Vanderpool Technology for the Intel® Itanium® Architecture (VT-i) Preliminary Specification

Revision 1.0

January 2005

Notice: The Intel® Itanium® 2 architecture processor may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are documented in this specification update.

Document Number: 305942-001

Notice: This document contains information on products in the design phase of development. Do not finalize a design with this information. Revised information will be published when the product is available. Verify with your local Intel sales office that you have the latest datasheet before finalizing a design.

Contents

Tables

1 Revision History

Version	Revision Number	Description	Date
-001	1.0	• Initial release of the document.	January 2005

2 Introduction

This document describes the software interfaces for Itanium® architecture-based processors which support VT-i (Vanderpool Technology for the Intel® Itanium® architecture). These additions allow for the virtualization of processor hardware in order to allow multiple instances of operating systems to be run on a single system. This document is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Note: Virtualization, or Vanderpool Technology, is also supported on IA-32 Intel architecture processors. However the implementation of Vanderpool technology for IA-32 architecture processors is different than VT-i due to many reasons, including the fundamental differences between the IA-32 and Itanium architectures. The IA-32 version of Vanderpool Technology is referred to as VT-x and documentation on VT-x can be found in the *Vanderpool Technology for IA-32 Processors (VT-x) - Preliminary Specification*.

2.1 Affected Documents/Related Documents

Title	Document #
Intel® Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture	245317-004
Intel® Itanium® Architecture Software Developer's Manual, Volume 2: System Architecture	245318-004
Intel® Itanium® Architecture Software Developer's Manual, Volume 3: Instruction Set Reference	245319-004
Intel® Itanium® Architecture Software Developer's Manual, Specification Update	248699-009
Vanderpool Technology for IA-32 Processors (VT-x) - Preliminary Specification	C97063-001

2.2 Virtualization Terminology

The following are terms related to Itanium architecture virtualization:

VT-i – Vanderpool Technology for the Itanium architecture.

VT-x – Vanderpool Technology for the IA-32 architecture.

Virtual Machine Monitor (VMM) – The VMM is the system software which implements software policies to manage/support virtualization of processor and platform resources.

Virtual Processor Descriptor (VPD) – Represents the abstraction of the processor resources of a single virtual processor. The VPD consists of per-virtual-processor control information together with performance-critical architectural state. See Section 5.2.1, "Virtual Processor Descriptor (VPD)" on page 52 for details.

Virtual Processor State – A memory data structure which represents the architectural state of a virtual processor. Part of the virtual processor state is located in the Virtual Processor Descriptor (VPD), and the rest is located in memory data structures maintained by the virtual machine monitor.

PAL intercepts – Interfaces where PAL transfers control to the VMM on virtualization events (execution of virtualized instructions/operations with PSR.vm==1). For details see Section 5.2.3, "PAL Intercepts in Virtual Environment" on page 58.

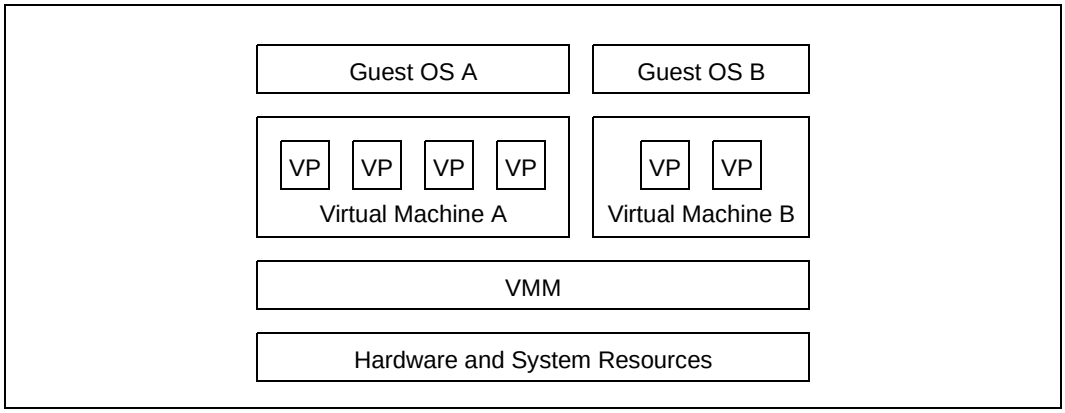
2.3 Virtualization Concept

Modern operating system designs typically assume the operating system has complete and direct control of hardware and system resources. The operating system implements the policies to manage these resources to allow multiple user-level applications to be run. The goal of virtualization is to allow multiple instances of operating systems[1] to be run on a system.

In a typical virtualized environment, there will be a piece of system software responsible for virtualizing the hardware and system resources to allow multiple instances of the operating systems to be run. In the Itanium virtualization architecture, the term **Virtual Machine Monitor** or **VMM** refers to the software component that provides such functionality. The VMM is a piece of host software and is aware of the Itanium virtualization architecture.

For each instance of guest operating system, the VMM will need to create and present a **virtual machine** to the guest operating system. A virtual machine includes all the hardware and system resources (processors, memory, disk, network devices, and other peripherals) expected by the guest operating system. From the VMM perspective, these hardware and system resources are "virtualized". In the Itanium virtualization architecture, a **virtual processor** is a virtualized logical processor. The number of virtual processors created by a VMM in a virtual machine represents the number of logical processors presented to a guest operating system. For example, in Figure 2-1, Guest OS A will see a 4-way system, and Guest OS B will see a 2-way system. There will be at least one virtual processor in a virtual machine. Architecturally there is no limit on the number of virtual machines and virtual processors that can be created by the VMM[2] on a system.

Figure 2-1. Virtual Processor Concept

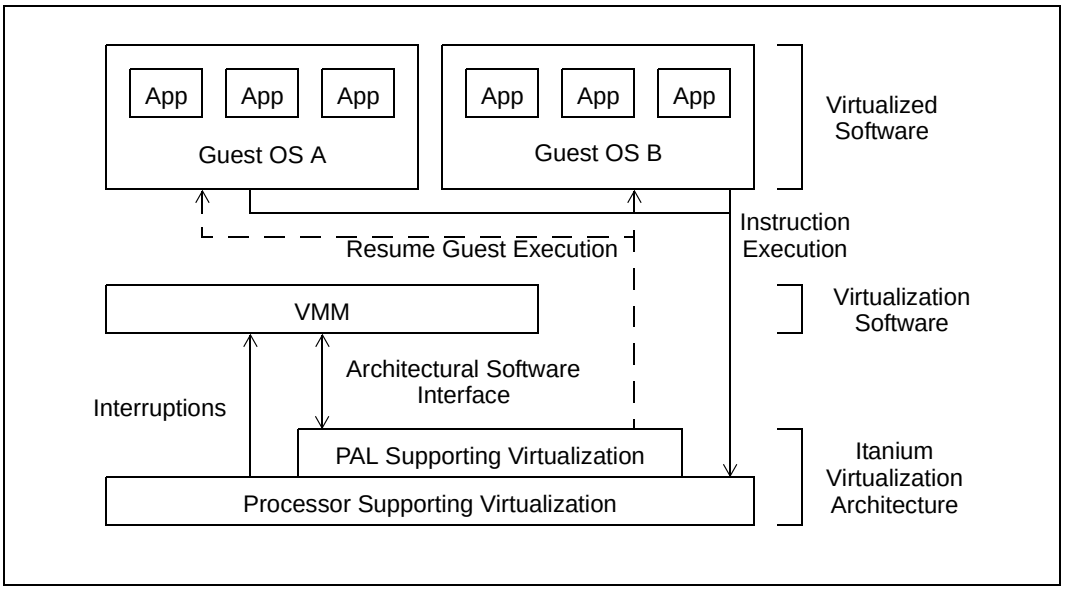


1. The operating systems can be same or different versions, and can come from different operating system vendors.
2. Although there is no architectural limit on the number of virtual machines and virtual processors on a system. There will be limits from the implementation of the hardware and system resources. In addition, there will also be limitations from VMM implementation (e.g., time to perform virtual processor switch).

2.4 Virtualization Environment Overview

The term **virtualization environment** refers to the system environment created by the VMM to run virtualized software[1]. Figure 2-2 shows the main components in a virtualization environment[2], and the interactions between them. A virtualization environment will include one or more processors supporting virtualization, the PAL supporting virtualization, the virtual machine monitor, and virtualized software. The VMM is required to allocate the resources and create the virtualization environment before guest software can be launched. In a virtualization environment, virtualized software will continue to execute on the processor unmodified. Interruptions from the processor will be handled by the VMM. A new architecture interface is defined between the VMM and PAL for access to configuration and optimization options, virtualization services, and virtualization intercept handling.

Figure 2-2. Interactions in a Virtualization Environment



2.5 Resource Virtualization Policies

In a virtualization environment, guest operating systems are running virtualized. For each hardware and system resource on the system, there are typically two policies the VMM can choose to run the virtual processor(s) of the guest operating systems:

- **Shared Policy** – With the shared policy, the actual hardware and system resources will be shared (time multiplexed) between multiple virtual processors. The VMM will need to implement the scheduling/switching/sharing mechanisms to support this policy. For example, in Figure 2-3, logical processor 1 is shared by two virtual processors, and logical processor 2 is shared by the other two virtual processors. In the Itanium virtualization architecture, virtualization accelerations are defined to optimize the usages of this policy. See Section 6.2.4.1, “Virtualization Accelerations” on page 61 in Itanium Architecture Virtualization Specification Update, Rev 2.0 for details.

1. Note that the term virtual machine used in Section 2.3, “Virtualization Concept” on page 8 represents the set of virtual resources presented to a guest operating system. Typically the VMM will create one or more virtual machines in a virtualization environment. The usage model and management policies of the virtualization environment is VMM-specific.
2. This is a simplified diagram to show the major components and their interactions, not all the components are listed (e.g., SAL, EFI…etc.), see Chapter 9, “Firmware Virtualization” for details.

- **Dedicated Policy** – With the dedicated policy, the actual hardware and system resources are dedicated to a particular virtual processor. There will be no sharing of that particular hardware and system resource between virtual processors. The virtual processor will have direct control of the particular hardware and system resource. For example, in Figure 2-4, logical processor 1 is dedicated to virtual processor A, and logical processor 2 is shared by multiple virtual processors. In the Itanium virtualization architecture, virtualization disables are defined to optimize the usages of this policy. See Section 6.2.4.2, "Virtualization Disables" on page 67 in Itanium Architecture Virtualization Specification Update, Rev 2.0 for details.

The VMM decides the resource virtualization policies for the virtual processors at creation time, the policies are applicable until the virtual processor is terminated.

Since the resource virtualization policy is per-resource, the VMM can apply different policies for different resources on a virtual processor basis. For example, on a given virtual processor, the VMM can use a shared policy for an I/O device (i.e., the I/O device is shared between virtual processors), and can use a dedicated policy for the performance counters (i.e., the performance counters on the logical processor is not shared and can be controlled directly by the running virtual processor). In the Itanium virtualization architecture, since there are optimizations defined to support both policies for each resource[1], the VMM cannot apply conflicting optimizations to these resources. The illegal settings are described in each acceleration and disable in Section 6.2.4.1, "Virtualization Accelerations" on page 61 in Itanium Architecture Virtualization Specification Update, Rev 2.0 and Section 6.2.4.2, "Virtualization Disables" on page 67 in Itanium Architecture Virtualization Specification Update, Rev 2.0.

Figure 2-3. Shared Virtualization Policy

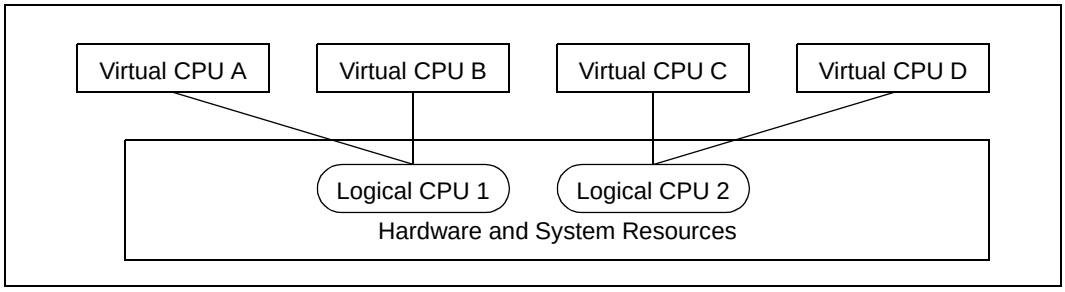
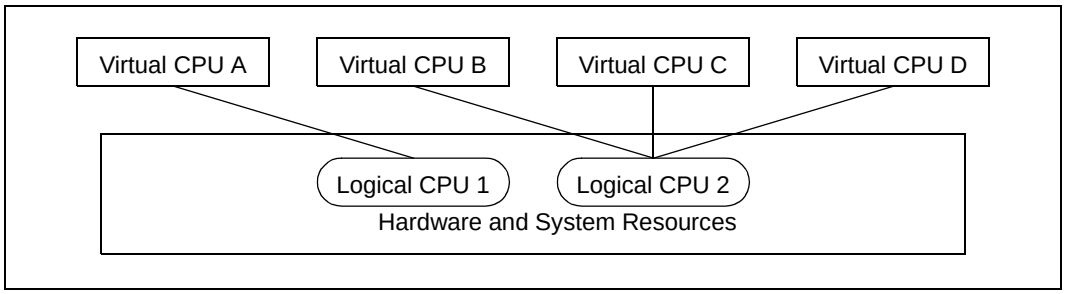


Figure 2-4. Dedicated Virtualization Policy



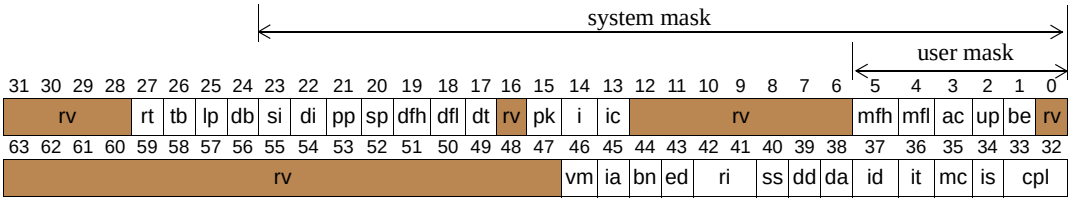
1. For example, external interruption resources like external interrupt control registers, TPR and PSR.i.

3 Itanium® Architecture Changes

The rest of this document is formatted as a specification update to the *Intel® Itanium® Architecture Software Developer's Manual*. This details out every change to the architecture for VT-i including the new instructions, processor behavior in the different virtualized modes, as well as the new PAL interfaces. The Itanium architecture is a living document and updates happen periodically. Future updates will be incorporated into the *Intel® Itanium® Architecture Software Developer's Manual* and specification updates.

1. Volume 2, Part I, Chapter 3 System State and Programming Model

1. New PSR.vm bit in Figure 3-2 Processor Status Register (PSR) (2:18):



2. New PSR.vm bit in Table 3-2 Processor Status Register Fields (2:19):

Field	Bits	Description	Interruption State	Serialization Required
vm	46	Virtual Machine – When 1, an attempt to execute certain instructions results in a virtualization fault. Implementation of this bit is optional. If the bit is not implemented, it is treated as a reserved bit. Written by the `rfi` and `vmsw` instructions.	0	rfi

3. New section, Section 3.4, Processor Virtualization (2:35):

   Itanium architecture processors may optionally implement a mechanism to support processor virtualization. This includes an additional PSR.vm bit (see Section 3.3.2, "Processor Status Register (PSR)"), which, when 1, causes certain instructions to take a virtualization fault (see Section 5.6, "Interruption Priorities" and "Virtualization Vector (0x6100)").

   The set of instructions which are virtualized by PSR.vm are listed in Table 3-10 below.

4. New Table 3-10, Virtualized Instructions (2:35):

Class	Virtualized Instructions
All privileged instructions	itc.i, itc.d, itr.i, itr.d, ptc.l, ptc.g, ptc.ga, ptc.e, ptr, tak, tpa, mov rr, mov pkr, mov cr, mov ibr, mov dbr, mov pmc, mov to pmd, ssm, rsm, mov psr, rfi, bsw
Some non-privileged instructions (virtualized at all privilege levels)	thash, ttag, mov from cpuid
Some non-privileged instructions (virtualized at privilege level 0)	cover

Class	Virtualized Instructions
Reading AR[ITC] with PSR.si==1 takes (virtualized at all privilege levels)	mov from ar.itc
Instructions which write privileged registers	mov to itc

5. New paragraph after Table 3-10 (2:35):

   Processors which support processor virtualization must provide an implementation-dependent mechanism for disabling the `vmsw` instruction. When enabled, the `vmsw` instruction functions as described on the `vmsw` instruction page. When disabled, the `vmsw` instruction always raises a virtualization fault when executed at the most privileged level.

   Processor virtualization is largely invisible to system software, and therefore its effects on virtualized instructions are not discussed in this document, except on the instruction description pages themselves.

2. Volume 2, Part I, Chapter 4, Addressing and Protection

1. Section 4.3.2, Unimplemented Virtual Address Bits, add the following paragraph before the final paragraph in the section (2:62):

   If the PSR.vm bit is implemented, and if PSR.vm is 1, then virtual addresses are treated as though one additional virtual address bit were unimplemented. If the PSR.vm bit is implemented, at least 52 virtual address bits must be implemented.

2. Section 4.3.3, Instruction Behavior with Unimplemented Addresses, add the following bullet after the last bullet (2:63):

   - The behavior of executing `vmsw.1` in a bundle whose address will become unimplemented after PSR.vm is set to 1 is undefined.

3. Volume 2, Part I, Chapter 5, Interruptions

1. Add Virtualization fault and Virtual External Interrupt in Table 5-6, Interruption Priorities (2:92):

Table 5-6 Interruption Priorities (Sheet 1 of 4)

Type	Instr. Set		Interruption Name	Vector Name	IA-32 Class[1]
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	
		6	Virtual External Interrupt (VINT)	Virtual External Interrupt vector	N/A

Table 5-6 Interruption Priorities (Sheet 2 of 4)

Type	Instr. Set		Interruption Name	Vector Name	IA-32 Class[1]
Faults	Intel Itanium	7	IR Unimplemented Data Address fault	General Exception vector	N/A
		8	IR Data Nested TLB fault	Data Nested TLB vector	
		9	IR Alternate Data TLB fault	Alternate Data TLB vector	
		10	IR VHPT Data fault	VHPT Translation vector	
		11	IR Data TLB fault	Data TLB vector	
		12	IR Data Page Not Present fault	Page Not Present vector	
		13	IR Data NaT Page Consumption fault	NaT Consumption vector	
		14	IR Data Key Miss fault	Data Key Miss vector	
		15	IR Data Key Permission fault	Key Permission vector	
		16	IR Data Access Rights fault	Data Access Rights vector	
		17	IR Data Access Bit fault	Data Access-Bit vector	
		18	IR Data Debug fault	Debug vector	
Faults	IA-32	19	IA-32 Instruction Breakpoint fault	IA-32 Exception vector (Debug)	A
		20	IA-32 Code Fetch fault[2]	IA-32 Exception vector (GPFault)	
	IA-32, Intel Itanium	21	Alternate Instruction TLB fault	Alternate Instruction TLB vector	
		22	VHPT Instruction fault	VHPT Translation vector	
		23	Instruction TLB fault	Instruction TLB vector	
		24	Instruction Page Not Present fault	Page Not Present vector	
		25	Instruction NaT Page Consumption fault	NaT Consumption vector	
		26	Instruction Key Miss fault	Instruction Key Miss vector	
		27	Instruction Key Permission fault	Key Permission vector	
		28	Instruction Access Rights fault	Instruction Access Rights vector	
		29	Instruction Access Bit fault	Instruction Access-Bit vector	
	Intel Itanium	30	Instruction Debug fault	Debug vector	
	IA-32	31	IA-32 Instruction Length > 15 bytes	IA-32 Exception vector (GPFault)	B
		32	IA-32 Invalid Opcode fault	IA-32 Intercept vector (Instruction)	
		33	IA-32 Instruction Intercept fault	IA-32 Intercept vector (Instruction)	
	Intel Itanium	34	Illegal Operation fault[3]	General Exception vector	
		35	Illegal Dependency fault	General Exception vector	
		36	Break Instruction fault	Break Instruction vector	
		37	Privileged Operation fault	General Exception vector	

Table 5-6 Interruption Priorities (Sheet 3 of 4)

Type	Instr. Set	Interruption Name	Vector Name	IA-32 Class[1]
	IA-32, Intel Itanium	38 Disabled Floating-point Register fault	Disabled FP-Register vector	B
		39 Disabled Instruction Set Transition fault	General Exception vector	
	IA-32	40 IA-32 Device Not Available fault	IA-32 Exception vector (DNA)	
		41 IA-32 FP Error fault[4]	IA-32 Exception vector (FPError)	C
	IA-32, Intel Itanium	42 Register NaT Consumption fault	NaT Consumption vector	
	Intel Itanium	43 Reserved Register/Field fault	General Exception vector	
		44 Unimplemented Data Address fault	General Exception vector	
		45 Privileged Register fault	General Exception vector	
		46 Speculative Operation fault	Speculation vector	
		47 Virtualization fault	Virtualization vector	
	IA-32	48 IA-32 Stack Exception	IA-32 Exception vector (StackFault)	C
		49 IA-32 General Protection Fault	IA-32 Exception vector (GPFault)	
Faults	IA-32, Intel Itanium	50 Data Nested TLB fault	Data Nested TLB vector	
		51 Alternate Data TLB fault[5]	Alternate Data TLB vector	
		52 VHPT Data fault[e]	VHPT Translation vector	
		53 Data TLB fault[e]	Data TLB vector	
		54 Data Page Not Present fault[e]	Page Not Present vector	
		55 Data NaT Page Consumption fault[e]	NaT Consumption vector	
		56 Data Key Miss fault[e]	Data Key Miss vector	
		57 Data Key Permission fault[e]	Key Permission vector	
		58 Data Access Rights fault[e]	Data Access Rights vector	
		59 Data Dirty Bit fault	Dirty-Bit vector	
		60 Data Access Bit fault[e]	Data Access-Bit vector	
	Intel Itanium	61 Data Debug fault[e]	Debug vector	
		62 Unaligned Data Reference fault[e]	Unaligned Reference vector	
	IA-32	63 IA-32 Alignment Check fault	IA-32 Exception vector (Align-mentCheck)	C
		64 IA-32 Locked Data Reference fault	IA-32 Intercept vector (Lock)	
		65 IA-32 Segment Not Present fault	IA-32 Exception vector (NotPresent)	
		66 IA-32 Divide by Zero fault	IA-32 Exception vector (Divide)	
		67 IA-32 Bound fault	IA-32 Exception vector (Bound)	
		68 IA-32 SSE Numeric Error fault	IA-32 Exception vector (Stream-SIMD)	

Table 5-6 Interruption Priorities (Sheet 4 of 4)

Type	Instr. Set		Interruption Name	Vector Name	IA-32 Class[1]
	Intel Itanium	69	Unsupported Data Reference fault	Unsupported Data Reference vector	
		70	Floating-point fault	Floating-point Fault vector	
Traps	Intel Itanium	71	Unimplemented Instruction Address trap[6]	Lower-Privilege Transfer Trap vector	
		72	Floating-point trap	Floating-point Trap vector	
		73	Lower-Privilege Transfer trap	Lower-Privilege Transfer Trap vector	
		74	Taken Branch trap	Taken Branch Trap vector	
		75	Single Step trap	Single Step Trap vector	
	IA-32	76	IA-32 System Flag Intercept trap	IA-32 Intercept vector (SystemFlag)	D
		77	IA-32 Gate Intercept trap	IA-32 Intercept vector (Gate)	
		78	IA-32 INTO trap	IA-32 Exception vector (Overflow)	
		79	IA-32 Breakpoint (INT 3) trap	IA-32 Exception vector (Debug)	
		80	IA-32 Software Interrupt (INT) trap	IA-32 Interrupt vector (Vector#)	
		81	IA-32 Data Breakpoint trap	IA-32 Exception vector (Debug)	
		82	IA-32 Taken Branch trap	IA-32 Exception vector (Debug)	
		83	IA-32 Single Step trap	IA-32 Exception vector (Debug)	

NOTES:
1. IA-32 Interruption Class, see Section 5.6.1, “IA-32 Interruption Priorities and Classes” on page 2:105 for details.
2. IA-32 Code Fetch faults include Code Segment Limit Violation and other Code Fetch checks defined in Section 6.2.3.3, “IA-32 Environment Runtime Integrity Checks” on page 121.
3. Illegal Operation faults can be taken for certain predicated off reserved opcodes. For details, refer to Section 4.1, “Format Summary” on page 272.
4. 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.
5. If not deferred.
6. Unimplemented Instruction Address traps on emulated check instructions have a lower priority than Taken Branch trap and Single Step trap. See “Speculation vector (0x5700)” on page 193.

2. Add Virtual External Interrupt vector and virtualization vector in Table 5-7, Interruption Vector Table (IVT) (2:96):

Offset	Vector Name	Interruption(s)	Page
0x3400	Virtual External Interrupt vector	6	2:183
0x6100	Virtualization vector	47	2:202

4. Volume 2, Part I, Chapter 8, Interruption Vector Descriptions

1. Add Virtual External Interrupt vector and virtualization vector in Table 8-1, Writing of Interruption Resources by Vector (2:146):

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												
Virtual External Interrupt vector	-	W	x	x	x	x	x	x	x	x	W	W
Virtualization vector	-	W	x	x	x	x	x	x	x	x	W	W

2. Add Virtual External Interrupt vector and Virtualization vector in Table 8-2, ISR Values on Interruption (2:147):

Vector / Interruption	ed	ei[1]	so	ni[2]	ir[3]	rs[4]	sp[5]	na[6]	r	w	x
Virtual External Interrupt Vector											
Virtual External Interrupt	0	ri	0	ni	ir[7]	0	0	0	0	0	0
Virtualization Vector											
Virtualization Fault	0	ri	0	ni	0	0	0	0	0	0	0

NOTES:
1. 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).
2. If ISR.ni is 1, the interruption occurred either when PSR.ic was 0 or was in-flight.
3. ISR.ri captures the value of RSE.CFLE at the time of an interruption.
4. ISR.rs is 1 for interruptions caused by mandatory RSE fills/spills and 0 for all others.
5. ISR.sp is 1 for interruptions caused by speculative loads and zero for all others.
6. ISR.na is 1 for interruptions caused by non-access instructions and zero for all others.
7. 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.

3. Add Virtual External Interrupt vector (0x3400) and Virtualization vector (0x6100) (2:186):

Name **Virtual External Interrupt Vector (0x3400)**

Cause The guest highest pending interrupt (VHPI) specified by the VMM is unmasked on the virtual processor.

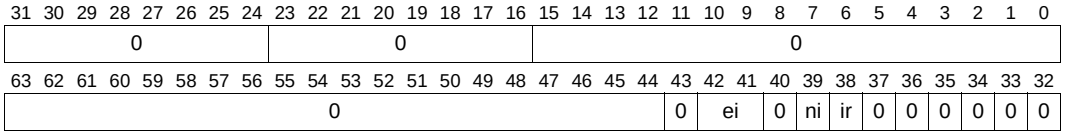
IPSR.is indicates which instruction set was executing at the time of the interruption.

Interruptions on this vector:

Virtual External interrupt

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:163 for a detailed description.

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.



Notes: Software is expected to avoid situations which could cause ISR.ni to be 1.

Name **Virtualization Vector (0x6100)**

Cause An attempt is made to execute an instruction which requires virtualization. This fault cannot be raised by IA-32 instructions.

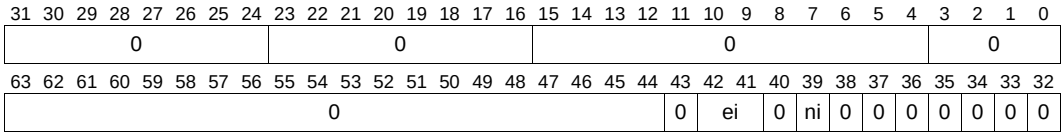
Interruptions on this vector:

Virtualization fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:163 for a detailed description.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception.

The defined ISR bits are specified below.



5. Volume 2, Part I, Chapter 11, Processor Abstraction Layer

1. Add PSR.vm bit in Table 11-19, State Requirements for PSR (2:289):

PSR Bit	Description	Entry	Exit	Class
vm	processor virtualization	0	0	unchanged

2. Add bits 40 and 54 in Table 11-54, Processor Features (2:360):

Bit	Class	Control	Description
40	Opt.	No	Virtual Machine features implemented. Denotes whether PSR.vm is implemented. 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.
54	Opt.	Req.	Enable the use of the VMSW instruction. When 0, the `vmsw` instruction causes a virtualization fault when executed at the most privileged level. When 1, this bit will enable normal operation of the `vmsw` instruction.

6. Volume 3, Chapter 2, Instruction Reference

See "Instruction Reference" on page 16 for changes related to the virtualized instructions.

7. Revised Chapter 11 of Volume 2, Processor Abstraction Layer (text included at end of this update)

Volume 2, Chapter 11, Processor Abstraction Layer has been modified to include new content to support processor virtualization. The new content from Chapter 11 is presented at the end of this update for convenience.

§

4 Instruction Reference

The subsequent pages list the changes related to the virtualized instructions.

bsw — Bank Switch

Format:

bsw.0	zero_form	B8
bsw.1	one_form	B8

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

A `bsw` instruction must be the last instruction in an instruction group. Otherwise, an Illegal Operation fault is taken. Instructions in the same instruction group that access GR16 to GR31 reference the previous register bank. Subsequent instruction groups reference the new register bank.

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

This instruction cannot be predicated.

Operation:

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

if (PSR.cpl != 0)
    privileged_operation_fault(0);

if (PSR.vm == 1)
    virtualization_fault();

if (zero_form)
    PSR.bn = 0;
else // one_form
    PSR.bn = 1;
```

Interruptions: Illegal Operation fault, Virtualization fault, Privileged Operation fault

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

cover — Cover Stack Frame

Format: cover B8

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

A `cover` instruction must be the last instruction in an instruction group. Otherwise, an Illegal Operation fault is taken.

If PSR.cpl is non-zero, this instruction can only be executed when PSR.vm is also 0. This instruction cannot be predicated.

Operation:

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

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

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

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

Interruptions: Illegal Operation fault Virtualization fault

itc — Insert Translation Cache

Format:

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

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

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

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

The TLB is first purged of any overlapping entries as specified by Table 4-1 on page 49.

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

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

Operation:

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

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

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

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

Interruptions:

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

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

itr — Insert Translation Register

Format:

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

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

As described in Table 4-1, “Purge Behavior of TLB Instructions” on page 49, the TLB is first purged of any entries that overlap with the newly inserted translation. The translation previously contained in the TR slot specified by GR r_3 is not necessarily purged from the processor's TLBs and may remain as a TC entry. To ensure that the previous TR translation is purged, software must use explicit `ptr` instructions before inserting the new TR entry.

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

Operation:

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

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

    tmp_tr_type = instruction_form ? ITR_TYPE : DTR_TYPE;

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

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

```
    }
}
```

Interruptions: Machine Check abort
Illegal Operation fault
fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Unimplemented Data Address
Virtualization fault

mov — Move Application Register

Format:

(*qp*) mov $r_1 = ar_3$	pseudo-op		
(*qp*) mov $ar_3 = r_2$	pseudo-op		
(*qp*) mov $ar_3 = imm_8$	pseudo-op		
(*qp*) mov.i $r_1 = ar_3$	i_form, from_form	I28	
(*qp*) mov.i $ar_3 = r_2$	i_form, register_form, to_form	I26	
(*qp*) mov.i $ar_3 = imm_8$	i_form, immediate_form, to_form	I27	
(*qp*) mov.m $r_1 = ar_3$	m_form, from_form	M31	
(*qp*) mov.m $ar_3 = r_2$	m_form, register_form, to_form	M29	
(*qp*) mov.m $ar_3 = imm_8$	m_form, immediate_form, to_form	M30	

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

In the from_form, the application register specified by ar_3 is copied into GR r_1 and the corresponding NaT bit is cleared.

In the to_form, the value in GR r_2 (in the register_form), or the sign-extended value in imm_8 (in the immediate_form), is placed in AR ar_3. In the register_form if the NaT bit corresponding to GR r_2 is set, then a Register NaT Consumption fault is raised.

Only a subset of the application registers can be accessed by each execution unit (M or I). Table 3-3 on page 28 indicates which application registers may be accessed from which execution unit type. An access to an application register from the wrong unit type causes an Illegal Operation fault.

This instruction has multiple forms with the pseudo operation eliminating the need for specifying the execution unit. Accesses of the ARs are always implicitly serialized. While implicitly serialized, read-after-write and write-after-write dependency violations must be avoided (e.g., setting CCV, followed by `cmpxchg` in the same instruction group, or simultaneous writes to the UNAT register by `ld.fill` and mov to UNAT).

Operation:

```
if (PR[qp]) {
    tmp_type = (i_form ? AR_I_TYPE : AR_M_TYPE);
    if (is_reserved_reg(tmp_type, ar3))
        illegal_operation_fault();

    if (from_form) {
        check_target_register(r1);
        if (((ar3 == BSPSTORE) || (ar3 == RNAT)) && (AR[RSC].mode
!= 0))
            illegal_operation_fault();

        if (ar3 == ITC && PSR.si && PSR.cpl != 0)
            privileged_register_fault();

        if (ar3 == ITC && PSR.si && PSR.vm == 1)
            virtualization_fault();

        GR[r1] = (is_ignored_reg(ar3)) ? 0 : AR[ar3];
        GR[r1].nat = 0;
    } else {                                          // to_form
        tmp_val = (register_form) ? GR[r2] : sign_ext(imm8, 8);

        if (is_read_only_register(AR_TYPE, ar3) ||
            (((ar3 == BSPSTORE) || (ar3 == RNAT)) && (AR[RSC].mode
!= 0)))
            illegal_operation_fault();

        if (register_form && GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(AR_TYPE, ar3, tmp_val))
            reserved_register_field_fault();

        if ((is_kernel_reg(ar3) || ar3 == ITC) && (PSR.cpl != 0))
            privileged_register_fault();

        if (ar3 == ITC && PSR.vm == 1)
            virtualization_fault();

        if (!is_ignored_reg(ar3)) {
            tmp_val = ignored_field_mask(AR_TYPE, ar3, tmp_val);
            // check for illegal promotion
            if (ar3 == RSC && tmp_val{3:2} u< PSR.cpl)
                tmp_val{3:2} = PSR.cpl;
            AR[ar3] = tmp_val;

            if (ar3 == BSPSTORE) {
                AR[BSP] =
rse_update_internal_stack_pointers(tmp_val);
                AR[RNAT] = undefined();
            }
        }
    }
}
```

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

mov — Move Control Register

Format:

(*qp*) mov $r_1 = cr_3$	from_form	M33
(*qp*) mov $cr_3 = r_2$	to_form	M32

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

For the from_form, the control register specified by cr_3 is read and the value copied into GR r_1.

For the to_form, GR r_2 is read and the value copied into CR cr_3.

Control registers can only be accessed at the most privileged level, and when PSR.vm is 0. Reading or writing an interruption control register (CR16-CR25), when the PSR.ic bit is one, will result in an Illegal Operation fault.

Operation:

```
if (PR[qp]) {
    if (is_reserved_reg(CR_TYPE, cr3)
        || to_form && is_read_only_reg(CR_TYPE, cr3)
        || PSR.ic && is_interruption_cr(cr3))
    {
        illegal_operation_fault();
    }

    if (from_form)
        check_target_register(r1);
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (from_form) {
        if (PSR.vm == 1)
            virtualization_fault();
        if (cr3 == IVR)
            check_interrupt_request();

        if (cr3 == ITIR)
            GR[r1] = impl_itir_cwi_mask(CR[ITIR]);
        else
            GR[r1] = CR[cr3];

        GR[r1].nat = 0;
    } else {            // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(CR_TYPE, cr3, GR[r2]))
            reserved_register_field_fault();
        if ((cr3 == IFA) && impl_check_mov_ifa() &&
            unimplemented_virtual_address(GR[r2], PSR.vm))
            unimplemented_data_address_fault(0);
        if (PSR.vm == 1)
            virtualization_fault();
        if (cr3 == EOI)
            end_of_interrupt();

        tmp_val = ignored_field_mask(CR_TYPE, cr3, GR[r2]);
        CR[cr3] = tmp_val;
        if (cr3 == IIPA)
            last_IP = tmp_val;
    }
}
```

Interruptions:

Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Virtualization fault

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

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

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

mov — Move Indirect Register

Format:

(*qp*) mov r_1 = *ireg*[r_3] from_form M43
(*qp*) mov *ireg*[r_3] = r_2 to_form M42

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

For move from indirect register, GR r_3 is read and the value used as an index into the register file specified by *ireg* (see Table 4-1 below). The indexed register is read and its value is copied into GR r_1.

For move to indirect register, GR r_3 is read and the value used as an index into the register file specified by *ireg*. GR r_2 is read and its value copied into the indexed register.

Table 4-1. Indirect Register File Mnemonics

ireg	Register File
cpuid	Processor Identification Register
dbr	Data Breakpoint Register
ibr	Instruction Breakpoint Register
pkr	Protection Key Register
pmc	Performance Monitor Configuration Register
pmd	Performance Monitor Data Register
rr	Region Register

For all register files other than the region registers, bits {7:0} of GR r_3 are used as the index. For region registers, bits {63:61} are used. The remainder of the bits are ignored.

Instruction and data breakpoint, performance monitor configuration, protection key, and region registers can only be accessed at the most privileged level. Performance monitor data registers can only be written at the most privileged level.

The CPU identification registers can only be read. There is no to_form of this instruction.

For move to protection key register, the processor ensures uniqueness of protection keys by checking new valid protection keys against all protection key registers. If any matching keys are found, duplicate protection keys are invalidated.

Apart from the PMC and PMD register files, access of a non-existent register results in a Reserved Register/Field fault. All accesses to the implementation-dependent portion of PMC and PMD register files result in implementation dependent behavior but do not fault.

Modifying a region register or a protection key register which is being used to translate:

- The executing instruction stream when PSR.it == 1, or
- The data space for an eager RSE reference when PSR.rt == 1

is an undefined operation.

Operation:

```
if (PR[qp]) {
    if (ireg == RR_TYPE)
        tmp_index = GR[r3]{63:61};
    else // all other register types
        tmp_index = GR[r3]{7:0};

    if (from_form) {
        check_target_register(r1);

        if (PSR.cpl != 0 && !(ireg == PMD_TYPE || ireg ==
CPUID_TYPE))
            privileged_operation_fault(0);

        if (GR[r3].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_reg(ireg, tmp_index))
            reserved_register_field_fault();

        if (PSR.vm == 1 && ireg != PMD_TYPE)
            virtualization_fault();

        if (ireg == PMD_TYPE) {
            if ((PSR.cpl != 0) && ((PSR.sp == 1) ||
                 (tmp_index > 3 &&
                 tmp_index <= IMPL_MAXGENERIC_PMCPMD &&
                 PMC[tmp_index].pm == 1)))
                GR[r1] = 0;
            else
                GR[r1] = pmd_read(tmp_index);
        } else
            switch (ireg) {
                case CPUID_TYPE: GR[r1] = CPUID[tmp_index];
break;
                case DBR_TYPE:   GR[r1] = DBR[tmp_index]; break;
                case IBR_TYPE:   GR[r1] = IBR[tmp_index]; break;
                case PKR_TYPE:   GR[r1] = PKR[tmp_index]; break;
                case PMC_TYPE:   GR[r1] = pmc_read(tmp_index);
break;
                case RR_TYPE:    GR[r1] = RR[tmp_index]; break;
            }
        GR[r1].nat = 0;
    } else {         // to_form
        if (PSR.cpl != 0)
            privileged_operation_fault(0);

        if (GR[r2].nat || GR[r3].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_reg(ireg, tmp_index)
            || is_reserved_field(ireg, tmp_index, GR[r2]))
            reserved_register_field_fault();
        if (PSR.vm == 1)
            virtualization_fault();
        if (ireg == PKR_TYPE && GR[r2]{0} == 1) { // writing
valid prot key
            if ((tmp_slot = tlb_search_pkr(GR[r2]{31:8})) !=
NOT_FOUND)
                PKR[tmp_slot].v = 0; // clear valid bit of
matching key reg
        }
        tmp_val = ignored_field_mask(ireg, tmp_index, GR[r2]);
```

```
        switch (ireg) {
            case DBR_TYPE:    DBR[tmp_index] = tmp_val; break;
            case IBR_TYPE:    IBR[tmp_index] = tmp_val; break;
            case PKR_TYPE:    PKR[tmp_index] = tmp_val; break;
            case PMC_TYPE:    pmc_write(tmp_index, tmp_val);
break;
            case PMD_TYPE:    pmd_write(tmp_index, tmp_val);
break;
            case RR_TYPE:     RR[tmp_index]= tmp_val; break;
        }
    }
}
```

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

Serialization: For move to data breakpoint registers, software must issue a data serialize operation before issuing a memory reference dependent on the modified register.

For move to instruction breakpoint registers, software must issue an instruction serialize operation before fetching an instruction dependent on the modified register.

For move to protection key, region, performance monitor configuration, and performance monitor data registers, software must issue an instruction or data serialize operation to ensure the changes are observed before issuing any dependent instruction.

To obtain improved accuracy, software can issue an instruction or data serialize operation before reading the performance monitors.

mov — Move Processor Status Register

Format:

(*qp*) mov r_1 = psr — from_form M36
(*qp*) mov psr.l = r_2 — to_form M35

Description:

The source operand is copied to the destination register. See Section 3.3.2, "Processor Status Register (PSR)" on page 22.

For move from processor status register, PSR bits {36:35} and {31:0} are read, and copied into GR r_1. All other bits of the PSR read as zero.

For move to processor status register, GR r_2 is read, bits {31:0} copied into PSR{31:0} and bits {45:32} are ignored. All bits of GR r_2 corresponding to reserved fields of the PSR must be 0 or a Reserved Register/Field fault will result.

Moves to and from the PSR can only be performed at the most privileged level, and when PSR.vm is 0.

The contents of the interruption resources (that are overwritten when the PSR.ic bit is 1) are undefined if an interruption occurs between the enabling of the PSR.ic bit and a subsequent instruction serialize operation.

Operation:

```
if (PR[qp]) {
    if (from_form)
        check_target_register(r1);
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (from_form) {
        if (PSR.vm == 1)
            virtualization_fault();
        tmp_val = zero_ext(PSR{31:0}, 32);    // read lower 32
bits
        tmp_val |= PSR{36:35} << 35;          // read mc and it
bits
        GR[r1] = tmp_val;                     // other bits read
as zero
        GR[r1].nat = 0;
    } else {          // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(PSR_TYPE, PSR_MOVPART, GR[r2]))
            reserved_register_field_fault();

        if (PSR.vm == 1)
            virtualization_fault();

        PSR{31:0} = GR[r2]{31:0};
    }
}
```

Interruptions:

Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Virtualization fault

Serialization:

Software must issue an instruction or data serialize operation before issuing instructions dependent upon the altered PSR bits. Unlike with the `rsm` instruction, the PSR.i bit is not treated specially when cleared.

ptc.e — Purge Translation Cache Entry

Format: (*qp*) ptc.e r_3 M47

Description: One or more translation entries are purged from the local processor's instruction and data translation cache. Translation Registers and the VHPT are not modified.

The number of translation cache entries purged is implementation specific. Some implementations may purge all levels of the translation cache hierarchy with one iteration of PTC.e, while other implementations may require several iterations to flush all levels, sets and associativities of both instruction and data translation caches. GR r_3 specifies an implementation-specific parameter associated with each iteration.

The following loop is defined to flush the entire translation cache for all processor models. Software can acquire parameters through a processor dependent layer that is accessed through a procedural interface. The selected region registers must remain unchanged during the loop.

```
disable_interrupts();
addr = base;
for (i = 0; i < count1; i++) {
    for (j = 0; j < count2; j++) {
            ptc.e(addr);
            addr += stride2;
        }
    addr += stride1;
}
enable_interrupts();
```

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

Operation:

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

Interruptions: Privileged Operation fault Virtualization fault
Register NaT Consumption fault

Serialization: Software must issue a data serialization operation to ensure the purge is complete before issuing a data access or non-access reference dependent upon the purge. Software must issue instruction serialize operation before fetching an instruction dependent upon the purge.

ptc.g, ptc.ga — Purge Global Translation Cache

Format:

(*qp*) ptc.g r_3, r_2	global_form	M45
(*qp*) ptc.ga r_3, r_2	global_alat_form	M45

Description:

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

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

Based on the processor model, the translation cache may be also purged of more translations than specified by the purge parameters up to and including removal of all entries within the translation cache.

`ptc.g` has release semantics and is guaranteed to be made visible after all previous data memory accesses are made visible. The memory fence instruction forces all processors to complete the purge prior to any subsequent memory operations. Serialization is still required to observe the side-effects of a translation being removed.

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

The behavior of the `ptc.ga` instruction is similar to `ptc.g`. In addition to the behavior specified for `ptc.g` the `ptc.ga` instruction encodes an extra bit of information in the broadcast transaction. This information specifies the purge is due to a page remapping as opposed to a protection change or page tear down. The remote processors within the coherence domain will then take what ever additional action is necessary to make their ALAT consistent. The local ALAT is not purged.

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

Unless specifically supported by the processors and platform, only one global purge transaction may be issued at a time by all processors, the operation is undefined otherwise. Software is responsible for enforcing this restriction. Implementations may optionally support multiple concurrent global purge transactions. The firmware returns if implementations support this optional behavior.

Propagation of `ptc.g` between multiple local TLB coherence domains is platform dependent, and must be handled by software. It is expected that the local TLB coherence domain covers at least the processors on the same local bus.

Operation:

```
if (PR[qp]) {
    if (!followed_by_stop())
        undefined_behavior();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();
```

```
    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);
    tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);

    if (global_alat_form) tmp_ptc_type = GLOBAL_ALAT_FORM;
    else tmp_ptc_type = GLOBAL_FORM;

    tlb_broadcast_purge(tmp_rid, tmp_va, tmp_size,
tmp_ptc_type);
}
```

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

Serialization: The broadcast purge TC is not synchronized with the instruction stream on a remote processor. Software cannot depend on any such synchronization with the instruction stream. Hardware on the remote machine cannot reload an instruction from memory or cache after acknowledging a broadcast purge TC without first retranslating the I-side access in the TLB. Hardware may continue to use a valid private copy of the instruction stream data (possibly in an I-buffer) obtained prior to acknowledging a broadcast purge TC to a page containing the i-stream data. Hardware must retranslate access to an instruction page upon an interruption or any explicit or implicit instruction serialization event (e.g., `srlz.i`, `rfi`).

Software must issue the appropriate data and/or instruction serialization operation to ensure the purge is completed before a local data access, non-access reference, or local instruction fetch access dependent upon the purge.

ptc.l — Purge Local Translation Cache

Format: (*qp*) ptc.l r_3, r_2 M45

Description: The instruction and data translation cache of the local processor is searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. All these entries are removed.

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

The processor ensures that all entries matching the purging parameters are removed. However, based on the processor model, the translation cache may be also purged of more translations than specified by the purge parameters up to and including removal of all entries within the translation cache.

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

This is a local operation, no purge broadcast to other processors occurs in a multiprocessor system.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);
    tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
}
```

Interruptions:

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

Serialization: Software must issue the appropriate data and/or instruction serialization operation to ensure the purge is completed before a data access, non-access reference, or instruction fetch access dependent upon the purge.

ptr — Purge Translation Register

Format:

(*qp*) ptr.d r_3, r_2	data_form	M45
(*qp*) ptr.i r_3, r_2	instruction_form	M45

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

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

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

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

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

This is a local operation, no purge broadcast to other processors occurs in a multiprocessor system.

As described in Section 4.1.1.2, “Translation Cache (TC)” on page 47, the processor may use the translation caches to cache virtual address mappings held by translation registers. The `ptr.i` and `ptr.d` instructions purge the processor’s translation registers as well as cached translation register copies that may be contained in the respective translation caches.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    if (data_form) {
        tlb_must_purge_dtr_entries(tmp_rid, tmp_va, tmp_size);
        tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
    } else {                                          //
instruction_form
        tlb_must_purge_itr_entries(tmp_rid, tmp_va, tmp_size);
        tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    }
}
```

Interruptions: Privileged Operation fault; Unimplemented Data Address fault; Register NaT Consumption fault; Virtualization fault

Serialization: For the data form, software must issue a data serialization operation to ensure the purge is completed before issuing an instruction dependent upon the purge. For the instruction form, software must issue an instruction serialization operation to ensure the purge is completed before fetching an instruction dependent on that purge.

rfi — Return From Interruption

Format: `rfi` B8

Description: The machine context prior to an interruption is restored. PSR is restored from IPSR, IPSR is unmodified, and IP is restored from IIP. Execution continues at the bundle address loaded into the IP, and the instruction slot loaded into PSR.ri.

This instruction must be immediately followed by a stop. Otherwise, an Illegal Operation fault is taken. This instruction switches to the register bank specified by IPSR.bn. Instructions in the same instruction group that access GR16 to GR31 reference the previous register bank. Subsequent instruction groups reference the new register bank.

This instruction performs instruction serialization, which ensures:

- Prior modifications to processor register resources that affect fetching of subsequent instruction groups are observed.
- Prior modifications to processor register resources that affect subsequent execution or data memory accesses are observed.
- Prior memory synchronization (`sync.i`) operations have taken effect on the local processor instruction cache.
- Subsequent instruction group fetches (including the target instruction group) are re-initiated after `rfi` completes.

The `rfi` instruction must be in an instruction group after the instruction group containing the operation that is to be serialized.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0. This instruction can not be predicated.

Execution of this instruction is undefined if PSR.ic or PSR.i are 1 and PSR.vm is 0. Software must ensure that an interruption cannot occur that could modify IIP, IPSR, or IFS between when they are written and the subsequent `rfi`.

This instruction does not take Lower Privilege Transfer, Taken Branch or Single Step traps.

If this instruction sets PSR.ri to 2 and the target is an MLX bundle, then an Illegal Operation fault will be taken on the target bundle.

If IPSR.is is 1, control is resumed in the IA-32 instruction set at the virtual linear address specified by IIP{31:0}. PSR.di does not inhibit instruction set transitions for this instruction. If PSR.dfh is 1 after `rfi` completes execution, a Disabled FP Register fault is raised on the target IA-32 instruction.

If IPSR.is is 1 and an Unimplemented Instruction Address trap is taken, IIP will contain the original 64-bit target IP. (The value will not have been zero extended from 32 bits.)

When entering the IA-32 instruction set, the size of the current stack frame is set to zero, and all stacked general registers are left in an undefined state. Software can not rely on the value of these registers across an instruction set transition. Software must ensure that BSPSTORE==BSP on entry to the IA-32 instruction set, otherwise undefined behavior may result.

If IPSR.is is 1, software must set other IPSR fields properly for IA-32 instruction set execution; otherwise processor operation is undefined. See Table 3-2, “Processor Status Register Fields” on page 23 for details.

Software must issue a `mf` instruction before this instruction if memory ordering is required between IA-32 processor-consistent and Itanium unordered memory references. The processor does not ensure Itanium-instruction-set-generated writes into the instruction stream are seen by subsequent IA-32 instructions.

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

IA-32 instruction set execution leaves the contents of the ALAT undefined. Software can not rely on ALAT state across an instruction set transition. On entry to IA-32 code, existing entries in the ALAT are ignored.

Operation:

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

unimplemented_address = 0;
if (PSR.cpl != 0)
    privileged_operation_fault(0);

if (PSR.vm == 1)
    virtualization_fault();

if (PSR.ic == 1 || PSR.i == 1)
    undefined_behavior();

taken_rfi = 1;

PSR = CR[IPSR];
if (CR[IPSR].is == 1) {        //resume IA-32 instruction set
    if (CR[IPSR].ic == 0 || CR[IPSR].dt == 0 ||
        CR[IPSR}.mc == 1 || CR[IPSR].it == 0)
        undefined_behavior();
    tmp_IP = CR[IIP];
    if ((CR[IPSR].it && unimplemented_virtual_address(tmp_IP,
IPSR.vm))
        || (!CR[IPSR].it &&
unimplemented_physical_address(tmp_IP)))
        unimplemented_address = 1;
                                   //compute effective instruction
pointer
    EIP{31:0} = CR[IIP]{31:0} - AR[CSD].Base;
                                   //force zero-sized restored
frame
    rse_restore_frame(0, 0, CFM.sof);
    CFM.sof = 0;
    CFM.sol = 0;
    CFM.sor = 0;
    CFM.rrb.gr = 0;
    CFM.rrb.fr = 0;
```

```
    CFM.rrb.pr = 0;
    rse_invalidate_non_current_regs();
    //The register stack engine is disabled during IA-32
    //instruction set execution.
} else {                        //return to Itanium instruction
set
    tmp_IP = CR[IIP] & ~0xf;
    slot = CR[IPSR].ri;
    if ((CR[IPSR].it && unimplemented_virtual_address(tmp_IP,
IPSR.vm))
        || (!CR[IPSR].it &&
unimplemented_physical_address(tmp_IP)))
        unimplemented_address = 1;
    if (CR[IFS].v) {
        tmp_growth = -CFM.sof;
        alat_frame_update(-CR[IFS].ifm.sof, 0);
        rse_restore_frame(CR[IFS].ifm.sof, tmp_growth, CFM.sof);
        CFM = CR[IFS].ifm;
    }
    rse_enable_current_frame_load();
}
IP = tmp_IP;
instruction_serialize();
if (unimplemented_address)
    unimplemented_instruction_address_trap(0, tmp_IP);
```

Interruptions: Illegal Operation fault — Virtualization fault
Privileged Operation fault — Unimplemented Instruction Address trap

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

Serialization: An implicit instruction and data serialization operation is performed.

rsm — Reset System Mask

Format: (*qp*) rsm imm_{24} M44

Description: The complement of the imm_{24} operand is ANDed with the system mask (PSR{23:0}) and the result is placed in the system mask. See Section 3.3.2, "Processor Status Register (PSR)" on page 22.

The PSR system mask can only be written at the most privileged level, and when PSR.vm is 0.

When the current privilege level is zero (PSR.cpl is 0), an `rsm` instruction whose mask includes PSR.i may cause external interrupts to be disabled for an implementation-dependent number of instructions, even if the qualifying predicate for the `rsm` instruction is false. Architecturally, the extents of this external interrupt disabling "window" are defined as follows:

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

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

When the current privilege level is non-zero (PSR.cpl is not 0), an `rsm` instruction whose mask includes PSR.i may briefly disable external interrupts, regardless of the value of the qualifying predicate of the `rsm` instruction. However, processor implementations guarantee that non-privileged code cannot lock out external interrupts indefinitely (e.g., via an arbitrarily long sequence of `rsm` instructions with zero-valued qualifying predicates).

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (is_reserved_field(PSR_TYPE, PSR_SM, imm24))
        reserved_register_field_fault();

    if (PSR.vm == 1)
        virtualization_fault();

    if (imm24{1})     PSR{1} = 0;)     // be
    if (imm24{2})     PSR{2} = 0;)     // up
    if (imm24{3})     PSR{3} = 0;)     // ac
    if (imm24{4})     PSR{4} = 0;)     // mfl
    if (imm24{5})     PSR{5} = 0;)     // mfh
    if (imm24{13})    PSR{13} = 0;)    // ic
    if (imm24{14})    PSR{14} = 0;)    // i
    if (imm24{15})    PSR{15} = 0;)    // pk
    if (imm24{17})    PSR{17} = 0;)    // dt
    if (imm24{18})    PSR{18} = 0;)    // dfl
```

```
    if (imm24{19})    PSR{19} = 0;)    // dfh
    if (imm24{20})    PSR{20} = 0;)    // sp
    if (imm24{21})    PSR{21} = 0;)    // pp
    if (imm24{22})    PSR{22} = 0;)    // di
    if (imm24{23})    PSR{23} = 0;)    // si
}
```

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

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

ssm — Set System Mask

Format: (*qp*) ssm imm_{24} M44

Description: The imm_{24} operand is ORed with the system mask (PSR{23:0}) and the result is placed in the system mask. See Section 3.3.2, “Processor Status Register (PSR)” on page 22.

The PSR system mask can only be written at the most privileged level, and when PSR.vm is 0.

The contents of the interruption resources (that are overwritten when the PSR.ic bit is 1), are undefined if an interruption occurs between the enabling of the PSR.ic bit and a subsequent instruction serialize operation.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (is_reserved_field(PSR_TYPE, PSR_SM, imm24))
        reserved_register_field_fault();

    if (PSR.vm == 1)
        virtualization_fault();

    if (imm24{1})     PSR{1} = 1;)     // be
    if (imm24{2})     PSR{2} = 1;)     // up
    if (imm24{3})     PSR{3} = 1;)     // ac
    if (imm24{4})     PSR{4} = 1;)     // mfl
    if (imm24{5})     PSR{5} = 1;)     // mfh
    if (imm24{13})    PSR{13} = 1;)    // ic
    if (imm24{14})    PSR{14} = 1;)    // i
    if (imm24{15})    PSR{15} = 1;)    // pk
    if (imm24{17})    PSR{17} = 1;)    // dt
    if (imm24{18})    PSR{18} = 1;)    // dfl
    if (imm24{19})    PSR{19} = 1;)    // dfh
    if (imm24{20})    PSR{20} = 1;)    // sp
    if (imm24{21})    PSR{21} = 1;)    // pp
    if (imm24{22})    PSR{22} = 1;)    // di
    if (imm24{23})    PSR{23} = 1;)    // si
}
```

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

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

tak — Translation Access Key

Format: (*qp*) tak $r_1 = r_3$ M46

Description: The protection key for a given virtual address is obtained and placed in GR r_1.

When PSR.dt is 1, the DTLB and the VHPT are searched for the virtual address specified by GR r_3 and the region register indexed by GR r_3 bits {63:61}. If a matching present translation is found the protection key of the translation is placed in GR r_1. If a matching present translation is not found or if an unimplemented virtual address is specified by GR r_3, the value 1 is returned.

When PSR.dt is 0, only the DTLB is searched, because the VHPT walker is disabled. If no matching present translation is found in the DTLB, the value 1 is returned.

A translation with the NaTPage attribute is not treated differently and returns its key field.

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

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS|TAK;
    check_target_register(r1);

    if (PSR.cpl != 0)
        privileged_operation_fault(itype);

    if (GR[r3].nat)
        register_nat_consumption_fault(itype);

    if (PSR.vm == 1)
        virtualization_fault();

    GR[r1] = tlb_access_key(GR[r3], itype);
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Virtualization fault

thash — Translation Hashed Entry Address

Format: (*qp*) thash $r_1 = r_3$ M46

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

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

When the processor is configured to use the region-based short format VHPT (PTA.vf=0), the value returned by `thash` is defined by the architected short format hash function. See Section 4.1.5.3, “Region-based VHPT Short Format” on page 60

When the processor is configured to use the long format VHPT (PTA.vf=1), `thash` performs an implementation-specific long format hash function on the virtual address to generate a hash index into the long format VHPT.

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

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

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

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

Operation:

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

    if (PSR.vm == 1)
        virtualization_fault();

    if (GR[r3].nat || unimplemented_virtual_address(GR[r3],
PSR.vm)) {
        GR[r1] = undefined();
        GR[r1].nat = 1;
    } else {
        tmp_vr = GR[r3]{63:61};
        tmp_va = GR[r3]{60:0};
        GR[r1] = tlb_vhpt_hash(tmp_vr, tmp_va, RR[tmp_vr].rid,
                               RR[tmp_vr].ps);
        GR[r1].nat = 0;
    }
}
```

Interruptions: Illegal Operation fault Virtualization fault

tpa — Translate to Physical Address

Format: (*qp*) tpa $r_1 = r_3$ M46

Description: The physical address for the virtual address specified by GR r_3 is obtained and placed in GR r_1.

When PSR.dt is 1, the DTLB and the VHPT are searched for the virtual address specified by GR r_3 and the region register indexed by GR r_3 bits {63:61}. If a matching present translation is found the physical address of the translation is placed in GR r_1. If a matching present translation is not found the appropriate TLB fault is taken.

When PSR.dt is 0, only the DTLB is searched, because the VHPT walker is disabled. If no matching present translation is found in the DTLB, an Alternate Data TLB fault is raised if psr.ic is one or a Data Nested TLB fault is raised if psr.ic is zero.

If this instruction faults, then it will set the non-access bit in the ISR. The ISR read and write bits are not set.

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

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS|TPA;
    check_target_register(r1);

    if (PSR.cpl != 0)
        privileged_operation_fault(itype);

    if (GR[r3].nat)
        register_nat_consumption_fault(itype);

    if (PSR.vm == 1)
        virtualization_fault();

    GR[r1] = tlb_translate_nonaccess(GR[r3], itype);
    GR[r1].nat = 0;
}
```

Interruptions:

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

ttag — Translation Hashed Entry Tag

Format: (*qp*) ttag $r_1 = r_3$ M46

Description: A tag used for matching during searches of the long format Virtual Hashed Page Table (VHPT) is generated and placed in GR r_1. The virtual address is specified by GR r_3 and the region register selected by GR r_3 bits {63:61}.

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

The tag generation function generates an implementation-specific long format VHPT tag. The tag generation function must use all implemented region bits and only virtual address bits {60:0}. PTA.vf is ignored by this instruction.

A translation in the long format VHPT must be uniquely identified by its hash index generated by the `thash` instruction and the tag produced from this instruction.

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

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

Operation:

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

    if (PSR.vm == 1)
        virtualization_fault();

    if (GR[r3].nat || unimplemented_virtual_address(GR[r3],
PSR.vm)) {
        GR[r1] = undefined();
        GR[r1].nat = 1;
    } else {
        tmp_vr = GR[r3]{63:61};
        tmp_va = GR[r3]{60:0};
        GR[r1] = tlb_vhpt_tag(tmp_va, RR[tmp_vr].rid,
RR[tmp_vr].ps);
        GR[r1].nat = 0;
    }
}
```

Interruptions: Illegal Operation fault Virtualization fault

vmsw — Virtual Machine Switch

Format:

vmsw.0	zero_form	B8
vmsw.1	one_form	B8

Description: This instruction sets the PSR.vm bit to the specified value. This instruction can be used to implement transitions to/from virtual machine mode without the overhead of an interruption.

If instruction address translation is enabled and the page containing the `vmsw` instruction has access rights equal to 7, then the new value is written to the PSR.vm bit. In the zero_form, PSR.vm is set to 0, and in the one_form, PSR.vm is set to 1.

Instructions after the `vmsw` instruction in the same instruction group may be executed with the old or new value of PSR.vm. Instructions in subsequent instruction groups will be executed with PSR.vm equal to the new value.

If the above conditions are not met, this instruction takes a virtualization fault.

This instruction can only be executed at the most privileged level. This instruction cannot be predicated.

Implementation of PSR.vm is optional. If it is not implemented, this instruction takes Illegal Operation fault. If it is implemented but is disabled, this instruction takes a virtualization fault when executed at the most privileged level. See Section 3.4, “Processor Virtualization” on page 40 and PAL_PROC_GET_FEATURES on page 385 for details.

Operation:

```
if (!implemented_vm())
    illegal_operation fault();

if (PSR.cpl != 0)
    privileged_operation_fault(0);

if (!(PSR.it == 1 && itlb_ar() == 7) || vm_disabled())
    virtualization_fault();

if (zero_form) {
    PSR.vm = 0;
}
else {
    PSR.vm = 1;
}
```

Interruptions:

Illegal Operation fault	Virtualization fault
Privileged Operation fault	

5 Processor Abstraction Layer

Note: This section and all of its subsections track directly to Section 11, Processor Abstraction Layer, in the *Intel® Itanium® Architecture Software Developer's Manual*.

5.1 Virtualization Terminology

The following are terms related to Itanium architecture virtualization:

Virtual Machine Monitor (VMM) – The VMM is the system software which implements software policies to manage/support virtualization of processor and platform resources.

Virtual Processor Descriptor (VPD) – Represents the abstraction of the processor resources of a single virtual processor. The Virtual Processor Descriptor (VPD) consists of per-virtual-processor control information together with performance-critical architectural state. See Section 5.2.1, "Virtual Processor Descriptor (VPD)" on page 52 for details.

Virtual Processor State – A memory data structure which represents the architectural state of a virtual processor. Part of the virtual processor state is located in the VPD, and the rest is located in memory data structures maintained by the virtual machine monitor.

PAL intercepts – Interfaces where PAL transfers control to the VMM on virtualization events (execution of virtualized instructions/operations with PSR.vm==1). For details see Section 5.2.3, "PAL Intercepts in Virtual Environment" on page 58.

5.2 PAL Virtualization Support

This section describes the PAL architectural support for Itanium architecture virtualization.

Itanium architecture processors that support processor virtualization, the PAL virtualization support described in this document will be available. Itanium architecture virtualization support can be determined by calling PAL_PROC_GET_FEATURES.

The virtualization support in PAL presents an implementation-independent interface to enable the VMM to implement software policies to manage/support virtualization of Itanium processors.

The PAL extensions for virtualization consist of three main components:

1. A set of procedures to support virtualization operations. These procedures allow the VMM to configure logical processors for virtualization operations and suspend/resume virtual processors on logical processors. Details for this component are described in Section 5.5, "PAL Procedures for Virtualization" on page 84.
2. A set of services to provide low-latency, low-overhead support for performance-critical VMM operations. Details for this component are described in Section 5.2.5, "PAL Virtualization Services" on page 70.
3. A PAL intercept interface to allow PAL to deliver virtualization events to the VMM in a low-latency, low-overhead manner. This PAL-to-VMM interface also allows PAL to provide

optimizations for VMM operations. Details for this component are described in Section 5.2.3, "PAL Intercepts in Virtual Environment" on page 58.

The VMM is responsible for managing the set of available system resources (CPU, memory, peripherals) and implement policies to virtualize these resources. In order to support virtual processor operations, the VMM will create a **virtual environment** and associate logical processors with the virtual environment. A virtual environment consists of one or more logical processors plus the memory resource allocated by the VMM during PAL_VP_INIT_ENV.

The VMM creates a virtual environment by calling PAL_VP_ENV_INFO to obtain the memory requirement for creating a virtual environment, and then by calling PAL_VP_INIT_ENV on each logical processor that is to be part of the virtual environment. After a virtual environment is created, the VMM can create and initialize virtual processors to run in the environment by calling PAL_VP_CREATE.

The state of a virtual processor belonging to a virtual environment can be restored/saved on a logical processor in the environment by calling PAL_VP_RESTORE or PAL_VP_SAVE respectively. The VMM starts virtual processor operations on a logical processor by invoking either PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER.

The VMM can add/remove a logical processor from a virtual environment at any time by calling PAL_VP_INIT_ENV or PAL_VP_EXIT_ENV respectively.

5.2.1 Virtual Processor Descriptor (VPD)

The Virtual Processor Descriptor (VPD) represents the abstraction of processor resources of a single virtual processor. The VPD consists of per-virtual-processor control information together with performance-critical architectural state. The VPD is 64K in size and the base must be 32K aligned. Table 5-1 shows the fields and layout of the VPD. The values in the VPD can be stored in little or big endian format, depending on the setting of *be* field setting in "config_options – Global Configuration Options" during PAL_VP_INIT_ENV call. See "PAL Initialize Virtual Environment" on page 89 for details. The VPD is divided into two classes – the first class stores control information and the second class stores the performance-critical architectural state of the virtual processor.

The VMM must keep the virtual processor state in the VPD for a particular state entry either: always, or only when one or more particular accelerations is enabled, as described in the Class column of Table 5-1. See Section 5.2.4.1, "Virtualization Accelerations" on page 61 for details.

Table 5-1. Virtual Processor Descriptor (VPD) (Sheet 1 of 3)

Name	Entries	Offset	Description	Class
vac	1	0	Virtualization Acceleration Control – these control bits enable virtualization acceleration of a particular resource or instruction. See Section 5.2.1.1, "Virtualization Controls" on page 54 for details.	Control [always]
vdc	1	8	Virtualization Disable Control – these control bits disable the virtualization of a particular resource or instruction. See Section 5.2.1.1, "Virtualization Controls" on page 54 for details.	Control [always]
Reserved	30	16	Reserved Area – Reserved for future expansion.	Reserved

Table 5-1. Virtual Processor Descriptor (VPD) (Sheet 2 of 3)

Name	Entries	Offset	Description	Class
vhpi	1	256	Virtual Highest Priority Pending Interrupt – Specifies the current highest priority pending interrupt for the virtual processor. See Table 5-26, "vhpi – Virtual Highest Priority Pending Interrupt" on page 78 for details.	Control [a_int]
Reserved	95	264	Reserved Area – Reserved for future expansion.	Reserved
vgr[16-31]	16	1024	Virtual General Registers – Represent the bank 1 general registers 16-31 of the virtual processor. When the virtual processor is running and vpsr.bn is 1, the values in these entries are undefined.	Architectural State [a_bsw]
vbgr[16-31]	16	1152	Virtual Banked General Registers – Represent the bank 0 general registers 16-31 of the virtual processor. When the virtual processor is running and vpsr.bn is 0, the values in these entries are undefined.	Architectural State [a_bsw]
vnat	1	1280	Virtual General Register NaTs – Bits 0-15 represent the NaT values corresponding to vgr16-31, where the NaT bit for vgr16 is in bit 0. Bits 16-63 are don't cares.	Architectural State [a_bsw]
vbnat	1	1288	Virtual Banked Register NaTs – Bits 16-31 represent the NaT values corresponding to vbgr16-31, where the NaT bit for vbgr16 is in bit 16. Bits 0-15 and 32-63 are don't cares.	Architectural State [a_bsw]
vcpuid[0-4]	5	1296	Virtual CPUID Registers – Represent cpuid registers 0-4 of the virtual processor. NOTE: vcpuid[0-1] and vcpuid[4]{63:32} must contain the same values as the corresponding values of the logical processor on which this virtual processor is running.	Architectural State [a_from_cpuid]
Reserved	11	1336	Reserved Area – Reserved for future expansion.	Reserved
vpsr	1	1424	Virtual Processor Status Register – Represents the Processor Status Register of the virtual processor.	Architectural State [always[1], a_int[2], a_from_psr, a_from_int_cr[3], a_to_int_cr[3], a_cover[4], a_bsw[5]]
vpr	1	1432	Virtual Predicate Registers – Represents the Predicate Registers of the virtual processor. The bit positions in vpr correspond to predicate registers in the same manner as with the mov predicates instruction.	Architectural State [always]
Reserved	76	1440	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state.	Reserved
vcr[0-127]	128	2048	Virtual Control Registers – Represent the control registers of the virtual processor. For the reserved control registers, the corresponding VPD entries are reserved.	Architectural State [a_int[6], a_from_int_cr[7], a_to_int_cr[7], a_cover[8]]

Table 5-1. Virtual Processor Descriptor (VPD) (Sheet 3 of 3)

Name	Entries	Offset	Description	Class
Reserved	128	3072	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state	Reserved
Reserved	3456	4096	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state	Reserved
vmm_avail	128	31744	Available for VMM use. This area is ignored by the processor and PAL.	Ignored
Reserved	4096	32768	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state	Reserved

NOTES:
1. If the value of the *opcode* field in the *config_options* parameter during PAL_VP_INIT_ENV is 1, then vpsr.ic must be kept in the VPD independent of any accelerations.
2. The a_int acceleration only requires that the vpsr.i bit be kept in the VPD; other bits of the virtual processor's psr need not be kept here.
3. The a_from_int_cr and a_to_int_cr accelerations only require that vpsr.ic be kept in the VPD.
4. The a_cover acceleration only requires that the vpsr.ic bit be kept in the VPD.
5. The a_bsw acceleration only requires that the vpsr.bn bit be kept in the VPD.
6. The a_int acceleration only requires that vtpr be kept in the VPD.
7. The a_from_int_cr and a_to_int_cr accelerations only require that the virtual interruption CRs (vipsr, visr, viip, vifa, vitir, viipa, vifs, viim, viha) be kept in the VPD.
8. The a_cover acceleration only requires that vifs be kept in the VPD.

5.2.1.1 Virtualization Controls

The Virtualization Acceleration Control (*vac*) and Virtualization Disable Control (*vdc*) fields in the VPD contain configuration control bits which define the set of events that will cause an intercept from PAL to the VMM. The virtualization controls are divided into two categories:

1. Virtualization Acceleration Control – these control bits enable virtualization optimization support of a particular resource or instruction. Figure 5-1 and Table 5-2 describe these control bits.
2. Virtualization Disable Control – these control bits disable the virtualization of a particular resource or instruction. Figure 5-2 and Table 5-3 describe these control bits.

The *vac* and *vdc* settings are specified by the VMM during virtual processor initialization when the PAL_VP_CREATE procedure is called, and cannot be changed until the virtual processor is terminated by PAL_VP_TERMINATE.

Figure 5-1. Virtualization Acceleration Control (*vac*)

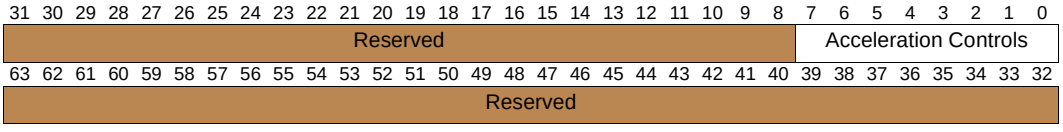


Table 5-2. Virtualization Acceleration Control (*vac*) Fields

Field	Bit	Description
a_int[1]	0	Enable the virtual external interrupt optimization. See Section 5.2.4.1.1, “Virtual External Interrupt Optimization” on page 61 for details.
a_from_int_cr[1]	1	Enable the interruption control register (CR16-25) read optimization. See Section 5.2.4.1.2, “Interruption Control Register Read Optimization” on page 63 for details.
a_to_int_cr[1]	2	Enable the interruption control register (CR16-25) write optimization. See Section 5.2.4.1.3, “Interruption Control Register Write Optimization” on page 64 for details.
a_from_psr[1]	3	Enable the processor status register read optimization. See Section 5.2.4.1.4, “MOV-from-PSR Optimization” on page 65 for details.
a_from_cpuid[1]	4	Enable the CPUID read optimization. See Section 5.2.4.1.5, “MOV-from-CPUID Optimization” on page 65 for details.
a_cover[1]	5	Enable the `cover` instruction optimization. See Section 5.2.4.1.6, “Cover Optimization” on page 66 for details.
a_bsw[1]	6	Enable the `bsw` instruction optimization. See Section 5.2.4.1.7, “Bank Switch Optimization” on page 66 for details.
Reserved	63:7	Reserved

NOTES:
1. The functionality provided by this field is not available if the value of the *opcode* field in the *config_options* parameter during PAL_VP_INIT_ENV is 0. For details see Table 5-27, “vp_env_info – Virtual Environment Information Parameter” on page 87.

Figure 5-2. Virtualization Disable Control (*vdc*)

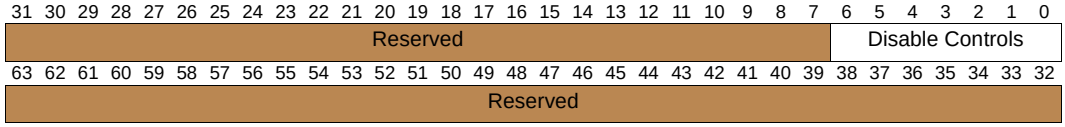


Table 5-3. Virtualization Disable Control (*vdc*) Fields (Sheet 1 of 2)

Field	Bits	Description
d_vmsw	0	Disable `vmsw` instruction – If 1, disables `vmsw` instruction on the logical processor. Execution of the `vmsw` instruction, independent of the state of PSR.vm, will cause a virtualization intercept.
d_extint[1]	1	Disable external interrupt control register virtualization – If 1, accesses (reads/writes) of the external interrupt control registers (CR65-71) are not virtualized. Code running with PSR.vm==1 can read and write the external interrupt control registers of the logical processor directly and without handling off to the VMM. See Section 5.2.4.2.9, “Disable External Interrupt Control Register Virtualization” on page 68 for details.
d_ibr_dbr[1]	2	Disable breakpoint register virtualization – If 1, accesses (reads/writes) of the data and instruction breakpoint registers (IBR/DBR) are not virtualized. Code running with PSR.vm==1 can read and write the data/instruction breakpoint registers of the logical processor directly and without handling off to the VMM. If 0, accesses of the breakpoint registers with PSR.vm==1 result in virtualization intercepts.
d_pmc[1]	3	Disable PMC virtualization – If 1, accesses (reads/writes) of the performance monitor configuration registers (PMCs) are not virtualized. Code running with PSR.vm==1 can read and write the performance monitor configuration registers of the logical processor directly and without handling off to the VMM. If 0, accesses of the performance counter configuration registers with PSR.vm==1 result in virtualization intercepts.

Table 5-3. Virtualization Disable Control (*vdc*) Fields (Sheet 2 of 2)

Field	Bits	Description
d_to_pmd[1]	4	Disable PMD write virtualization – If 1, writes to the performance monitor data registers (PMDs) are not virtualized. Code running with PSR.vm==1 can write the performance monitor data registers of the logical processor directly and without handling off to the VMM. If 0, writes of the performance counter data registers with PSR.vm==1 result in virtualization intercepts.
d_itm[1]	5	Disable ITM virtualization – If 1, writes to the Interval Timer Match (ITM) register are not virtualized. Code running with PSR.vm==1 can write the ITM register of the logical processor directly and without handling off to the VMM. If 0, writes of the ITM register with PSR.vm==1 result in virtualization intercepts.
d_psr_i[1]	6	Disable PSR.i virtualization – If 1, accesses (reads/writes) to the interrupt bit in processor state register (PSR.i) are not virtualized. Code running with PSR.vm==1 can read and write only the interrupt bit via the `ssm` and `rsm` instructions directly without handling off to the VMM. Attempts to modify other PSR bits in addition to the interrupt bit via the `ssm` and `rsm` instructions will result in virtualization intercepts. Attempts to modify the interrupt bit with the `mov psr.l` instruction will continue to result in virtualization intercepts. If 0, accesses to the PSR.i bit with PSR.vm==1 result in virtualization intercepts.
Reserved	63:7	Reserved

NOTES:
1. The functionality provided by this field is not available if the value of the *opcode* field in the *config_options* parameter during PAL_VP_INIT_ENV is 0. For details see Table 5-27, "vp_env_info – Virtual Environment Information Parameter" on page 87.

5.2.2 Interruption Handling in a Virtual Environment

For logical processors which have been added to a virtual environment through PAL_VP_INIT_ENV, all IVA-based interruptions continue to be delivered to the **host IVT** independent of the state of PSR.vm at the time of interruption. All IVA-based interruptions are serviced by the host IVT pointed to by the IVA (CR2) control register on the logical processor.

IVA-based interruptions that do not represent virtualization events will be delivered to the **guest IVT** by the VMM. The guest IVT is specified by the VIVA control register in the VPD of the virtual processor.

For IVA-based interruption handling during virtual processor operations, PAL provides maximum flexibility to the VMM by supporting **per-virtual-processor host IVTs**. This allows the VMM to provide a different host IVT with optimizations specific to a particular guest operating system on the virtual processor. The VMM can also choose to provide the same IVT for some or all of the virtual processors in a virtual environment.

Hence, at any time in a virtual environment, the IVA (CR2) control register of the logical processor will be pointing to either:

- The per-virtual-processor host IVT.
- The generic host IVT not specific to any virtual processor.

The per-virtual-processor host IVT for each virtual processor is setup by PAL when the virtual processor is first created (PAL_VP_CREATE) or registered (PAL_VP_REGISTER) in the virtual environment. The VMM passes a pointer to the host IVT specific to the virtual processor as an incoming parameter to the PAL_VP_CREATE or PAL_VP_REGISTER procedures. The per-virtual-processor host IVT is setup to perform long branches to the corresponding vector of the

IVT specified in the incoming parameter for all IVA-based interruptions except the Virtualization vector. Virtualization vector will be delivered as virtualization intercept in the per-virtual-processor host IVT. See Section 5.2.3, “PAL Intercepts in Virtual Environment” on page 58 for details on PAL intercepts.

In the virtual environment, the IVA (CR2) control register will be set by PAL virtualization-related procedures and services as summarized in Table 5-4.

Table 5-4. IVA Settings after PAL Virtualization-Related Procedures and Services

PAL Virtualization-related Procedure / Service	Description
PAL_VP_CREATE	These procedures do not change the IVA control register.
PAL_VP_ENV_INFO	
PAL_VP_EXIT_ENV	This procedure sets the IVA control register to point to the IVT specified by the caller.
PAL_VM_INIT_ENV	These procedures do not change the IVA control register.
PAL_VP_REGISTER	
PAL_VP_RESTORE / PAL_VPS_RESTORE	This procedure / service sets the IVA control register to point to the per-virtual-processor host IVT.
PAL_VP_SAVE / PAL_VPS_SAVE	This procedure / service does not change the IVA control register.
PAL_VP_TERMINATE	This procedure sets the IVA control register to point to the IVT specified by the caller.

After successful execution of PAL_VP_RESTORE procedure or PAL_VPS_RESTORE service, the IVA control register on the logical processor is set to point to the per-virtual-processor host IVT. After successful completion of PAL_VP_RESTORE procedure, the VMM must not change the IVA control register on the logical processor until after the next invocation of PAL_VP_SAVE or PAL_VPS_SAVE.

On IVA-based interruptions when a virtual processor is running (after PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER), the IVA control register on the logical processor is unchanged and will continue to point to the per-virtual-processor host IVT. On resume execution to the same virtual processor through PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER PAL services, the VMM must ensure the IVA control register on the logical processor is set to point to the per-virtual-processor host IVT at the time of interruption.[1]

Faults and virtualization intercepts on the following instructions can be handled in two ways, determined by the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV:

- mov-from-interruption-CR (CRs 16, 17, 19-25)
- mov-to-interruption-CR (CRs 16, 17, 19-25)
- `itc.d, itc.i`
- `itr.d, itr.i`

These instructions can raise one or more of these faults:

- Illegal Operation fault
- Privileged Operation fault

1. In other words, the VMM is allowed to change to another IVT after IVA-based interruptions happening during virtual processor execution. The VMM must ensure the per-virtural processor IVT is restored before resuming to the same virtual processor through PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER.

- Reserved Register/Field fault
- Unimplemented Data Address fault
- Register NaT Consumption fault

If the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1 when these instructions execute, the above faults may be raised at the General Exception vector of the host IVT based on the state of the virtual processor. If none of the above faults are raised, a virtualization intercept is raised at the Virtualization vector or at the optional Virtualization Intercept handler specified by the VMM, and there is no need for the VMM to check for the above faults at the virtualization intercept handler.

If the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 0, these instructions are delivered at the General Exception vector of the host IVT, with ISR indicating an Illegal Operation fault. In this case, the VMM is responsible to determine whether any of the above faults have caused by these instructions based on the state of the virtual processor before any handling code for these instructions.

5.2.3 PAL Intercepts in Virtual Environment

When the IVA control register on the logical processor is set to point to the per-virtual-processor host IVT, virtualization intercepts will be raised at the Virtualization vector or at an optional virtualization intercept handler specified by the VMM. By default, virtualization intercepts are delivered to the Virtualization vector of the IVT specified by the VMM during PAL_VP_CREATE / PAL_VP_REGISTER. If the VMM specified the optional virtualization intercept handler, all virtualization intercepts are delivered to that handler (instead of the Virtualization vector.)

Section 5.2.3.1, “PAL Virtualization Intercept Handoff State” on page 58 describes the handoff state of the PAL intercepts. For all interruption vectors other than Virtualization vector, the architectural state at the corresponding IVA-based interruption vector is the same as defined in Chapter 8, “Interruption Vector Descriptions” in Volume 2.

5.2.3.1 PAL Virtualization Intercept Handoff State

The state of the logical processor at virtualization intercept handoff is:

- GRs:
  - — Non-banked GRs: The contents of non-banked general registers are preserved from the time of the interruption.
  - — Bank 1 GRs: The contents of all bank one general registers are preserved from the time of the interruption.
  - — Bank 0: GR16-23: The contents of these bank zero general registers are preserved from the time of the interruption.
  - — Bank 0: GR24-31: Scratch, contains parameters/state for VMM:
    - GR24 indicates the cause of virtualization intercept. See Table 5-5, “PAL Virtualization Intercept Handoff Cause (GR24)” for details. This field is not provided to the VMM if the value of the *cause* field in the *config_options* parameter passed to PAL_VP_INIT_ENV is 0. If the value of the *cause* field in the *config_options* parameter passed to PAL_VP_INIT_ENV is 0, the value of GR24 on virtualization intercept handoff is undefined.
    - GR25 contains the 41-bit opcode in little endian format and the type of the instruction which caused the fault, excluding the qualifying predicate (qp) field. See Figure 5-3,

"PAL Virtualization Intercept Handoff Opcode (GR25)," on page :60 for details. This field is not provided to the VMM if the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV is 0. If the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV is 0, the value of GR25 on virtualization intercept handoff is undefined.

- GR26-31 are available for the VMM to use.

- FRs: The contents of all floating-point registers are preserved from the time of the interruption.
- Predicates: The contents of all predicate registers are undefined and available for use. The original contents are saved in the VPD.
- BRs: The contents of all branch registers are preserved from the time of the interruption.
- ARs: The contents of all application registers are preserved from the time of the interruption, except the ITC counter. The ITC register will not be directly modified by PAL, but will continue to count during the execution of the virtualization intercept handler.
- CFM: The contents of the CFM register is preserved from the time of the interruption.
- RSE: All RSE state is preserved from the time of the interruption.
- PSR: PSR fields are set according to the "Interruption State" column in Table 3-2, "Processor Status Register Fields" on page 2:23.
- CRs: The contents of all control registers are preserved from the time of the interruption with the exception of resources described below:
  - IRRs: The contents of IRRs are not changed by PAL. Incoming interruptions may change the contents.
  - IFS: IFS is unchanged from the time of the interruption.
  - IIP: Contains the value of IP at the time of the interruption.
  - IPSR: Contains the value of PSR at the time of the interruption.
- RRs: The contents of all region registers are preserved from the time of the interruption.
- PKRs: The contents of all protection key registers are preserved from the time of the interruption.
- DBRs/IBRs: The contents of all breakpoint registers are preserved from the time of the interruption.
- PMCs/PMDs: The contents of the PMC registers are preserved from the time of the virtualization intercept. The contents of the PMD registers are not modified by PAL code, but may be modified if events being monitored are encountered. The performance counters will be frozen if specified by the VMM through a parameter of PAL_VP_INIT_ENV procedure.
- Cache: The processor internal cache is not specifically modified by PAL handler but may be modified due to normal cache activity of running the handler code.
- TLB: The TRs are unchanged from the time of the interruption.

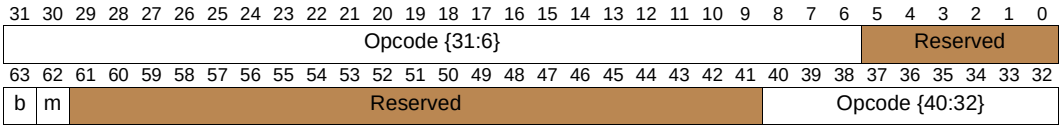
Table 5-5. PAL Virtualization Intercept Handoff Cause (GR24) (Sheet 1 of 2)

Value	Cause	Description
1	toAR	Due to MOV-to-AR instruction.
2	toARimm	Due to MOV-to-AR-imm instruction.
3	fromAR	Due to MOV-from-AR instruction.
4	toCR	Due to MOV-to-CR instruction.

Table 5-5. PAL Virtualization Intercept Handoff Cause (GR24) (Sheet 2 of 2)

Value	Cause	Description
5	fromCR	Due to MOV-from-CR instruction.
6	toPSR	Due to MOV-to-PSR instruction.
7	fromPSR	Due to MOV-from-PSR instruction.
8	itc_d	Due to `itc.d` instruction.
9	itc_i	Due to `itc.i` instruction.
10	toRR	Due to MOV-to-RR instruction.
11	toDBR	Due to MOV-to-DBR instruction.
12	toIBR	Due to MOV-to-IBR instruction.
13	toPKR	Due to MOV-to-PKR instruction.
14	toPMC	Due to MOV-to-PMC instruction.
15	toPMD	Due to MOV-to-PMD instruction.
16	itr_d	Due to `itr.d` instruction.
17	itr_i	Due to `itr.i` instruction.
18	fromRR	Due to MOV-from-RR instruction.
19	fromDBR	Due to MOV-from-DBR instruction.
20	fromIBR	Due to MOV-from-IBR instruction.
21	fromPKR	Due to MOV-from-PKR instruction.
22	fromPMC	Due to MOV-from-PMC instruction.
23	fromCPUID	Due to MOV-from-CPUID instruction.
24	ssm	Due to `ssm` instruction.
25	rsm	Due to `rsm` instruction.
26	ptc_l	Due to `ptc.l` instruction.
27	ptc_g	Due to `ptc.g` instruction.
28	ptc_ga	Due to `ptc.ga` instruction.
29	ptr_d	Due to `ptr.d` instruction.
30	ptr_i	Due to `ptr.i` instruction.
31	thash	Due to `thash` instruction.
32	ttag	Due to `ttag` instruction.
33	tpa	Due to `tpa` instruction.
34	tak	Due to `tak` instruction.
35	ptc_e	Due to `ptc.e` instruction.
36	cover	Due to `cover` instruction.
37	rfi	Due to `rfi` instruction.
38	bsw_0	Due to `bsw.0` instruction.
39	bsw_1	Due to `bsw.1` instruction.
40	vmsw	Due to `vmsw` instruction.
All other values	Reserved	Reserved for future expansion.

Figure 5-3. PAL Virtualization Intercept Handoff Opcode (GR25)



5.2.4 Virtualization Optimizations

After the PAL_VP_INIT_ENV procedure is called, execution of the virtualized instructions listed in Table 3-10, “Virtualized Instructions” on page 2:40 with PSR.vm==1 results in virtualization intercepts to the VMM. Virtualization optimizations allow these instructions to execute, with PSR.vm==1, without causing intercepts to the VMM. Virtualization optimizations are divided into two classes:

- Virtualization accelerations – Virtualization accelerations optimize the execution of virtualized instructions by supporting fast access to the virtual instance of the resource and perform the virtualized operations based on the virtual instance of the resource without handling off to the VMM. Section 5.2.4.1, “Virtualization Accelerations” on page 61 describes the supported Virtualization accelerations in the architecture.
- Virtualization disables – Virtualization disables optimize the execution of virtualized instructions by disabling virtualization of a particular resource or instruction. Accesses to the virtualization-disabled resources or executions of virtualization-disabled instructions, even with PSR.vm==1, will not cause intercepts to the VMM. Section 5.2.4.2, “Virtualization Disables” on page 67 describes the supported Virtualization disables in the architecture.

5.2.4.1 Virtualization Accelerations

Table 5-6 summarizes the virtualization accelerations supported in Itanium architecture.

Table 5-6. Virtualization Accelerations Summary

Optimization	Virtualization Acceleration Control (*vac*)[1]	Description
Virtual External Interrupt Optimization	a_int	Section 5.2.4.1.1
Interruption Control Register Read Optimization	a_from_int_cr	Section 5.2.4.1.2
Interruption Control Register Write Optimization	a_to_int_cr	Section 5.2.4.1.3
MOV-from-PSR Optimization	a_from_psr	Section 5.2.4.1.4
MOV-from-CPUID Optimization	a_from_cpuid	Section 5.2.4.1.5
Cover Optimization	a_cover	Section 5.2.4.1.6
Bank Switch Optimization	a_bsw	Section 5.2.4.1.7

NOTES:
1. The Virtualization Acceleration Control (*vac*) field resides in the Virtual Processor Descriptor (VPD), see Section 5.2.1, “Virtual Processor Descriptor (VPD)” on page 52 for details.

For each of the accelerations, certain virtual processor control and architectural state is managed directly by hardware/firmware, and hence must be maintained in the VPD, and synchronization is required when the VMM reads or writes this state in the VPD. Some entries must be maintained in the VPD independent of any accelerations. (These are marked as [always].) See Table 5-1 for details on which VPD state is used with each of the accelerations. See Section 5.2.5, “PAL Virtualization Services” on page 70 for a description of the synchronization services.

5.2.4.1.1 Virtual External Interrupt Optimization

The virtual external interrupt optimization allows the VMM to specify the virtual highest priority pending interrupt so that a virtual external interrupt is raised on changes of vtpr or vpsr.i only when that the virtual highest priority pending interrupt is unmasked. For details on virtual external interrupts, see “Virtual External Interrupt vector (0x3400)” on page 2:183.

The virtual external interrupt optimization is enabled by the a_int bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, the VMM specifies the virtual highest priority pending interrupt (vhpi) through the PAL_VPS_SET_PENDING_INTERRUPT service described in Section 5.2.5, "PAL Virtualization Services" on page 70. If this optimization is disabled, processor behavior is undefined if PAL_VPS_SET_PENDING_INTERRUPT is invoked.

When this optimization is enabled, execution of `rsm` and `ssm` instructions, with PSR.vm==1, which modify only vpsr.i will not intercept to the VMM and vpsr.i is updated with the new value, unless a fault condition is detected (see Table 5-9 for details). A virtual external interrupt is raised if the virtual highest priority pending interrupt (vhpi) is unmasked by the new vpsr.i and vtpr. If the virtual highest priority pending interrupt (vhpi) is still masked by the new vpsr.i or vtpr, no virtual external interrupt will be raised. Note that execution of MOV-to-PSR instructions with PSR.vm==1 always results in a virtualization intercept no matter which PSR bits are modified.

When this optimization is enabled, execution of `rsm` and `ssm` instructions, with PSR.vm==1, which modify any bits in addition to vpsr.i result in a virtualization intercepts. No virtual external interrupts are raised and the VMM is responsible for delivering a virtual external interrupt if the virtual highest priority pending interrupt (vhpi) is unmasked.

When this optimization is enabled, execution of a MOV-from-CR instruction, with PSR.vm==1, targeting vtpr reads the most recent value, unless a fault condition is detected (see Table 5-9 for details).

When this optimization is enabled, on execution of MOV-to-TPR instructions with PSR.vm==1, vtpr will be updated with the new value without handling off to the VMM, unless a fault condition is detected (see Table 5-9 for details). A virtual external interrupt is raised if the virtual highest priority pending interrupt (vhpi) is unmasked by the new vpsr.i and vtpr. No virtual external interrupt is raised if the virtual highest priority pending interrupt is still masked by vpsr.i or vtpr.

When this optimization is enabled, after completion of an instruction with PSR.vm==1 which modifies vtpr or vpsr.i (if the instruction completes without an intercept), a determination is made as to whether the new state unmasks the virtual highest priority pending interrupt. If it does, then a virtual external interrupt will be raised and the VMM will be entered on the Virtual External Interrupt vector. See Table 5-7 for details on the detection of virtual external interrupts.

Table 5-7. Detection of Virtual External Interrupts

Condition	Virtual External Interrupt
vhpi <= (!vpsr.i << 5 | vtpr.mmi <<4 | vtpr.mic)	No – virtual highest priority pending interrupt is still masked.
vhpi > (!vpsr.i << 5 | vtpr.mmi <<4 | vtpr.mic)	Yes – virtual highest priority pending interrupt is unmasked.

This optimization is available only if the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1.

Synchronization is required when this optimization is enabled, see Table 5-8 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 5-1, "Virtual Processor Descriptor (VPD)" on page 52.

Table 5-8. Synchronization Requirements for Virtual External Interrupt Optimization

VPD Resource	Synchronization Required
vtpr	Read, Write
vpsr.i	Read, Write
vhpi	Write

Table 5-9. Interruptions when Virtual External Interrupt Optimization is Enabled

Instructions	Interruptions
`rsm`, `ssm`	When the virtual external interrupt optimization is enabled, execution of `rsm` and `ssm` instructions with PSR.vm==1 which modify only vpsr.i, may raise the following faults: • Privileged Operation fault – if vpsr.cpl is not zero.
MOV-from-TPR	When the virtual external interrupt optimization is enabled, execution of MOV-from-CR instruction targeting vtpr with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if the target operand specifies GR 0 or an out-of-frame stacked register. • Privileged Operation fault – if vpsr.cpl is not zero.
MOV-to-TPR	When the virtual external interrupt optimization is enabled, execution of MOV-to-CR instruction targeting vtpr with PSR.vm==1, may raise the following faults: • Privileged Operation fault – if vpsr.cpl is not zero. • Register NaT Consumption fault – if the NaT bit in the source register is one. • Reserved Register/Field fault – if the reserved field in the vtpr is being written with a non-zero value.

5.2.4.1.2 Interruption Control Register Read Optimization

The interruption control register read optimization is enabled by the a_from_int_cr bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, and vpsr.ic is 0, software running with PSR.vm==1 will be able to read the virtual interruption control registers (vipsr, visr, viip, vifa, vitir, viipa, vifs, viim, viha) without any intercepts to the VMM, unless a fault condition is detected (see Table 5-11 for details).

If this optimization is disabled, a read of the interruption CRs with PSR.vm==1 results in a virtualization intercept.

This optimization is available only if the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1.

Synchronization is required when this optimization is enabled, see Table 5-10 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 5-1, “Virtual Processor Descriptor (VPD)” on page 52.

Table 5-10. Synchronization Requirements for Interruption Control Register Read Optimization

VPD Resource	Synchronization Required
vipsr, visr, viip, vifa, vitir, viipa, vifs, viim, viha	Write

Table 5-11. Interruptions When Interruption Control Register Read Optimization is Enabled

Instructions	Interruptions
Move from interruption control registers	When the interruption control register read optimization is enabled, reads of interruption control registers with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if vpsr.ic is not zero or the target operand specifies GR 0 or an out-of-frame stacked register • Privileged Operation fault – if vpsr.cpl is not zero

5.2.4.1.3 Interruption Control Register Write Optimization

The interruption control register write optimization is enabled by the a_to_int_cr bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, and vpsr.ic is 0, software running with PSR.vm==1 will be able to write the virtual interruption control registers (vipsr, visr, viip, vifa, vitir, viipa, vifs, viim, viha) without any intercepts to the VMM, unless a fault condition is detected (see Table 5-13 for details).

If this optimization is disabled, a write of the interruption control registers with PSR.vm==1 results in a virtualization intercept.

This optimization is available only if the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1.

Synchronization is required when this optimization is enabled, see Table 5-12 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 5-1, “Virtual Processor Descriptor (VPD)” on page 52.

Table 5-12. Synchronization Requirements for Interruption Control Register Write Optimization

VPD Resource	Synchronization Required
vipsr, visr, viip, vifa, vitir, viipa, vifs, viim, viha	Read

Table 5-13. Interruptions when Interruption Control Register Write Optimization is Enabled

Instructions	Interruptions
Move to interruption control registers	When the interruption control register write optimization is enabled, writes to interruption control registers with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if vpsr.ic is not zero. • Privileged Operation fault – if vpsr.cpl is not zero. • Register NaT Consumption fault – if the NaT bit of the source operand is one. • Reserved Register/Field fault – if any reserved field in the specified control register is written with a non-zero value. • Unimplemented Data Address fault – if writing to vifa and an unimplemented virtual address is specified.

5.2.4.1.4 MOV-from-PSR Optimization

The MOV-from-PSR optimization is enabled by the a_from_psr bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, software running with PSR.vm==1 will be able to execute MOV-from-PSR instructions to read the virtual processor status register without any intercepts to the VMM; and the last value written to the vpsr will be returned, unless a fault condition is detected (see Table 5-15 for details). The value returned for the fml, mfh, ac, up and be bits are simply the values of those bits in the PSR of the logical processor, since those bits are not virtualized.

If this optimization is disabled, execution of a MOV-from-PSR instruction with PSR.vm==1 results in a virtualization intercept.

This optimization is available only if the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1.

Synchronization is required when this optimization is enabled, see Table 5-14 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 5-1, "Virtual Processor Descriptor (VPD)" on page 52.

Table 5-14. Synchronization Requirements for MOV-from-PSR Optimization

VPD Resource	Synchronization Required
vpsr	Write

Table 5-15. Interruptions when MOV-from-PSR Optimization is Enabled

Instructions	Interruptions
MOV-from-PSR	When the MOV-from-PSR optimization is enabled, MOV-from-PSR instructions with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if the target operand specifies GR 0 or an out-of-frame stacked register. • Privileged Operation fault – if vpsr.cpl is not zero.

5.2.4.1.5 MOV-from-CPUID Optimization

The MOV-from-CPUID optimization is enabled by the a_from_cpuid bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, software running with PSR.vm==1 will be able to execute MOV-from-CPUID instruction to read the virtual CPUID registers without any intercepts to the VMM; and the corresponding VCPUID value from the VPD will be returned, unless a fault condition is detected (see Table 5-17 for details).

If this optimization is disabled, execution of a MOV-from-CPUID instruction with PSR.vm==1 results in a virtualization intercept.

This optimization is available only if the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1.

Synchronization is required when this optimization is enabled, see Table 5-16 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 5-1, "Virtual Processor Descriptor (VPD)" on page 52.

Table 5-16. Synchronization Requirements for MOV-from-CPUID Optimization

VPD Resource	Synchronization Required
vcpuid0-4	Write

Table 5-17. Interruptions when MOV-from-CPUID Optimization is Enabled

Instructions	Interruptions
MOV-from-CPUID	When the MOV-from-CPUID optimization is enabled, MOV-from-CPUID instructions with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if the target operand specifies GR 0 or an out-of-frame stacked register. • Register NaT Consumption fault – if the NaT bit in the target register is one. • Reserved Register/Field fault – if a reserved CPUID register is being read.

5.2.4.1.6 Cover Optimization

The cover optimization is enabled by the a_cover bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, software running with PSR.vm==1 will be able to execute `cover` instructions without any intercepts to the VMM, unless a fault condition is detected (see Table 5-19 for details). The `cover` instruction will execute and vcr.ifs will be updated if vpsr.ic is 0.

If this optimization is disabled, execution of the `cover` instruction with PSR.vm==1 results in a virtualization intercept.

This optimization is available only if the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1.

Synchronization is required when this optimization is enabled, see Table 5-18 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 5-1, “Virtual Processor Descriptor (VPD)” on page 52.

Table 5-18. Synchronization Requirements for Cover Optimization

VPD Resource	Synchronization Required
vifs	Read, Write

Table 5-19. Interruptions when Cover Optimization is Enabled

Instructions	Interruptions
`cover`	When the cover optimization is enabled, `cover` instructions with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if the instruction is not the last instruction in an instruction group.

5.2.4.1.7 Bank Switch Optimization

The bank switch optimization is enabled by the a_bsw bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, execution of the `bsw` instruction with PSR.vm==1 spills the currently active banked registers and the corresponding NaT bits to the

VPD, and loads the other banked registers and the corresponding NaT bits from the VPD. vpsr.bn is updated to reflect the new register bank without any intercepts to the VMM, unless a fault condition is detected (see Table 5-21 for details).

If this optimization is disabled, execution of the `bsw` instruction with PSR.vm==1 results in a virtualization intercept.

This optimization is available only if the value of the *opcode* field in the *config_options* parameter passed to PAL_VP_INIT_ENV is 1.

This optimization requires no special synchronization.

Table 5-20. Interruptions When Bank Switch Optimization is Enabled

Instructions	Interruptions
`bsw`	When the bank switch optimization is enabled, `bsw` instructions with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if the instruction is not the last instruction in an instruction group • Privileged Operation fault – if vpsr.cpl is not zero

5.2.4.2 Virtualization Disables

Table 5-6 summarizes the virtualization disables supported in Itanium architecture.

Table 5-21. Virtualization Disables Summary

Disable	Virtualization Disable Control (*vdc*)[1]	Description
Disable `VMSW` Instruction	d_vmsw	Section 5.2.4.2.8
Disable External Interrupt Control Register Virtualization	d_extint	Section 5.2.4.2.9
Disable Breakpoint Register Virtualization	d_ibr_dbr	Section 5.2.4.2.10
Disable PMC Virtualization	d_pmc	Section 5.2.4.2.11
Disable MOV-to-PMD Virtualization	d_to_pmd	Section 5.2.4.2.12
Disable ITM Virtualization	d_itm	Section 5.2.4.2.13
Disable PSR Interrupt-bit Virtualization	d_psr_i	Section 5.2.4.2.14

NOTES:
1. The Virtualization Disable Control (*vdc*) field resides in the Virtual Processor Descriptor (VPD), see Section 5.2.1, “Virtual Processor Descriptor (VPD)” on page 52 for details.

5.2.4.2.8 Disable `vmsw` Instruction

The `vmsw` instruction disable is controlled by the d_`vmsw` bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, the `vmsw` instruction is disabled on the logical processor. Execution of the `vmsw` instruction, independent of the state of PSR.vm, results in a virtualization intercept.

If this control is set to 0, the `vmsw` instruction can be executed by both the VMM and guest without virtualization intercepts, if PSR.it is 1 and the `vmsw` instruction is executed on a page with access rights of 7.

5.2.4.2.9 Disable External Interrupt Control Register Virtualization

The external interrupt control register virtualization disable is controlled by the d_extint bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, the external interrupt control registers (CR65-71) are not virtualized, and code running with PSR.vm==1 can read and write these resources directly without any intercepts to the VMM.

If this control is set to 0, accesses (reads/writes) to the external interruption control registers with PSR.vm==1 result in virtualization intercepts.

The functionality provided by this field is not available if the value of the *opcode* field in the *config_options* parameter during PAL_VP_INIT_ENV is 0.

5.2.4.2.10 Disable Breakpoint Register Virtualization

The breakpoint register virtualization disable is controlled by the d_ibr_dbr bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, accesses (reads/writes) to the data and instruction breakpoint registers (DBR/IBR) are not virtualized, and code running with PSR.vm==1 can read and write these resources directly without any intercepts to the VMM.

If this control is set to 0, accesses (reads/writes) to the breakpoint registers with PSR.vm==1 result in virtualization intercepts.

The functionality provided by this field is not available if the value of the *opcode* field in the *config_options* parameter during PAL_VP_INIT_ENV is 0.

5.2.4.2.11 Disable PMC Virtualization

The PMC virtualization disable is controlled by the d_pmc bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, accesses (reads/writes) to the performance monitor configuration registers (PMCs) are not virtualized, and code running with PSR.vm==1 can read and write these resources directly without any intercepts to the VMM.

If this control is set to 0, accesses (reads/writes) to the performance counter configuration registers with PSR.vm==1 result in virtualization intercepts.

The functionality provided by this field is not available if the value of the *opcode* field in the *config_options* parameter during PAL_VP_INIT_ENV is 0.

5.2.4.2.12 Disable MOV-to-PMD Virtualization

The MOV-to-PMD[1] virtualization disable is controlled by the d_to_pmd bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, writes to the performance monitor data registers (PMDs) are not virtualized, and code running with PSR.vm==1 can write these resources directly without any intercepts to the VMM.

If this control is set to 0, writes to the performance monitor data registers with PSR.vm==1 result in virtualization intercepts.

The functionality provided by this field is not available if the value of the *opcode* field in the *config_options* parameter during PAL_VP_INIT_ENV is 0.

1. The MOV-from-PMD instruction is not virtualzied. Hence there is no need to provide optimizations for the MOV-from-PMD instruction.

5.2.4.2.13 Disable ITM Virtualization

The ITM virtualization disable is controlled by the d_itm bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, writes to the Interval Timer Match (ITM) register are not virtualized, and code running with PSR.vm==1 can write this resource directly without any intercepts to the VMM.

If this control is set to 0, writes to the ITM register with PSR.vm==1 result in virtualization intercepts.

The functionality provided by this field is not available if the value of the *opcode* field in the *config_options* parameter during PAL_VP_INIT_ENV is 0.

5.2.4.2.14 Disable PSR Interrupt-bit Virtualization

The PSR interrupt-bit virtualization disable is controlled by the d_psr_i bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, accesses (reads/writes) to the interrupt bit in processor state register (PSR.i) are not virtualized. Code running with PSR.vm==1 can read and write to PSR.i through `ssm` and `rsm` instructions without any intercepts to the VMM. Attempts to modify other PSR bits in addition to the interrupt bit via the `ssm` and `rsm` instructions will result in virtualization intercepts.

This control has no effect on `mov psr.l` instructions; attempts to modify the interrupt bit with the mov psr.l instruction result in virtualization intercepts.

The functionality provided by this field is not available if the value of the *opcode* field in the *config_options* parameter during PAL_VP_INIT_ENV is 0.

Note: This field overrides the a_int Virtualization Acceleration Control (*vac*) described in Section 5.2.4.1.1, “Virtual External Interrupt Optimization” on page 61. If this control is enabled (set to 1), the a_int Virtualization Acceleration Control (*vac*) is ignored.

5.2.4.3 Virtualization Synchronizations

When certain virtualization accelerations described in Section 5.2.4.1, “Virtualization Accelerations” on page 61 are enabled, processor implementations can provide implementation-specific control resources to enhance the performance of virtual processors. Two PAL services are provided to synchronize the implementation-specific control resources and the resources in the VPD. There are two types of synchronizations:

1. **Read Synchronization** – When a specific acceleration is enabled, after interruptions and intercepts that occur when PSR.vm was 1, the VMM must invoke PAL_VPS_SYNC_READ to synchronize the related resources before reading their values from the VPD.
2. **Write Synchronization** – When a specific acceleration is enabled, the VMM must invoke PAL_VPS_SYNC_WRITE to synchronize the related resources after modifying their values in the VPD and before resuming the virtual processor.

For details on PAL_VPS_SYNC_READ and PAL_VPS_SYNC_WRITE, see Section 5.2.5, “PAL Virtualization Services” on page 70.

Read and/or write synchronizations are required only if the specific acceleration is enabled. For the resources that require synchronizations if the acceleration is enabled, failure to perform the proper synchronizations will result in undefined processor behavior[1].

1. Virtual machine monitors must perform all the required synchronizations specified. Virtual machine monitors not conforming to this specification are not guaranteed to work on all processor implementations.

The synchronization requirements of the related resources for each acceleration are described in the corresponding sections for each acceleration in Section 5.2.4.1, “Virtualization Accelerations” on page 61.

No synchronization is required for any of the virtualization disables.

5.2.5 PAL Virtualization Services

In order to support efficient handling of interruptions when PSR.vm was 1, a set of PAL virtualization services is defined to allow certain frequent PAL functions to be performed in a low-latency and low-overhead manner.

Upon successful completion of PAL_VP_INIT_ENV, the virtual base address of the PAL virtualization services (VSA) is returned to the VMM. VMM can invoke PAL services by branching to the defined offsets from the virtual base address. See Table 5-22 for the defined services. See Section 5.4, “PAL Virtualization Services Specification” on page 72 for details on PAL virtualization services.

These PAL virtualization services will only make references to the PAL virtual environment buffer. The VMM is required to maintain the ITR and DTR translations of the PAL virtual environment buffer during any PAL virtualization service calls.

Table 5-22. PAL Virtualization Services

Offset	PAL Service
0x0000	PAL_VPS_RESUME_NORMAL
0x0400	PAL_VPS_RESUME_HANDLER
0x0800	PAL_VPS_SYNC_READ
0x0c00	PAL_VPS_SYNC_WRITE
0x1000	PAL_VPS_SET_PENDING_INTERRUPT
0x1400	PAL_VPS_THASH
0x1800	PAL_VPS_TTAG
0x1c00	PAL_VPS_RESTORE
0x2000	PAL_VPS_SAVE
All other offsets	Reserved

5.2.5.1 PAL Virtualization Service Invocation Convention

This section describes the required parameters applicable to all PAL Virtualization Services. Additional parameters are listed in the description section of specific PAL Virtualization Services. Architectural state not listed in this section is managed by the VMM and can contain both VMM and/or virtual processor state. The architectural state not listed is unchanged by PAL virtualization services.

The state of the processor on handing off to any PAL Virtualization Service is:

- GR24-31: Parameters for PAL virtualization services.
- BRs:
  - — BR0: Scratch, the VMM will use BR0 to specify the 64-bit host virtual address of the PAL Virtualization Service being invoked.
- Predicates: The predicates are preserved by the PAL virtualization services.

- PSR State (see Table 5-23 for details):
  - PSR.be, i, cpl, is, ss, db, tb, vm must be 0.
  - PSR.dt, rt and it must be 1.
  - All other values are don't cares.

Table 5-23. State Requirements for PSR for PAL Virtualization Services (Sheet 1 of 2)

PSR Bit	Description	Value
be	big-endian memory access enable	-[1]
up	user performance monitor enable	-
ac	alignment check	-
mfl	floating-point registers f2-f31 written	-
mfh	floating-point registers f32-f127 written	-
ic	interruption state collection enable	0[2]
		-[3]
i	interrupt enable	0
pk	protection key validation enable	-
dt	data address translation enable	1
dfl	disabled FP register f2 to f31	-
dfh	disabled FP register f32 to f127	-
sp	secure performance monitors	-
pp	privileged performance monitor enable	-
di	disable ISA transition	-
si	secure interval timer	-
db	debug breakpoint fault enable	0
lp	lower-privilege transfer trap enable	-
tb	taken branch trap enable	0
rt	register stack translation enable	1
cpl	current privilege level	0
is	instruction set	0
mc	machine check abort mask	-
it	instruction address translation enable	1
id	instruction debug fault disable	-
da	data access and dirty-bit fault disable	-
dd	data debug fault disable	-
ss	single step trap enable	0
ri	restart instruction	-
ed	exception deferral	-
bn	register bank	-[4]
		0[5]

Table 5-23. State Requirements for PSR for PAL Virtualization Services (Sheet 2 of 2)

PSR Bit	Description	Value
ia	instruction access-bit fault disable	-
vm	processor virtualization	0

NOTES:
1. PAL services can be called with PSR.be bit equal to 0 or 1. The behavior is undefined if PSR.be setting does not match the *be* parameter during PAL_VP_INIT_ENV. See "PAL Initialize Virtual Environment" on page 89 for details.
2. Most PAL services are invoked with PSR.ic equal to 0.
3. Specific PAL services can be invoked with PSR.ic equal to 1 or 0. See the description of specific PAL services for details.
4. Most PAL services can be invoked with PSR.bn equal to 1 or 0.
5. Specific PAL services must be invoked with PSR.bn equal to 0. See the description of specific PAL services for details.

5.3 PAL Procedure Summary

Table 5-24. PAL Virtualization Support Procedures

Procedure	Idx	Class	Conv.	Mode	Description
PAL_VP_CREATE	265	Opt.	Stacked	Virt.	Initializes a new VPD for the operation of a new virtual processor in the virtual environment.
PAL_VP_ENV_INFO	266	Opt.	Stacked	Virt.	Returns the parameters needed to enter a virtual environment.
PAL_VP_EXIT_ENV	267	Opt.	Stacked	Virt.	Allows a logical processor to exit a virtual environment.
PAL_VP_INIT_ENV	268	Opt.	Stacked	Virt.	Allows a logical processor to enter a virtual environment.
PAL_VP_REGISTER	269	Opt.	Stacked	Virt.	Register a different host IVT for the virtual processor.
PAL_VP_RESTORE	270	Opt.	Stacked	Virt.	Restore virtual processor state on the logical processor.
PAL_VP_SAVE	271	Opt.	Stacked	Virt.	Save virtual processor state on the logical processor.
PAL_VP_TERMINATE	272	Opt.	Stacked	Virt.	Terminates operation for the specified virtual processor.

5.4 PAL Virtualization Services Specification

The following pages provide detailed interface specifications for each of the PAL Virtualization Services.

PAL_VPS_RESUME_NORMAL – Resume Virtual Processor Normal (0x0000)

Purpose: Resumes the current virtual processor. This service is used when vpsr.ic is 1. This service can also be used independently of the state of vpsr.ic if all virtualization accelerations and disables are disabled.

Arguments:

Argument	Description
GR24	VBR0
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Reserved
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns: PAL_VPS_RESUME_NORMAL does not return to the VMM.

Description: On interruptions or intercepts, PAL_VPS_RESUME_NORMAL allows the VMM to resume the same virtual processor where the vpsr.ic is 1. PAL_VP_RESTORE can be used to restore the state of a different virtual processor.

The VMM specifies the VBR0 of the virtual processor in GR24 and the 64-bit virtual pointer to the VPD in GR25.

The VMM is responsible for setting up all the required virtual processor state in the architectural registers as well as in the VPD prior to invoking this service. See Table 5-25, “Virtual Processor Settings in Architectural Resources for PAL_VPS_RESUME_NORMAL and PAL_VPS_RESUME_HANDLER” on page 73 for details.

PAL_VPS_RESUME_NORMAL must be called with PSR.bn equal to 0.

If all virtualization accelerations and disables are disabled, PAL_VPS_RESUME_NORMAL can also be used to resume to the guest independent on the state of vpsr.ic.

Table 5-25. Virtual Processor Settings in Architectural Resources for PAL_VPS_RESUME_NORMAL and PAL_VPS_RESUME_HANDLER (Sheet 1 of 2)

Resource	Description
Bank 1 GRs	Contains state of bank 0/1 GRs of the virtual processor (depends on vpsr.bn.)
FRs	Contains floating-point register state of the virtual processor.
Predicate Register	Contains the predicates of the virtual processor.
Branch Registers	BR1-BR7 contains the state of the virtual processor. BR0 of the virtual processor resides in bank 0 GR24.
Application Registers	Contains application register state of the virtual processor.
Interruption Control Registers	IIP, IPSR and IFS contains the IP, PSR and CFM of the virtual processor. The rest of the interruption control registers are don’t cares. For PAL_VPS_RESUME_HANDLER, the virtual interruption control registers are specified in the VPD. See Section 5.2.4, “Virtualization Optimizations” on page 61 for synchronization of VPD resources before resuming the virtual processor.

Table 5-25. Virtual Processor Settings in Architectural Resources for PAL_VPS_RESUME_NORMAL and PAL_VPS_RESUME_HANDLER (Sheet 2 of 2)

Resource	Description
External Interrupt Control Registers	The external interrupt control registers contain the state of the virtual processor if d_extint in Virtualization Disable Control (*vdc*) is 1. Otherwise the external interrupt control registers are virtualized by the VMM and contain VMM state.
Data/Instruction Breakpoint Registers	The data/instruction breakpoint registers contain the state of the virtual processor if d_ibr_dbr in Virtualization Disable Control (*vdc*) is 1. Otherwise the data/instruction breakpoint registers are virtualized by the VMM and contain VMM state.
Performance Monitor Configuration Registers	The performance monitor configuration registers contain the state of the virtual processor if d_pmc in Virtualization Disable Control (*vdc*) is 1. Otherwise the performance monitor configuration registers are virtualized by the VMM and contain VMM state.
Performance Monitor Data Registers	Contain the state of the virtual processor.

PAL_VPS_RESUME_NORMAL performs the following actions:

- Perform any implementation-specific setup to run a virtual processor.
- Re-enable performance counters if the value of the *fr_pmc* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1.
- Resume the virtual processor.

PAL_VPS_RESUME_HANDLER – Resume Virtual Processor Handler (0x0400)

Purpose: Resumes the current virtual processor. This service is used when vpsr.ic is 0.

Arguments:

Argument	Description
GR24	VBR0
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Virtualization Acceleration Control (*vac*) field from the VPD specified in GR25
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns: PAL_VPS_RESUME_HANDLER does not return to the VMM.

Description: On interruptions or intercepts, PAL_VPS_RESUME_HANDLER allows the VMM to resume to the same virtual processor where the vpsr.ic is 0[1].

The VMM specifies the BR0 of the virtual processor in GR24, the 64-bit virtual pointer to the VPD in GR25 and the *vac* field of the VPD in GR26. Behavior is undefined if the *vac* in GR26 does not match the *vac* field in the VPD argument specified in GR25.

The VMM is responsible for setting up all the required virtual processor state in the architectural registers as well as in the VPD prior to invoking this service. See Table 5-25, “Virtual Processor Settings in Architectural Resources for PAL_VPS_RESUME_NORMAL and PAL_VPS_RESUME_HANDLER” on page 73 for details.

PAL_VPS_RESUME_HANDLER must be called with PSR.bn equal to 0.

PAL_VPS_RESUME_HANDLER performs the following actions:

- Perform any implementation-specific setup to run a virtual processor.
- Re-enable performance counters if the value of the *fr_pmc* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1.
- Resume the virtual processor.

1. PAL_VP_RESTORE can be used to restore the state of a different virtual processor.

PAL_VPS_SYNC_READ – Synchronize VPD State for Reads (0x0800)

Purpose: Synchronize VPD with the latest implementation-specific virtual architectural state.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Reserved
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Description: On processor implementations that support virtualization accelerations, implementation-specific control resources can be provided to enhance performance of virtual processors. When a specific acceleration is enabled, after interruptions and intercepts which occur when PSR.vm was 1, the VMM must invoke this service to synchronize the related resources before reading the value from the VPD. For the accelerations that are disabled, the corresponding resources in the VPD are unchanged.

The synchronization requirements of the related resources for each acceleration are described in the corresponding sections for each acceleration in Section 5.2.4.1, “Virtualization Accelerations” on page 61.

PAL_VPS_SYNC_READ performs the following actions:

- Copy implementation-specific control resources of the enabled accelerations into VPD.
- Return to VMM by an indirect branch specified in the GR24 parameter.

PAL_VPS_SYNC_WRITE – Synchronize VPD State for Writes (0x0c00)

Purpose: Synchronize the implementation-specific virtual architectural state with VPD.

Arguments:

Argument	Description
GR24	64-bit host virtual return address.
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD.)
GR26	Reserved
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Description: On processor implementations that support virtualization accelerations, implementation-specific control resources can be provided to enhance performance of virtual processors. When a specific acceleration is enabled, the VMM must invoke this service to synchronize the related resources after modifying the value in the VPD and before resuming the virtual processor. For the accelerations that are disabled, the corresponding resources in the VPD are ignored.

The synchronization requirements of the related resources for each acceleration are described in the corresponding sections for each acceleration in Section 5.2.4.1, “Virtualization Accelerations” on page 61.

PAL_VPS_SYNC_WRITE performs the following actions:

- Copy values of the enabled accelerations in the VPD into implementation-specific control resources.
- Return to VMM by an indirect branch specified in the GR24 parameter.

PAL_VPS_SET_PENDING_INTERRUPT – Register Highest Priority Pending Interrupt (0x1000)

Purpose: Register highest priority pending interrupt of the running virtual processor.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Reserved
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Description: PAL_VPS_SET_PENDING_INTERRUPT allows the VMM to register the highest priority pending interrupt for the virtual processor. The virtual highest priority pending interrupt is specified in the vhpi field in the VPD. See Table 5-26, "vhpi – Virtual Highest Priority Pending Interrupt" on page 78 for details.

PAL_VPS_SET_PENDING_INTERRUPT can be called with PSR.ic equal to 1 or 0.

Table 5-26. *vhpi* – Virtual Highest Priority Pending Interrupt (Sheet 1 of 2)

Value	Description
0	Nothing pending.
1	Class 1 interrupt pending.
2	Class 2 interrupt pending.
3	Class 3 interrupt pending.
4	Class 4 interrupt pending.
5	Class 5 interrupt pending.
6	Class 6 interrupt pending.
7	Class 7 interrupt pending.
8	Class 8 interrupt pending.
9	Class 9 interrupt pending.
10	Class 10 interrupt pending.
11	Class 11 interrupt pending.
12	Class 12 interrupt pending.
13	Class 13 interrupt pending.
14	Class 14 interrupt pending.

Table 5-26. *vhpi* – Virtual Highest Priority Pending Interrupt (Sheet 2 of 2)

Value	Description
15	Class 15 interrupt pending.
16	ExtINT pending.
17-31	Reserved.
32	NMI pending.
33+	Reserved.

PAL_VPS_SET_PENDING_INTERRUPT performs the following actions:

- Copy the virtual highest priority pending interrupt from the VPD into implementation-specific resources.
- Return to VMM by an indirect branch specified in the GR24 parameter.

PAL_VPS_THASH – Compute Long Format VHPT Entry Address (0x1400)

Purpose: Compute a long format VHPT entry address.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit virtual address used to compute the hash entry address
GR26	Region register value used to compute the hash entry address
GR27	Virtual PTA
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	64-bit VHPT entry address

Description: PAL_VPS_THASH computes a long format Virtual Hashed Page Table (VHPT) entry address based on the input arguments and the result is returned in GR31. The format of the region register parameter (GR26) is defined in Section 4.1.2, “Region Registers (RR)” on page 2:55, the ve field is ignored by the service. The format of the Virtual PTA parameter (GR27) is defined in Section 3.3.4.4, “Page Table Address (PTA – CR8)” on page 2:32, the vf field is ignored by the service.

PAL_VPS_THASH returns the same long format VHPT entry address given the same input arguments across different implementations. The long format VHPT entry address returned may not be the same as the long format VHPT entry address generated by the `thash` instruction of the processor.

PAL_VPS_THASH can be called with PSR.ic equal to 1 or 0.

PAL_VPS_TTAG – Compute Translated Hashed Entry Tag (0x1800)

Purpose: Compute the long format translated hashed entry tag.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit virtual address used to compute the hash entry tag
GR26	Region register value used to compute the hash entry tag
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	64-bit VHPT entry tag

Description: PAL_VPS_TTAG computes the tag value of the long format Virtual Hashed Page Table (VHPT) based on the input arguments and the result is returned in GR31. The format of the region register parameter (GR26) is defined in Section 4.1.2, “Region Registers (RR)” on page 2:55, the ve field is ignored by the service.

PAL_VPS_TTAG returns the same tag value given the same input arguments across different implementations. The tag value returned may not be the same as the tag value generated by the `ttag` instruction of the processor.

PAL_VPS_TTAG can be called with PSR.ic equal to 1 or 0.

PAL_VPS_RESTORE – Fast Restore Virtual Processor State (0x1c00)

Purpose: Performs an implementation-specific lightweight restore operation for the specified VPD on the logical processor.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Description: PAL_VPS_RESTORE performs an implementation-specific lightweight restore operation of the virtual processor specified by the VPD parameter (GR25) on the logical processor. The host virtual to host physical translation of the 64K region specified by the VPD parameter (GR25) and the PAL virtual environment buffer must be mapped by instruction and data translation registers (TR). The instruction and data translation must be maintained until after the next invocation of PAL_VP_SAVE or PAL_VPS_SAVE and a different host IVT is set up by the VMM by writing to the IVA control register. PAL_VPS_RESTORE configures the logical processor to run the specified virtual processor by loading the minimal implementation-specific virtual processor context from the VPD, and returns control back to the VMM.

This service performs an implicit PAL_VPS_SYNC_WRITE; there is no need for the VMM to invoke PAL_VPS_SYNC_WRITE unless the VPD values are modified before resuming the virtual processor. After the service, the caller is responsible for restoring all of the architectural state before resuming to the new virtual processor through PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER.

Upon completion of this service, the IVA-based interruptions will be delivered to the host IVT associated with this virtual processor.

This service does not restore any PAL procedure implementation-specific state[1]. The caller of this service is responsible to manage the difference in settings for the PAL procedures between the VMM and virtual processors.

1. PAL_VP_RESTORE can be used to restore PAL procedure implementation-specific state. See "PAL Restore Virtual Processor" on page 93 for details.

PAL_VPS_SAVE – Fast Save Virtual Processor State (0x2000)

Purpose: Performs an implementation-specific lightweight save operation for the specified VPD on the logical processor.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Description: PAL_VPS_SAVE performs an implementation-specific lightweight save operation of the virtual processor specified by the VPD parameter (GR25) on the logical processor. The host virtual to host physical translation of the 64K region specified by the VPD parameter (GR25) must be mapped by instruction and data translation registers (TR).

This service performs an implicit PAL_VPS_SYNC_READ; there is no need for the VMM to invoke PAL_VPS_SYNC_READ to synchronize the implementation-specific control resources before this service.

Upon completion of this service, the IVA-based interruptions will continue to be delivered to the host IVT associated with this virtual processor. After this service, the VMM can setup the IVA control register to use a different host IVT.

This service does not save any PAL procedure implementation-specific state[1]. The caller of this service is responsible to manage the difference in settings for the PAL procedures between the VMM and virtual processors.

1. PAL_VP_SAVE can be used to save PAL procedure implementation-specific state. See "PAL Save Virtual Processor" on page 95 for details.

5.5 PAL Procedures for Virtualization

This section describes the procedures for Itanium architecture virtualization.

PAL Create New Virtual Processor

Purpose: Initializes a new *vpd* for the operation of a new virtual processor in the virtual environment.

Calling Conv: Stacked Registers

Mode: Virtual

Arguments:

Argument	Description
index	Index of PAL_VP_CREATE within the list of PAL procedures
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
host_iva	64-bit host virtual pointer to the host IVT for the virtual processor
opt_handler	64-bit non-zero host-virtual pointer to an optional handler for virtualization intercepts. See Section 5.2.3, “PAL Intercepts in Virtual Environment” on page 58 for details.

Returns:

Return Value	Description
status	Return status of the PAL_VP_CREATE 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 – Indicates internal error in PAL

Description: Initializes a new *vpd* for the operation of a new virtual processor within the virtual environment.

The caller must pass a pointer to the new Virtual Processor Descriptor (*vpd*) as argument. The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped with either a DTR or DTC. See Section 11.9.2.1.3, “Making PAL Procedure Calls in Physical or Virtual Mode” on page 2:312 for details on data translation requirements of memory buffer pointers passed as arguments to PAL procedures. The *vac* and *vdc* parameters in the VPD must already be initialized before calling this procedure.

The *host_iva* parameter specifies the host IVT to handle IVA-based interruptions when this virtual processor is running. The VMM can use the same or different *host_iva* for each virtual processor. The *opt_handler* specifies an optional virtualization intercept handler. If a non-zero value is specified, all virtualization intercepts are delivered to this handler. If a zero value is specified, all virtualization intercepts are delivered to the Virtualization vector in the host IVT. If the VMM relocates the IVT specified by the *host_iva* parameter and/or the virtualization intercept handler specified by the *opt_handler* parameter after this procedure, PAL_VP_REGISTER must be called to register the new host IVT and virtualization intercept handler before resuming virtual processor execution or allowing any IVA-based interruptions to occur; otherwise processor operation is undefined.

Upon return, the VMM is responsible for setting up the rest of the VMD state before the new virtual processor is launched (via PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER).

PAL Virtual Environment Information

Purpose: Returns the parameters needed to enter a virtual environment.

Calling Conv: Stacked Registers

Mode: Virtual

Arguments:

Argument	Description
index	Index of PAL_VP_ENV_INFO within the list of PAL procedures
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_ENV_INFO procedure
buffer_size	Unsigned integer denoting the number of bytes required by the PAL virtual environment buffer during PAL_VP_INIT_ENV
vp_env_info	64-bit vector of virtual environment information. See Table 5-27. for details
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error – Indicates internal error in PAL

Description: This procedure returns the configuration options and the PAL virtual environment buffer size required by PAL_VP_INIT_ENV. This procedure is used by the VMM to setup a virtual environment and determine the amount of memory / resources required. The VMM can then allocate the required amount of physical memory, set up the virtual to physical instruction and data translations that cover the PAL virtual environment buffer in TRs and call PAL_VP_INIT_ENV. The buffer allocated must be at least 4K aligned.

On a multiprocessor system, this procedure need only be invoked once (on any one logical processor) to obtain virtual environment information.

Table 5-27. *vp_env_info* – Virtual Environment Information Parameter

Field	Bit	Description
Reserved	7:0	Reserved
opcode	8	If 1, hardware support to provide opcode information during PAL intercepts is available. If 1, and if the opcode field of the config_options parameter to PAL_VP_INIT_ENV is set to 1, then the opcode (and the decoding of cause) passed as parameters to the VMM on intercept will represent the instruction that triggered the intercept. If 0, opcode information during PAL intercepts is provided by PAL. If 0, and if the opcode field of the config_options parameter to PAL_VP_INIT_ENV is set to 1, then the opcode (and the decoding of cause) passed as parameters to the VMM on intercept will not necessarily represent the instruction that triggered the intercept, but may represent some value that was written to memory between the time the instruction that triggered the intercept was fetched, and when the intercept was triggered.
Reserved	63:9	Reserved

PAL Exit Virtual Environment

Purpose: Allows a logical processor to exit a virtual environment.

Calling Conv: Stacked Registers

Mode: Virtual

Arguments:

Argument	Description
index	Index of PAL_VP_EXIT_ENV within the list of PAL procedures
iva	Optional 64-bit host virtual pointer to the IVT when this procedure is done
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_EXIT_ENV 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 – Indicates internal error in PAL

Description: This procedure allows a logical processor to exit a virtual environment.

Upon successful execution of the PAL_VP_EXIT_ENV procedure and if the *iva* parameter is non-zero, the IVA control register will contain the value from the *iva* parameter.

On a multiprocessor system, the VMM must allow the last logical processor in this environment to complete the procedure before freeing the memory resource allocated to the virtual environment.

PAL Initialize Virtual Environment

Purpose: Allows a logical processor to enter a virtual environment.

Calling Conv: Stacked Registers

Mode: Virtual

Arguments:

Argument	Description
index	Index of PAL_VP_INIT_ENV within the list of PAL procedures
config_options	64-bit vector of global configuration settings - See Table 5-28. for details
pbase_addr	Host physical base address of a block of contiguous physical memory for the PAL virtual environment buffer - This memory area must be allocated by the VMM and be 4K aligned. The first logical processor to enter the environment will initialize the physical block for virtualization operations.
vbase_addr	Host virtual base address of the corresponding physical memory block for the PAL virtual environment buffer - The VMM must maintain the host virtual to host physical data and instruction translations in TRs for addresses within the allocated address space. Logical processors in this virtual environment will use this address when transitioning to virtual mode operations.

Returns:

Return Value	Description
status	Return status of the PAL_VP_INIT_ENV procedure
vsa_base	Virtualization Service Address – VSA specifies the virtual base address of the PAL virtualization services in this virtual environment.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument – Invalid incoming arguments/environment
-3	Call completed with error – Indicates internal error in PAL.

Description: This procedure allows a logical processor to enter a virtual environment. This call must be made after calling PAL_VP_ENV_INFO and before calling other PAL virtualization procedures and services. All of the logical processors in a virtual environment share the same **PAL virtual environment buffer**. The buffer must be 4K aligned. The first logical processor entering the virtual environment initializes the buffer provided by the VMM. Subsequent processors can enter the virtual environment at any time and will not perform initialization to the buffer.

PAL_VP_ENV_INFO must be called before this procedure to determine the configuration options and size requirements for the virtual environment. The VMM is required to maintain the ITR and DTR translations of the PAL virtual environment buffer throughout this procedure. See "PAL Virtual Environment Information" on page 86 for more information on PAL_VP_ENV_INFO.

After this procedure, it is optional for the VMM to maintain the TR mapping for the PAL virtual environment buffer. If the TR translations for the buffer are not installed, the VMM must not make any PAL virtualization service calls; and the VMM must be prepared to handle DTLB faults during any PAL virtualization procedure calls.

Table 5-28 shows the layout of the *config_options* parameter. The *config_options* parameter configures the global configuration options for all the logical processors in the virtual environment. All logical processors in the virtual environment must specify the same configuration options in the *config_options* parameter, otherwise processor operation is undefined.

Table 5-28. *config_options* – Global Configuration Options

Field	Bit	Description
initialize	0	If 1, this procedure will initialize the PAL virtual environment buffer for this virtual environment. If 0, this procedure will not initialize the PAL virtual environment buffer. On a multiprocessor system, the VMM must wait until this procedure completes on the first logical processor before calling this procedure on additional logical processors; otherwise processor operation is undefined.
fr_pmc	1	If 1, performance counters are frozen on all IVA-based interruptions when virtual processors are running. If 0, the performance counters will not be frozen on IVA-based interruptions when virtual processors are running.
be	2	Big-endian – Indicates the endian setting of the VMM. If 1, the values in the VPD are stored in big-endian format and the PAL services calls are made with PSR.be bit equals to 1. If 0, the values in the VPD are stored in little-endian format and the PAL services calls are made with PSR.be bit equals to 0.
Reserved	7:3	Reserved.
opcode	8	If 1, opcode information will be provided to the VMM during PAL intercepts within the virtual environment. This opcode may or may not be guaranteed to be the opcode that triggered the intercept. See Table 5-27, “vp_env_info – Virtual Environment Information Parameter” on page 87 for details. If 0, most virtualization optimizations cannot be enabled through the virtualization acceleration control (*vac*) and virtualization disable control (*vdc*) fields in the VPD. For details on specific optimizations supported in *vac* and *vdc*, see Table 5-2, “Virtualization Acceleration Control (vac) Fields” on page 55 and Table 5-3, “Virtualization Disable Control (vdc) Fields” on page 55. The value of this field also determines how virtualization events and General Exception faults are delivered to the VMM on certain instructions. See Section 5.2.2, “Interruption Handling in a Virtual Environment” on page 56 for details.
cause	9	If 1, the causes of virtualization intercepts will be provided to the VMM during PAL intercept handoffs within the virtual environment. No information will be provided if 0. If this field is 1, the *opcode* field also be 1, otherwise processor operation is undefined. See Section 5.2.3.1, “PAL Virtualization Intercept Handoff State” on page 58 for details of virtualization intercept handoffs.
Reserved	63:10	Reserved.

The *fr_pmc* bit in the global *config_options* parameter specifies whether the performance counters will be frozen when the Virtualization optimizations specified in the Virtualization Acceleration Control (*vac*) and Virtualization Disable Control (*vdc*) are running. When a virtual processor is running, the *vac* field in the corresponding VPD specifies whether a certain virtualization accelerations are enabled. If the *fr_pmc* in the virtual environment was also enabled, the performance counters will be frozen when the enabled virtualization optimizations are running. See Section 5.2.4, “Virtualization Optimizations” on page 61 for details on Virtualization Acceleration Control (*vac*) and Virtualization Disable Control (*vdc*).

PAL Register Virtual Processor

Purpose: Register a different host IVT and/or a different optional virtualization intercept handler for the virtual processor specified by *vpd*.

Calling Conv: Stacked Registers

Mode: Virtual

Arguments:

Argument	Description
index	Index of PAL_VP_REGISTER within the list of PAL procedures
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
host_iva	64-bit host virtual pointer to the host IVT for the virtual processor
opt_handler	64-bit non-zero host-virtual pointer to an optional handler for virtualization intercepts. See Section 5.2.3, "PAL Intercepts in Virtual Environment" on page 58 for details.

Returns:

Return Value	Description
status	Return status of the PAL_VP_REGISTER 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 – Indicates internal error in PAL

Description: PAL_VP_REGISTER registers a different host IVT and/or a different optional virtualization intercept handler specific to the virtual processor specified by *vpd*. On creation of a virtual processor by PAL_VP_CREATE, the VMM specifies a host IVT specific to the virtual processor. This procedure allows the VMM to specify a host IVT different from the one specified during PAL_VP_CREATE.

The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped with either a DTR or DTC. See Section 11.9.2.1.3, "Making PAL Procedure Calls in Physical or Virtual Mode" on page 2:312 for details on data translation requirements of memory buffer pointers passed as arguments to PAL procedures.

The *host_iva* parameter specifies the host IVT to handle IVA-based interruptions when this virtual processor is running. The VMM can use the same or different *host_iva* for each virtual processor. The *opt_handler* specifies an optional virtualization intercept handler. If a non-zero value is specified, all virtualization intercepts are delivered to this handler. If a zero value is specified, all virtualization intercepts are delivered to the Virtualization vector in the host IVT. Upon completion of this procedure, the VMM must not relocate the IVT specified by the *host_iva* parameter and/or the virtualization intercept handler specified by the *opt_handler* parameter. The VMM can call this procedure again in case it wishes to associate a different host IVT and/or virtualization intercept handler with the virtual processor.

This procedure can be used by the VMM to:

- Relocate the host IVT associated with the virtual processor.
- Specify a different optional virtualization intercept handler for the virtual processor.

PAL Restore Virtual Processor

Purpose: Restores virtual processor state for the specified *vpd* on the logical processor.

Calling Conv: Stacked Registers

Mode: Virtual

Arguments:

Argument	Description
index	Index of PAL_VP_RESTORE within the list of PAL procedures.
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD.)
pal_proc_vector	Vector specifies PAL procedure implementation-specific state to be restored
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_RESTORE 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 – Indicates internal error in PAL.

Description: PAL_VP_RESTORE performs an implementation-specific restore operation of the virtual processor specified by the *vpd* parameter on the logical processor. The host virtual to host physical translation of the 64K region specified by *vpd* and the PAL virtual environment buffer must be mapped by instruction and data translation registers (TR). The instruction and data translation must be maintained until after the next invocation of PAL_VP_SAVE or PAL_VPS_SAVE and a different host IVT is set up by the VMM by writing to the IVA control register. PAL_VP_RESTORE configures the logical processor to run the specified virtual processor by loading implementation-specific virtual processor context from the VPD, and returns control back to the VMM.

The *pal_proc_vector* parameter for PAL_VP_RESTORE allows the VMM to control the PAL procedure implementation-specific state to be saved. Table 5-29 shows the format of *pal_proc_vector*. When a bit is set to 1 in the vector, the implementation-specific state for the corresponding PAL procedures will be restored by PAL_VP_RESTORE. When a bit is set to 0 in the vector, no implementation-specific state will be restored for the corresponding PAL procedures.

Table 5-29. Format of *pal_proc_vector*

Bit	PAL Procedures
0	PAL_PROC_GET_FEATURES, PAL_PROC_SET_FEATURES
1	PAL_GET_PSTATE, PAL_SET_PSTATE
63:2	Reserved

This procedure performs an implicit PAL_VPS_SYNC_WRITE; there is no need for the VMM to invoke PAL_VPS_SYNC_WRITE unless the VPD values are modified before resuming the virtual processor. After the procedure, the caller is responsible for restoring all of the architectural state before resuming to the new virtual processor through PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER.

Upon completion of this procedure, the IVA-based interruptions will be delivered to the host IVT associated with this virtual processor.

PAL Save Virtual Processor

Purpose: Saves virtual processor state for the specified *vpd* on the logical processor.

Calling Conv: Stacked Registers

Mode: Virtual

Arguments:

Argument	Description
index	Index of PAL_VP_SAVE within the list of PAL procedures
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
pal_proc_vector	Vector specifies PAL procedure implementation-specific state to be saved
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_SAVE 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 – Indicates internal error in PAL

Description: PAL_VP_SAVE performs an implementation-specific save operation of the virtual processor specified by the *vpd* parameter on the logical processor. The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped by instruction and data translation registers (TR).

The *pal_proc_vector* parameter for PAL_VP_SAVE allows the VMM to control the PAL procedure implementation-specific state to be saved. Table 5-29 on page 93 shows the format of *pal_proc_vector*. When a bit is set to 1 in the vector, the implementation-specific state for the corresponding PAL procedures will be saved by PAL_VP_SAVE. When a bit is set to 0 in the vector, no implementation-specific state will be saved for the corresponding PAL procedures.

This procedure performs an implicit PAL_VPS_SYNC_READ; there is no need for the VMM to invoke PAL_VPS_SYNC_READ to synchronize the implementation-specific control resources before this procedure.

Upon completion of this procedure, the IVA-based interruptions will continue to be delivered to the host IVT associated with this virtual processor. After this procedure, the VMM can setup the IVA control register to use a different host IVT.

PAL Terminate Virtual Processor

Purpose: Terminates operation for the specified virtual processor.

Calling Conv: Stacked Registers

Mode: Virtual

Arguments:

Argument	Description
index	Index of PAL_VP_TERMINATE within the list of PAL procedures
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
iva	Optional 64-bit host virtual pointer to the IVT when this procedure is done
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_TERMINATE 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 – Indicates internal error in PAL

Description: Terminates operation of the virtual processor specified by *vpd* on the logical processor. The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped by instruction and data translation registers (TR). See Section 11.9.2.1.3, “Making PAL Procedure Calls in Physical or Virtual Mode” on page 2:312 for details on data translation requirements of memory buffer pointers passed as arguments to PAL procedures. All resources allocated for the execution of the virtual machine are freed.

Upon successful execution of PAL_VP_TERMINATE procedure and if the *iva* parameter is non-zero, the IVA control register will contain the value from the *iva* parameter.