on IPSR values which are moved into the PSR. The IIP value becomes the new IP.

13.2 Runtime Procedure Calls

The PAL, SAL, and EFI firmware layers provide entry points as runtime interfaces to the OS. These runtime interfaces allow the OS to obtain information about the processor and platform as well as perform implementation specific functions on the processor and platform.

The calling conventions for these runtime procedures are documented in the respective firmware architecture specifications. In general the first input argument to the procedure call specifies the index of the procedure within the list of supported procedures for each firmware layer.

13.2.1 PAL Procedure Calls

PAL procedure calls are classified into two types: static and stacked. The static calls are intended for boot-time use before main memory is available or in error recovery situations where memory or the RSE may not be reliable. All parameters will be passed in the general registers GR28 to GR31 of Bank 1. The stacked registers (GR32 to GR127) will not be used for these calls. The static calls can be called at both boot-time and runtime.

Stacked register calls are intended for use after memory has been made available. The stacked registers are used for parameter passing and local variable allocation. These calls also allow memory pointers may be passed as arguments. These calls can be made at boot-time after memory has been tested and initialized as well as runtime.

For a listing of all the PAL procedures and their classification please see Section 11.9.1, "PAL Procedure Summary".

All PAL calls are re-entrant and can be executed simultaneously on multiple processors.

13.2.1.1 Making a Static PAL Call

Since the static PAL calls do not use stacked registers, these calls are made as a pure jump with branch register B0 containing the address of the bundle to which control will return. The following code example describes how to make a static PAL call:

```
GetFeaturesCall:

             mov     r14 = ip                     // Get the ip of the current
bundle
             movl    r28 = PAL_PROC_GET_FEATURES// Index of the PAL procedure
             movl    r4 = AddressOfPALProc;;      // Address of the PAL proc entry
point
             ld8     r4 = [r4];;                  // Read address from local
pointer
             mov     b5 = r4                      // Move address into a branch
register

// Compute the return address in a position independent manner

             addl    r14 = (BackHome - GetFeaturesCall),r14;;
             mov     b0 = r14                     // b0 is the return link
             mov     r29 = r0                     // Initialize rest of input
arguments
             mov     r30 = r0                     // to zero as required by the
             mov     r31 = r0                     // architecture.

             br.sptk b5;;                         // Make the PAL call.
```

The sample code below is position independent and functions in both physical and virtual addressing modes. Since the return address is evaluated by using the runtime instruction pointer (IP value), it will run from any address. This attribute is important for any relocatable code.

The address of the PAL procedure entry point is passed to SAL at the hand-off from PAL to SAL during reset. SAL will pass this information on to the OS during OS boot as well.

13.2.1.2 Making a Stacked PAL Call

A stacked PAL call uses the stacked registers for argument passing and local variable allocation. The stacked PAL calls conform to the calling conventions document [SWC], with the exception that general register GR28 must also contain the function index input argument. The following code example describes how to make a stacked PAL call.

```
             movlr4 = AddressOfPALProc;;       // Address of the PAL proc entry poi
             ld8 r4 = [r4];;                   // Read address from local pointer
             mov b5 = r4                       // Move address into a branch regist

// Make the PAL_HALT_INFO procedure call. PAL_HALT_INFO uses stacked register
// convention and parameters are passed with in0-in3

             mov r28 = PAL_HALT_INFO;;         // Index of the PAL procedure
             mov out0 = r28                    // r28 and in0 must both contain the
                                               // index value for stacked PAL calls
             mov out1 = ScratchMem_Pointer     // Pointer to the memory argument
             mov out2 = 0x0                    // Write zero to unused input
arguments
             mov out3 = 0x0

             br.call.sptk.few b0 = b5;;        // PAL stacked call

// PAL will return here when the call is completed
```

13.2.1.3 PAL Procedure Calls and Performance

PAL procedure calls are designed for a number of different functions varying from boot-time usage before platform memory is available to processor specific functions used during runtime by the OS. PAL runtime procedure calls made by the OS are designed to be flexible with minimal overhead. The following features aid in this goal:

- PAL procedure calls are relocatable. This feature is useful for platforms that have PAL stored in non-volatile storage, such as flash. During OS boot the PAL procedures are copied into RAM which will reduce the memory latency.
- A number of PAL procedure calls are defined to be called in both physical and virtual addressing. This allows the caller to make the call in its currently executing addressing mode, thus reducing the need to switch between physical and virtual addressing.

13.2.2 SAL Procedure Calls

All SAL procedure calls use the stacked register calling convention. SAL follows the floating-point register conventions specified in the calling conventions document [SWC], with the exception that SAL does not use the floating-point registers FR32 to FR127. This exception eliminates the need for the OS to save these registers across SAL procedure calls.

SAL procedures are non re-entrant. The OS is required to enforce single threaded access to the SAL procedures except for the following procedures:

- SAL_MC_RENDEZ, SAL_CACHE_INIT, SAL_CACHE_FLUSH

13.2.3 EFI Procedure Calls

EFI procedure calls are classified into the following two categories: boot services and runtime services. The EFI boot services execute in physical addressing mode only. The runtime services can execute in either physical or virtual addressing mode. The EFI boot services are only available during the boot process and are terminated by a call to the EfiExitBootServices() procedure. After this call, only the SAL and EFI runtime services may be invoked by the OS. The EFI runtime services execute in physical mode until the OS invokes the EFISetVirtualAddress() function to switch the EFI to virtual mode. After this point, the EFI runtime services may be invoked in virtual mode only. For full information on all the EFI boot and runtime services please refer to the EFI specification [EFI].

13.2.4 Physical and Virtual Addressing Mode Considerations

All of the PAL procedures can be called in the physical addressing mode. A subset of PAL calls can be made using the virtual addressing mode. For PAL calls that can be invoked using virtual addressing mode, it is the responsibility of the caller to map these PAL procedures with an ITR as well as either a DTR or DTC. If the caller chooses to map the PAL procedures using a DTC it must be able to handle TLB faults that could occur. See Section 11.9.1, "PAL Procedure Summary" for a summary of all PAL procedures and the calling conventions.

The SAL and the EFI firmware layers have been designed to operate in virtual addressing mode. EFI provides an interface to the OS loader that describes the physical memory addresses used by firmware and indicates whether the virtual address of such areas need to be registered by the OS with EFI. The EFI Specification [EFI] also provides the interfaces for the OS to register the virtual address mappings. In a MP configuration, the virtual addresses registered by the OS must be valid globally on all the processors in the system.

The SAL runtime services may be called either in virtual or physical addressing mode. SAL procedures that execute during machine check, INIT, and PMI handling must be invoked in physical addressing mode.

The parameters passed to the firmware runtime services must be consistent with the addressing environment, i.e. PSR.dt, PSR.rt setting. Additionally, the global pointer (gp) register [SWC] must contain the physical or virtual address for use by the firmware.

13.2.4.1 SAL Procedures that Invoke PAL Procedures

Some of the SAL runtime services, e.g. SAL_CACHE_FLUSH, will need to invoke PAL procedures. While invoking these SAL procedures in virtual mode, the OS must provide the appropriate translation resources required by PAL (i.e. ITR and DTC covering the PAL code area).

In general, if SAL needs to invoke a PAL procedure, it will do so in the same addressing mode in which it was called by the OS (i.e. without changing the PSR.dt, PSR.rt, and PSR.it bits). If a particular PAL procedure can only be invoked in physical mode, SAL will turn off translations and then invoke the PAL procedure. SAL will then restore translations before returning to the caller. The PAL_CACHE_INIT procedure invoked by the SAL_CACHE_INIT is an example of a procedure that would require such an addressing mode transition.

13.3 Event Handling in Firmware

The PAL and SAL firmware layers are responsible for handling three events. These events are the machine check abort (MCA), the initialization event (INIT) and the platform management interrupt (PMI). When the processor detects these events it will pass control to PAL for handling. The following sections describe the high level overview of the firmware handling of these events.

13.3.1 Machine Check Abort (MCA) Flows

In order to have a highly reliable and fault tolerant computing environment a great deal of coordination and cooperation between the system entities (i.e. the processor, platform, and system software) is required. The PAL firmware, the SAL firmware, and the operating system all work together to meet this goal. This section will provide an overview of the machine check abort handling.

When the processor detects an error, control is transferred to the PAL_MCA entrypoint. PAL_MCA will perform error analysis and processor error correction where possible. Subsequently, PAL hands off control to the SAL MCA component. SAL_MCA will perform error logging and platform error correction where possible. Errors that are corrected by the PAL and SAL firmware are logged and control is returned back to the interrupted process/context. For corrected errors, no OS intervention is required for error handling, but the OS is notified of the event for logging purposes through a low priority asynchronous corrected machine check interrupt (CMCI). See Section 5.8.3.8, "Corrected Machine Check Vector (CMCV – CR74)" for more information on the CMCI. If the error was not corrected by firmware, SAL hands off control to the OS_MCA handler.

Within the firmware the entire machine check is handled with virtual address translations disabled. However, the OS machine check handler may optionally enable virtual addressing and execute most of MCA handler in virtual mode.

Figure 13-3 and Figure 13-4 depict an overview of Itanium machine check processing. The control flows are slightly different for corrected and uncorrected machine checks.

Figure 13-3. Correctable Machine Check Code Flow

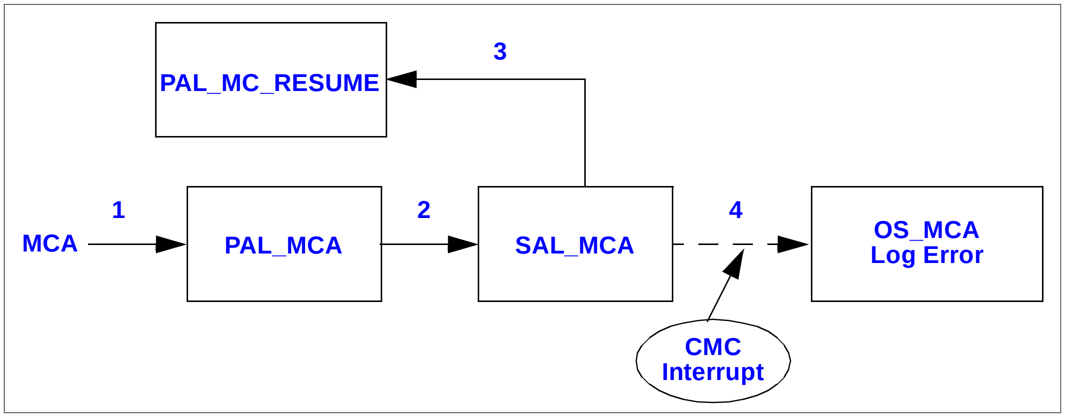
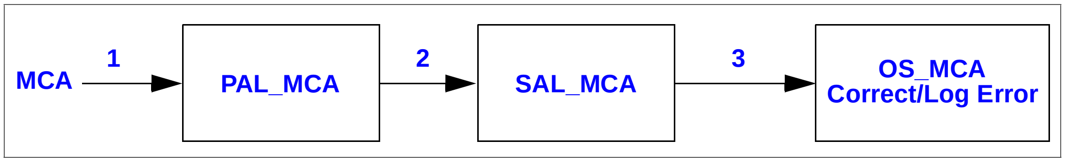


Figure 13-4. Uncorrectable Machine Check Code Flow



For multi-processor systems, machine checks are classified as local and global. A global MCA implies a system wide broadcast by hardware of an error condition. During a global MCA condition, all the processors in the system will be notified of the MCA, detected by one or more system components, and each of the processors in the system will start processing the MCA in their respective handlers. The SAL firmware and OS layers will coordinate the handling of the error among the processors.

A local MCA has a scope of influence that is limited to the particular processor which encountered the error. This local MCA will not be broadcast to other processors in the system and will be handled on an individual processor basis. At any point in time, more than one processor in the system may experience a local MCA and handle it without notifying other processors in the system.

The next sections will provide an overview of the responsibilities that the PAL, SAL and OS have for handling machine checks. These sections are not an exhaustive description of the functionality of the handlers but provides a high level description of how the MCA handling is split among the different components.

13.3.1.1 Machine Check Handling in PAL

All machine check abort events are first handled in the PAL firmware layer. The following provides a brief description of some of the functions of the PAL machine check handler:

- Correct processor errors if possible.
- Attempt to contain the error by requesting a rendezvous for all processors in the system if needed.
- Hand off control to SAL for further processing, such as error logging.
- Return processor error log information upon request by SAL.
- Return to the interrupted context by restoring the state of the processor.
- Notify the OS about corrected machine check conditions through the CMC interrupt.

13.3.1.2 Machine Check Handling in SAL

Before SAL is ready to handle machine checks, it must register with PAL an uncacheable memory buffer that PAL can use to save away processor state. This area is known as the min-state save area. If a machine check occurs before this memory location has been registered, return to the interrupted context is not possible and the machine check is not recoverable.

The following provides a description of some of the functions of the SAL machine check handler.

- Attempt to rendezvous the other processors in the system on a PAL request.
- Process MCA handling after handoff from PAL.
- Retrieve processor error log information via PAL procedure calls and store this information for logging purposes.
- Issue a PAL clear log request to clear the processor error logs, which enables further logging.
- Log platform state for MCA and retain it until it is retrieved by the OS.
- Attempt to correct processor machine check errors which are not corrected by PAL.
- Attempt to correct platform machine check errors.
- Branch to the OS MCA handler for uncorrected errors or optionally reset the system.
- Return to the interrupted context via a PAL procedure call.

13.3.1.3 Machine Check Abort Handling in OS

Before the OS kernel is ready to handle machine checks, it must register the address of the OS_MCA entry point and the GP [SWC] value for the OS_MCA handler with SAL. If the OS does not register its entry point, the occurrence of a machine check will cause a system reset. In MP configurations, the OS must also register with SAL:

- A rendezvous interrupt vector which SAL firmware can use to rendezvous the processors.
- The mechanism that the OS will employ to wake up the processors at the end of machine check processing.

When the OS registers the OS_MCA entry point with SAL, it also supplies the length of the code (or at least the length of the first level OS_MCA handler). SAL computes and saves the checksum of this code area. Prior to entering OS_MCA, SAL ensures that the OS_MCA vector is valid by verifying the checksum of the OS_MCA code. Hence, the OS_MCA code must not contain any self modifying code.

When an uncorrected machine check event occurs, SAL will invoke the OS_MCA handler. The functionality of this handler is dependent on the OS. At a minimum, it must call a SAL procedure to retrieve the error logging and state information and then call another SAL procedure to release these resources for future error logging and state save.

When the OS_MCA code completes, it decides whether or not to return to the interrupted context. The OS must take into account the state information retrieved from the SAL with respect to the continuability of the processor and system. Thus, even if the OS could correct the error, if PAL or SAL reports that it did not capture the entire processor context, resumption of the interrupted context will not be possible.

The OS must also determine from values stored by PAL in the min-state save area whether the machine check occurred while operating with PSR.ic set to 0 and whether the processor supports recovery for this case. Please refer to Section 11.3.1.1, "Resources Required for Machine Check and Initialization Event Recovery" for more information on processor recovery under this condition.

To provide better software error handling, some operating systems build mechanisms to identify whether machine checks occurred during execution of the OS kernel code or in the application context. One technique to achieve this is to call the PAL_MC_DRAIN procedure when an application makes a system call to the OS. This procedure completes all outstanding transactions within the processor and reports any pending machine checks. This technique impacts system call and interrupt handling performance significantly, but will improve system reliability by allowing the OS to recover from more errors than if this mechanism was not included.

13.3.2 INIT Flows

INIT is an initialization event generated by the platform or by software through an inter-processor interrupt message. The INIT can be due to a platform INIT event or due to a failed rendezvous on an application processor.

The INIT event will pass control to the PAL firmware INIT handler. The PAL INIT handler saves processor state to the registered min-state save area and sets up the architected hand off state before branching to SAL. See Section 11.5, "Platform Management Interrupt (PMI)" for more information on the PAL INIT handling.

The SAL INIT handler logs processor state and platform state information and then calls the OS_INIT handler if one is registered. The OS_INIT handler gains control in physical mode but may switch to virtual mode if necessary. The OS may choose to implement a crash dump or an interactive debugger within the OS_INIT handler.

The OS must register the OS_INIT entry point with SAL, otherwise the occurrence of an INIT event will cause a system reset. At the end of OS_INIT handling, the OS must return to SAL with the appropriate exit status.

Figure 13-5 illustrates the flow of control during INIT processing.

13.3.3 PMI Flows

Processors based on the Itanium architecture implement the Platform Management Interrupt (PMI) to enable platform developers to provide high level system functions, such as power management and security, in a manner that is transparent not only to the application software but also to the operating system.

When the processor detects a PMI event it will transfer control to the registered PAL PMI entrypoint. PAL will set up the hand off state which includes the vector information for the PMI and hand off control to the registered SAL PMI handler. To reduce the PMI overhead time, the PAL PMI handler will not save any processor architectural state to memory. Please see Section 11.5, "Platform Management Interrupt (PMI)" for more information on PAL PMI handling.

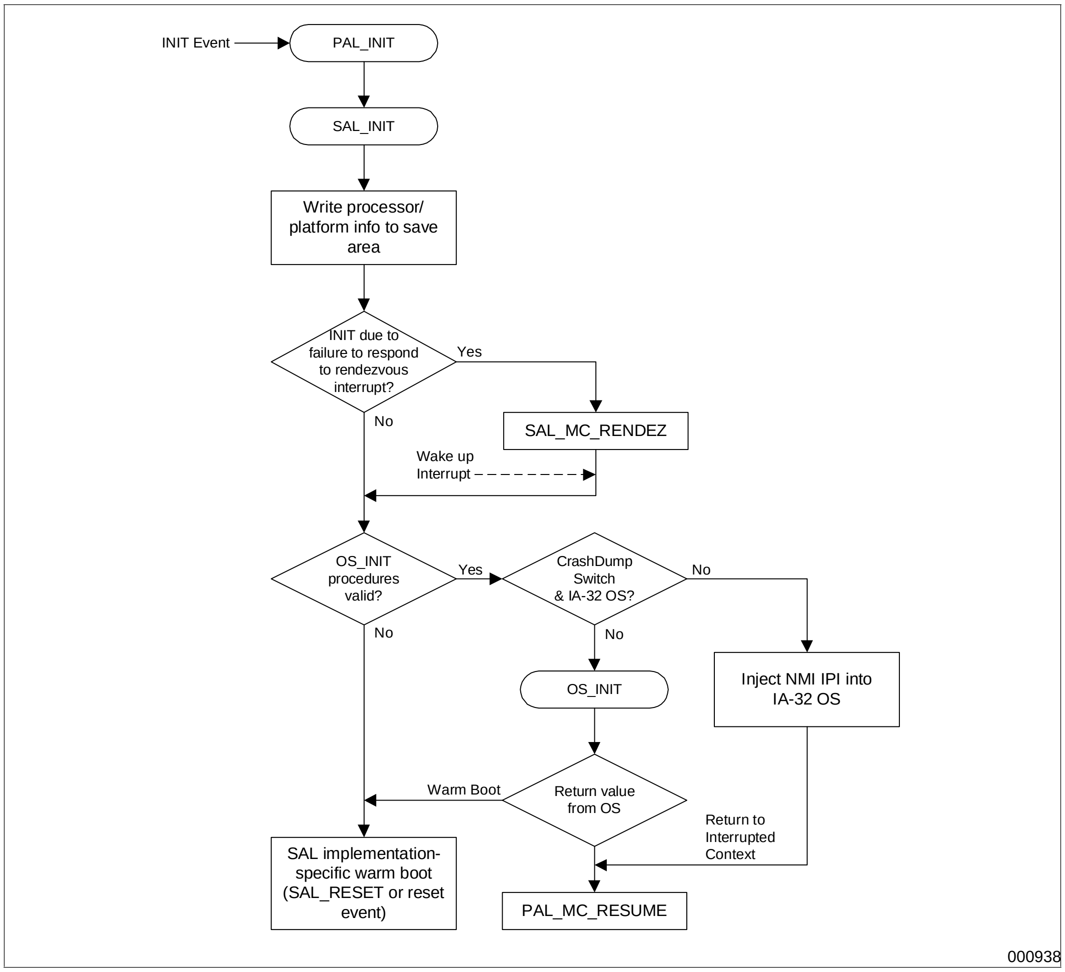
The SAL PMI handler may choose to save some additional register state to SAL allocated memory to handle the specific platform event that generated the PMI.

The OS will not see the PMI events generated by the platform. The platform developer can use PMI interrupts to provide features to differentiate their platform.

PMI handling was designed to be executed with minimal overhead. The SAL firmware code copies the PAL and SAL PMI handlers to RAM during system reset and registers these entry-points with the processor. This code is then run with the cacheable memory attribute to improve performance.

There is no special hardware protection of the PMI code’s memory area in RAM. The protection of this code space is through the OS memory management’s paging mechanism. SAL sets the correct attributes for this memory space and passes this information to the OS through the EFI System table entries [EFI].

Figure 13-5. INIT Flow


INIT Event
PAL_INIT
SAL_INIT
Write processor/ platform info to save area
INIT due to failure to respond to rendezvous interrupt?
Yes
No
SAL_MC_RENDEZ
Wake up Interrupt
OS_INIT procedures valid?
Yes
No
CrashDump Switch & IA-32 OS?
No
No
OS_INIT
Inject NMI IPI into IA-32 OS
Warm Boot
Return value from OS
Return to Interrupted Context
SAL implementation-specific warm boot (SAL_RESET or reset event)
PAL_MC_RESUME
000938

Code Examples A

A.1 OS Boot Flow Sample Code

The sample code given below is a example of setting up operating system register state to prepare the processor for running in virtual mode as described in Section 13.1.2, “Operating System Boot Steps” on page 2:483.

```
// This code will perform the following steps:
//              1.  Initialize PSR with interrupt disabled (bit 13)
//              2.  Invalidate ALAT via invala instruction
//              3.  Invalidate register stack
//              4.  Set region registers rr[r0] - rr[r7] to RID=0, PS=8K, E=0.
//              5.  Disable the VHPT
//              6.  Initialize protection key registers
//              7.  Initialize SP
//              8.  Initialize BSP
//              9.  Enable register stack engine.
//              10. Setup IVA
//              11. Setup virtual->physical address translation
//              12. Setup GP.

                .file“start.s”

// globals

        .global main
        .type main, @function                   // C function we will return to

                   .global __GLOB_DATA_PTR      // External pointer to Global Data area
                   .global IVT_BASE             // External pointer to IVT_BASE

        .text

// This is the entry point where primary boot loader
// passes control.

pstart::

                   mov     psr.l = r0           // Initialize psr.l
                   ;;
                   invala                       // Invalidate ALAT
                   mov     ar.rsc = r0          // Invalidate register stack
                   ;;
                   loadrs

// Initialize Region Registers

                   mov     r2 = (13 << 2)       // 8K page size
                   mov     r3 = r0
                   mov     r4 = 61
                   ;;

Loader_RRLoop:
                   shl     r10 = r3, r4
                   ;;
                   mov     rr[r10] = r2
                   add     r3 = 1, r3
                   ;;
                   cmp4.geu p6, p7 = 8, r3
(p6)               br.cond.sptk.few.clr Loader_RRLoop
                   ;;

// Disable the VHPT walker and set up the minimum size for it (32K) by writing
// to the page table address register (cr.pta)

                   mov r2 = (15<<2)
                       ;;
                   mov cr.pta = r2

// Initialize the protection key registers for kernel

                   mov r2 = (1<< 0)
                   mov r3 = r0
                   ;;
                   mov pkr[r3] = r2                  // validate pkr[zero]
                   ;;
                   mov r2 = r0
                   ;;

pkr_loop:
                   add  r3=r3,r0, 1                  // start with index 1
                   ;;
                   cmp.gtu p6,p7 = 8,r3
                   ;;
(p6)               mov pkr[r3] = r2
(p6)               br.cond.sptk.few.clr pkr_loop     // loop until 8

// Setup kernel stack pointer (r12)

                   movl    sp = kstack + (64*1024)   // 64K stack
                   ;;

// Set up the scratch area on stack

                   add     sp = - 32, sp

// Setup the Register stack backing store
//
// 1st deal with Register Stack Configuration register
//
// NOTE: the RSC mode must be enforced lazy (00) to write to bspstore
//
// mode: = enforced lazy
// be = little endian

                   mov  ar.rsc = r0
                   ;;

//             Now have to setup the RSE backing store pointer
//
//             NOTE: initializing the bspstore has effects on all 3 RSE pointers
//                 (BSP, BSPSTORE, and RSE.BspLoad)

                   movl r2 = kstack + ((96 + (96/63))*8)
                   ;;
                   mov  ar.bspstore = r2

//             Need to setup base address for interrupt vector table...

                   movl r3 = IVT_BASE
                   ;;
                   mov  cr.iva = r3
```

```
//             Setup system address translation for the kernel
//
//      The Translation Insertion Format looks like the following...
//
//      Below is the register interface to insert entries into the TLB
//
//           1) A general register contains an address,attributes,and permissions
//           2) ITIR: additional info such as protection key page size info
//           3) IFA: specifies the virtual page number for instruction and data
//              TLB inserts
//
//             Registers used:
//             ---------------
//                     | 63 53     | 52 | 51 50 | 49 12 | 11 9              | 8 7 | 6 | 5 |4 1| 0 |
//             GR      |  ig | ed |  rv              |  ppn | ar  | pl | d | a | ma | p |
//
// ITIR                | rv {63:32} | key {31:8} | ps {7:2} | rv {1:0}|
//
//             IFA     | vpn {63:12}| ignored {11:0} |
//
//             RR[vrn] | reserved{63:32} | rid {31:8}| ignored {7:2) | rv{1} | ignored {0}|
//
//
//             where
//               ig  = ignored bits
//               rv  = reserved bits
//               p   = present bit
//               ma  = memory attribute
//               a   = accessed bit
//               d   = dirty bit
//               pl  = privilege level
//               ar  = access rights
//               ppn = physical page number
//               ed  = exception deferral
//               ps  = page size of mapping (2**ps)
//               vpn = virtual page number
//
// Setup virtual page number
//
// NOTE:             The virtual page number depends on a translation’s
//                   page size.
//
// Add entry for TEXT section

                   movl r2 = 0x0
                   ;;
                   mov  cr.ifa = r2

//             setup ITIR (Interruption TLB Insertion Register)

                   movl r3=( ( 24 << 2 ) | ( 0 << 8 ) ) // set page size to 16 MB
                   ;;
                   mov  cr.itir = r3

//             now setup the general register to use with itr (insert translation
//             register), use physical page of zero

                   movl r10 =((1 << 52 )| ( 0x00000000 << 12 )|( 3 << 9 )|( 0 << 7 )| \
                              (1 <<6 ) | ( 1 << 5 ) | ( 1 << 0 ))
                   mov r11 = r0
                   ;;
                   itr.i itr[r11] = r10      // Insert translation register


//             Entry for OS Data section

                   add r11 = 1, r11              // skip to tr next index
                   movl r2 = 0x0                // use vpn 0
                   ;;
                   mov  cr.ifa = r2

//             Setup ITIR (Interruption TLB Insertion Register)

                   movl r3 = ( ( 24 << 2 ) | ( 0 << 8 ) )   // 16 MB
                   ;;
                   mov  cr.itir = r3

//             Now setup the general register to use with itr (insert translation
//             register)

                   movl r10 =((1 << 52 ) | (0x0 << 12 ) | (3 << 9 ) | (0 << 7) |\
                              (1 << 6) | ( 1 << 5 ) | (1 << 0))
                   ;;
                   itr.d dtr[r11] = r10             // Insert translation register
                   ;;

//             It is now time to set the appropriate bits in the PSR (processor
//             status register)

                   movl r3 = ((1 << 44) | (1 << 36) |(1 << 38) |(1 << 27) |(1 << 17) | \
                              (1 << 15) | (1 << 14) | (1 <<      13))
                   ;;
                   mov cr.ipsr = r3

//             Initialize DCR to defer all speculation faults

                   movl r2 = 0x7f00
                   ;;
                   mov cr.dcr = r2

//             Initialize the global pointer (gp = r1)

                   movl gp = __GLOB_DATA_PTR

// Clear out ifs


                   mov cr.ifs=r0

// Need to do a “rfi” in order to synchronize above instructions and set
// “it” and “ed” bits in the PSR.

                   movl r3 = main               // Setup for main, C code
                   ;;
                   mov cr.iip = r3              // Setup iip to hit main
                   ;;
                   rfi
                   ;;

// Setup kernel stack

               .data
               .globalkstack
               .align 16
kstack:
               .skip(64*1024)
```

Entries in this index are described by the volume number and page or range of pages where the entries can be found. The volume number appears to the left of the colon. The page or range of pages appears to the right of the colon. A range of pages is separated by a hyphen.