Intel® Itanium® Processor Family Interrupt Architecture Guide

March 2003

Document Number: 251350-001

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

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO SALE AND/OR USE OF INTEL PRODUCTS, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

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

The Itanium architecture and IA-32 architecture processors may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available upon request.

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

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

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Copyright © 2003, Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

Contents

1 Introduction 1-1

1.1 About this Manual.. 1-1
1.2 Reference Documents.. 1-1
1.3 Objectives of SAPIC 1-1
1.4 Terminology.. 1-2
1.4.1 Glossary 1-3
1.5 Revision History 1-5

2 SAPIC Interrupt Architecture 2-1

2.1 Platform Interrupt Architecture 2-1
2.2 Intel® Itanium® Architecture Interrupt Delivery Overview 2-2
2.2.1 I/O Interrupts from Remote I/O xAPIC 2-2
2.2.2 Processor Local Interrupts (ITV, CMC, Performance Monitor).............. 2-2
2.2.3 Interprocessor Interrupts 2-3
2.3 Software Model 2-3
2.3.1 Local SAPIC 2-3
2.3.2 I/O xAPIC 2-6
2.3.3 Interprocessor Interrupts 2-10
2.4 Interrupt and Processor Priorities.. 2-10
2.4.1 Batch Processing of Interrupts 2-11
2.4.2 Redirectable Interrupt Delivery 2-12
2.4.3 Interrupt Nesting.. 2-13
2.4.4 Interrupt Masking... 2-14
2.5 Interrupt Handling.. 2-15
2.5.1 Edge-Triggered Interrupts 2-15
2.5.2 Level-Triggered Interrupts 2-15
2.5.3 Greater than 240 Devices 2-18
2.5.4 NMI Handling... 2-19
2.5.5 ExtINT Handling 2-19
2.5.6 Timer Interrupt Handling... 2-19
2.5.7 PMI Handling... 2-19
2.6 8259 Interrupt Controller Support (Optional) 2-19
2.7 SAPIC Memory Map.. 2-20
2.7.1 Processor Interrupt Block 2-21
2.7.2 I/O xAPIC Configuration Space... 2-23

3 Platform Level Implementation... 3-1

3.1 Platform Interrupt Delivery Overview... 3-1
3.2 External I/O Interrupt Delivery 3-1
3.2.1 I/O xAPIC Duties 3-1
3.2.2 Interrupt Transaction on the I/O Bus 3-1
3.2.3 Bridge Controller Duties 3-2
3.2.4 Interrupt Transactions on the System Bus 3-3
3.3 IPI Delivery... 3-3
3.4 Ordering Issues 3-3
3.5 Platform Topology and Routing Issues 3-3

A Differences from APIC ... A-1
A.1 Addressing ... A-1
A.2 Interrupts ... A-1
A.3 Interrupt Delivery ... A-1
A.4 Messages ... A-2
A.5 Processor Support ... A-2
A.6 Registers ... A-2
A.7 Support Disabling ... A-3

Figures

2-1 Sample Implementation of an Intel® Itanium® Architecture-Based Multiprocessor Platform ... 2-1
2-2 Local Vector Registers (CMC Vector, ITV, PMV) ... 2-4
2-3 Local Redirection Register ... 2-4
2-4 I/O xAPIC Redirection Table Entry Definition ... 2-8
2-5 I/O SAPIC Version Register ... 2-9
2-6 Expanded View of the Medium Extended Memory Range ... 2-20
2-7 Default Processor Interrupt Block ... 2-21
2-8 XTP Register Format ... 2-21
2-9 Format of Address to Generate IPIs ... 2-22
2-10 Format of IPI Data Being Stored ... 2-22
3-1 Format of Interrupt Address on I/O Bus ... 3-2
3-2 Format of Interrupt Data on I/O Bus ... 3-2

Tables

2-1 Interrupt Control Registers (Subset) ... 2-4
2-2 External and Internal Register Description ... 2-6

intel®

Introduction 1

1.1 About this Manual

This document is a guide for the Intel® Itanium® architecture Streamlined Advanced Programmable Interrupt Controller (SAPIC). SAPIC is the high performance interrupt architecture for the Itanium architecture.

This guide describes the Itanium architecture SAPIC and platform level implementation considerations. In order for the SAPIC architecture to be used effectively in a platform, the user must be aware of some considerations for hardware and software design. This guide provides many of those important design considerations. Failure to follow the guidelines established in this guide may result in incompatibilities with future Intel products.

This document is intended for platform hardware architects (both component and platform) as well as platform software architects (operating system [OS] and platform firmware). For the benefit of those who are already familiar with the Advanced Programmable Interrupt Controller (APIC) architecture, this document describes some of the differences when programming in the Itanium architecture system environment.

1.2 Reference Documents

For implementation details, refer to the documents listed below. These documents can be downloaded on Intel's Developer Site at http://developer.intel.com:

- *Intel® Itanium™ Processor Hardware Developer's Manual* (Document Number: 248701)
- *Intel® Itanium® 2 Processor Hardware Developer's Manual* (Document Number: 251109)
- *Intel® Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture* (Document Number: 245317)
- *Intel® Itanium® Architecture Software Developer's Manual, Volume 2: System Architecture (*Document Number: 245318)
- *Intel® Itanium® Architecture Software Developer's Manual, Volume 3: Instruction Set Reference* (Document Number: 245319)

The following documents are available from their respective organizations:

- Advanced Configuration and Power Interface Specification v2.0 (available at www.acpi.info)
- PCI Bus Specification v2.2 (available at www.pcisig.com)

1.3 Objectives of SAPIC

When the Itanium architecture was being developed, it was determined that a new interrupt architecture would be necessary to match the performance and scalability goals of the new processor and platform architectures. The goal was to streamline the APIC used with the IA-32 Architecture processors for the Itanium Architecture.

At the platform level, the software view of the interrupt architecture below the processor is identical between SAPIC and APIC; because of this compatibility, the I/O unit is called an I/O xAPIC in this document.

The I/O xAPIC can be easily integrated into the I/O devices. Some advantages of an integrated I/O xAPIC are that the delay incurred when routing interrupts through interrupt controller is eliminated along with the additional cost required to implement another component in the system.

With the APIC serial bus, the interrupts travel over a different path than the normal data which travels through the system bus. This creates an ordering problem that must be overcome through complex logic on the platform. If a data write was followed by an interrupt, it is possible for the interrupt to reach the processor first before the data write takes effect, allowing the processor to access stale data. By using the system bus for interrupt delivery as well as data traffic, the ordering problems can be avoided.

Higher performance is obtained by replacing the APIC serial bus implementation with a system bus interrupt delivery implementation. This interrupt handling capability is therefore scalable with the processor's system bus speed. External I/O interrupts are delivered via the I/O bus, which also allows for a faster delivery to the system bus.

The SAPIC architecture minimizes software context switching overhead, which is inherent with the APIC architecture. It allows for the interrupt service routine (ISR) to process all pending interrupts within the current priority level. This reduces the number of times context switching is performed and the number of repeated interruptions to the processor, thereby improving performance.

Finally, it was an objective of the SAPIC architecture to move from a pin-based interrupt mechanism to a signal-based interrupt mechanism. In order to add more interrupt sources with a pin-based mechanism, more pins must be allocated to accommodate the devices; with a signal-based mechanism, there is no limit to the number of interrupt sources that can reside on a bus.

1.4 Terminology

The following terms are used in this document:

- In the register descriptions, fields containing <rv> or <reserved> are reserved by the architecture. Reserved fields are read as zeros, and writing of a non-zero value results in reserved operand exception (part of the General Exception). Register fields with encodings that are listed as <reserved> should not be used by software.
- The processor is said to *receive* an interrupt if the processor's interrupt pins are asserted or an interrupt message bus transaction containing the processor's unique identifier is detected by the processor.
- After receiving an interrupt, the processor internally *holds* the interrupt pending until the interrupt is acquired by software and placed in service. The interrupt is said to be *pended* when it is received and held by the processor. The processor maintains one interrupt pending indication for each possible unique interrupt, signified by a unique interrupt vector number. An occurrence of an interrupt that is already pending cannot be distinguished from previous occurrences. All occurrences to a given vector are pended in the same internal interrupt pending bit, and are therefore treated as "the same" interrupt occurrence.
- When interrupts are enabled and the highest priority pending interrupt is unmasked (as defined below), the processor accepts the pending interrupt, interrupts the control flow of the processor, and transfers control to the software interrupt handler.

- When software acquires the interrupt vector from the processor, the interrupt is considered *in-service*. The processor then removes the pending indication for the interrupt vector. The processor maintains one in-service indicator for each unique vector number.
- The interrupt remains *in service* until software indicates service for the interrupt has been completed (by writing into the EOI register). The processor then removes the in service indication for the interrupt vector.
- Interrupts are *enabled* when software programs the processor to accept any unmasked interrupt. External interrupts are enabled if Processor Status Register (PSR.i) is 1.
- *Unmasked interrupts* are interrupts of higher priority than the highest priority interrupt vector currently in service (if any) and whose priority level is higher than the specified current priority masking level (as indicated by the Task Priority Register, TPR).

1.4.1 Glossary

8259
: The interrupt controller used by legacy systems.

ACPI
: Advanced Configuration and Power Interface

CMC
: Corrected Machine-Check

EOI
: End of Interrupt

ExtINT
: External Controller Interrupt

ICR
: Interrupt Command Register

INIT
: Initialization Interrupt

IPI
: Interprocessor Interrupt

IPSR
: Interruption Processor Status Register

IRR
: Interrupt Request Register

ITV
: Interval Timer

IVR
: Interrupt Vector Register

LID
: Local ID Register

LRR
: Local Redirection Register

LVR
: Local Vector Register

MIC
: Mask Interrupt Class

MADT
: Multiple APIC Descriptor Table

MCA
: Machine Check Abort

MMI
: Mask Maskable Interrupt

MSI
: Message-Signaled Interrupts

NMI
: Non-Maskable Interrupt

OS
: Operating System

PAL
: Processor Abstraction Layer

PIB
: Processor Interrupt Block

PMI
: Platform Management Interrupt

PMV
: Performance Monitoring Event

PSR
: Processor Status Register

RT
: Redirection Table

RTE
: Redirection Table Entry

SAL
: System Abstraction Layer

SMI
: System Management Interrupt

TPR
: Task Priority Register

XTP
: eXternal Task Priority

Note: The term “Local SAPIC unit” is used throughout this document only to facilitate the description of the architecture. The interrupt architecture is an integral part of the Itanium architecture. The Local SAPIC unit is not a discrete component.

1.5 Revision History

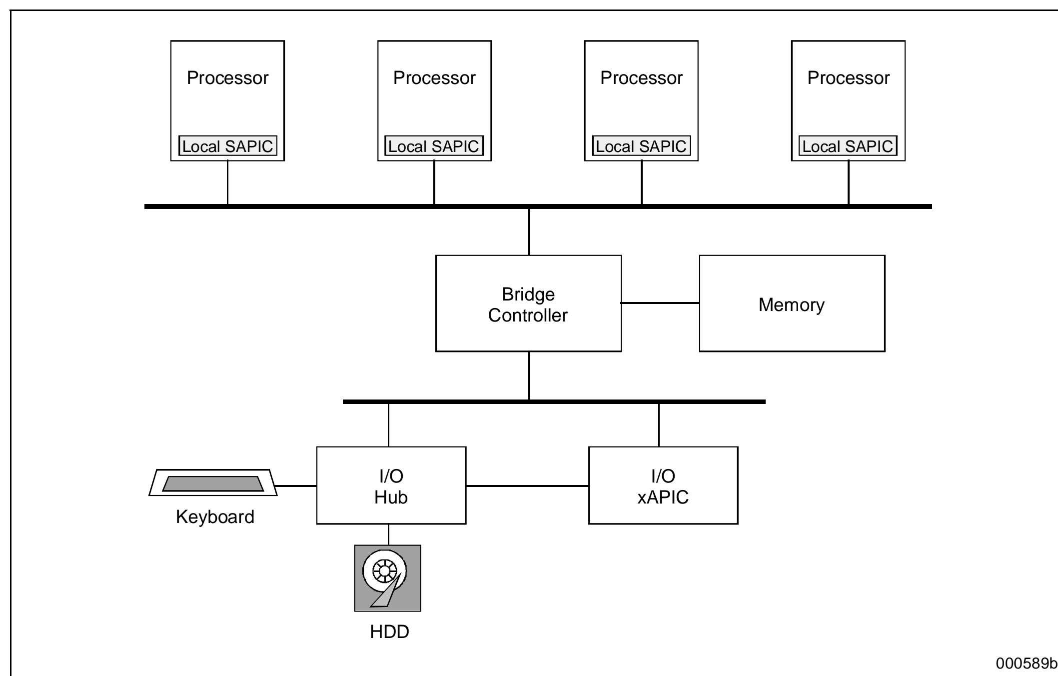
Version	Description	Date
001	Initial release of the document.	March 2003

SAPIC Interrupt Architecture 2

2.1 Platform Interrupt Architecture

The SAPIC has two roles: it is responsible for generating interrupts and for receiving, accepting, and processing the interrupts. Interrupts are asynchronous events that are used by the platform to communicate to the processor and software that something requires their attention. The portion that resides in the processor is known as the Local SAPIC unit, or Local SAPIC, and its primary responsibility is to receive, accept, and process the interrupts. The processor can also generate interrupts of its own (for interprocessor communication as well as for certain interrupts that originate within the processor core). The portion that resides in the I/O subsystem is known as the I/O xAPIC unit, or I/O xAPIC, and its only responsibility is to generate interrupts directed to each selected processor's Local SAPIC unit.

Figure 2-1. Sample Implementation of an Intel® Itanium® Architecture-Based Multiprocessor Platform



The I/O xAPIC unit provides the interrupt input pins on which I/O devices inject interrupts into the system in the form of an edge or a level signal. The I/O xAPIC contains a Redirection Table (RT) with an entry for each interrupt input pin. Each entry in the RT can be individually programmed to indicate whether an interrupt on the pin is recognized as either an edge or a level, what vector (and hence what priority) the interrupt has, and which of all possible processors should service the interrupt. The contents of the RT are under software control and are assigned default values upon reset. The information in the table is used to route a message to the target Local SAPIC unit of a processor via the system bus. The functionality of the I/O xAPIC may be integrated with an I/O device, but any component of the system that is capable of injecting interrupt messages on the I/O bus must appear as an I/O xAPIC to the platform and must have the functionality of the I/O xAPIC.

The Local SAPIC unit determines whether or not its processor should receive interrupts sent on the system bus. The Local SAPIC unit also provides local registry of pending interrupts, nesting, and masking of interrupts, and handles all interactions, such as the INTR/INTA/EOI protocol, with its local processor. The Local SAPIC unit further provides IPIs capability to its local processor. The register level interface of a processor to its Local SAPIC is identical for each processor.

2.2 Intel® Itanium® Architecture Interrupt Delivery Overview

Interrupts received by the processor are generated by a number of different interrupt sources in the system. Possible interrupt sources are:

- Remote I/O xAPIC: Interrupts from these external sources manifest themselves as interrupt messages on the system bus and can be directed to any processor. PCI devices capable of Message-Signaled Interrupts (MSI) fall in this category (refer to the PCI Bus Specification v2.2)
- Local I/O devices: These originate as edge/level signals on interrupt input pins of the processor, but they are directed to the local processor only.
- Local processor-generated interrupts such as Interval Timer (ITV), performance monitoring, and Corrected Machine Checks (CMC); these are directed to the local processor only.
- Processors: A processor can interrupt any individual processor, including itself. This supports software self-interrupts, synchronization, scheduling, interrupt forwarding, etc. These interrupts also manifest themselves as interrupt messages on the system bus and are indistinguishable from other interrupts originating from remote I/O xAPICs.

2.2.1 I/O Interrupts from Remote I/O xAPIC

I/O interrupts from remote I/O devices that are not capable of generating their own interrupt messages (i.e. they generate interrupts via a pin) must be converted to an interrupt message by an I/O xAPIC unit. When an I/O device needs servicing, it sends an interrupt to the external I/O xAPIC via a pin. The I/O xAPIC does a look up in its I/O RT. Each entry in this table is dedicated to one interrupt pin. The I/O xAPIC uses the information in the Redirection Table Entry (RTE) to translate the interrupt signal into an interrupt message. This interrupt message contains the interrupt vector, delivery mode, trigger mode, and destination processor ID. The I/O xAPIC transmits this message along the I/O bus (i.e. in the form of a memory write transaction) to a bridge controller. Next, the bridge controller converts the message into an interrupt transaction and sends it on to the system bus. Each processor on the system bus then checks its own identification ID against the destination ID specified in the interrupt transaction. Only the processor with a match receives the interrupt transaction.

2.2.2 Processor Local Interrupts (ITV, CMC, Performance Monitor)

Besides the responsibility of receiving and processing interrupts, the Local SAPIC unit in the processor also functions as an I/O xAPIC unit in the sense that it can generate interrupts to the processor core. Local to the processor are two separate sources of interrupts: those generated by the processor core such as the timer, performance monitoring, or CMC; and those generated by an external device connected to one of the interrupt pins of the processor (i.e. LINT0 and LINT1, if these pins are supported by the processor).

Within the processor Local SAPIC unit, there is a set of Local Vector Registers (LVRs) and a set of Local Redirection Registers (LRRs), which are similar in format, function, and capability to the RT contained in the I/O xAPIC unit. The purpose of the LVRs and LRRs is to describe all the locally generated interrupts in the same fashion as the RT in the I/O xAPIC, with the exception that the destination ID fields are not used because all the locally generated interrupts are directed to the local processor only.

2.2.3 Interprocessor Interrupts

One processor can interrupt another processor (including itself) by generating an interprocessor interrupt (IPI). A processor generates interrupts by performing an uncacheable store to an aligned address within the Processor Interrupt Block (PIB). Interrupt information is exchanged between different Local SAPIC units on the system bus in the form of interrupt messages, the format of which is identical to interrupts generated by the I/O xAPIC. The address used for the store instruction determines which processor the interrupt is delivered to, and the data being stored is similar in format to the RTE used for the I/O xAPIC or the LVR/LRR registers (but only includes the Delivery Mode and the Vector. See Section 5.8.4 of the *Intel® Itanium® Architecture Software Developer's Manual, Volume 2*).

2.3 Software Model

This section describes the software model for both the Local and the I/O xAPIC units of the SAPIC architecture.

2.3.1 Local SAPIC

The Local SAPIC is the interrupt resource. It is responsible for pending local interrupt sources, receiving SAPIC interrupt messages, and dispensing interrupts to the processor core. At most one Local SAPIC unit will receive an interrupt if the Local SAPIC ID/EID matches that of the interrupt message. If a Local SAPIC receives an interrupt, it will deliver the interrupt to its processor.

2.3.1.1 Local SAPIC Registers

Software interacts with the Local SAPIC by reading and writing its registers. The register set of each Local SAPIC unit appears as control registers for each processor. All registers can be accessed by using a 64-bit move instruction. This implies that to modify a field (e.g., a bit or a byte) in any register, the entire (64-bit) register must be read, the field modified, and the entire register written back. These registers are used to prioritize and deliver interrupts. Table 2-1 is a subset of the interrupt control registers. Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for additional information.

When reading and writing to the Local SAPIC registers, multiple clocks may be required to finish the write. To avoid slow access to the control registers, software can implement reads at the beginning and writes at the end of data serialization to take full advantage of the capabilities of the processor architecture. For example:

```
   r1 = CR.IVR;
// Do other stuff
   srlz.d;
   CR.EOI = ...;
```

Table 2-1. Interrupt Control Registers (Subset)

Register		Register Description	
LID	Local ID Register	R/W	This register contains the processor's SAPIC ID and EID. The ID/EID serves as a physical address of the Local SAPIC unit and is loaded after reset by the PAL and SAL firmware. This ID is compared to the target address of an interrupt message for receiving interrupts.
IRR[3:0]	Interrupt Request Register	R	Four 64-bit read-only registers allow software to determine which of the 256 external interrupts are pending. Each vector bit is set when the processor receives the interrupt.
IVR	Interrupt Vector Register	R	This register allows the software interrupt handlers to determine the highest priority pending external interrupt. Each IVR register read will clear the reported interrupt vector pending indication bit in the IRR register.
EOI	End of Interrupt Register	R/W	Software writes to this register when it is ready to accept another interrupt at a higher or same priority level than specified by the TPR.
TPR	Task Priority Register	R/W	Interrupt messages received by the Local SAPIC cause an interruption in the normal instruction execution only if the interrupt corresponds to higher priority than the class indicated by the TPR. This register supports 240 maskable interrupt priority levels: interrupt vectors 16 to 255, divided into 15 interrupt classes.
CMCV ITV PMV	Local Vector Registers	R/W	These registers contain the vector number for internal asynchronous events, e.g. ITV, corrected machine-check, and performance monitoring events (PMV). These registers duplicate the vector number in the IVR for their respective interrupt events.
LRR[1:0]	Local Redirection Registers	R/W	The LRRs contain the vector number and delivery mode for the local I/O devices (i.e. LINT0/LINT1 external devices, if available).

2.3.1.2 Local SAPIC Redirection Registers

The Local SAPIC contains two sets of Redirection registers: the LVRs (ITV, CMC, PMV), which contain information on interrupts generated locally within the processor and the LRRs, which contain information on interrupts generated locally from the local external interrupt pins (i.e. LINT0 and LINT1, if available). Refer to Figure 2-2 and Figure 2-3). There is one entry in each redirection table for each interrupt source. Software can decide the vector (and hence the priority) dedicated to each individual interrupt source. This vector is then used for priority arbitration along with all other pending interrupts received by the Local SAPIC unit.

Figure 2-2. Local Vector Registers (CMC Vector, ITV, PMV)

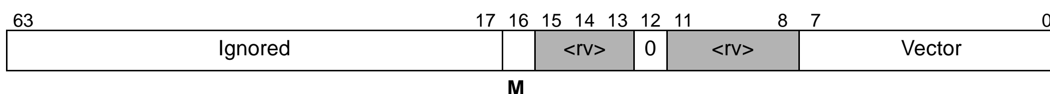
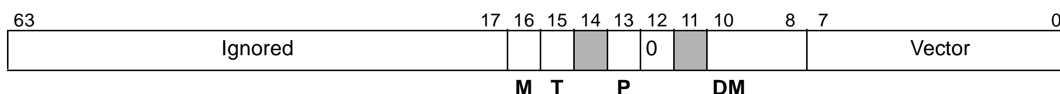


Figure 2-3. Local Redirection Register



- **Vector:** An 8-bit field containing the interrupt vector. Vector values can range between 0x10 and 0xFF.
- **Mask (M):** Masks the delivery of this interrupt.

0 (Unmasked)	Indicates that occurrences of this interrupt are pended.
1 (Masked)	Indicates that occurrences of this interrupt are not pended. The interrupts signaled on a masked LVR entry are simply ignored.

The only control available for locally generated interrupts is whether to mask them or not. The destination is always the local processor, and the triggering mode is always edge.

Note: Certain delivery modes will only operate as intended when used in conjunction with a specific trigger mode. These restrictions are indicated in the list for each delivery mode.

- **Vector:** An 8-bit field containing the interrupt vector.
- **Delivery Mode (DM):** The delivery mode is a 3-bit field that specifies how the local processor should act upon receipt of this signal.

000 (INT)	Deliver the signal to the local processor. Trigger mode can be edge or level. The vector value can range between 0x10 and 0xFF.
010 (PMI)	Deliver the signal on the Platform Management interrupt (PMI) signal of the local processor. The interpretation of the vector field is implementation defined: the vector ranges from 0x1-0xF (0x1-0x3 are useable by SAL; all other values are Intel reserved).
100 (NMI)	Deliver the signal on the NMI signal of the local processor; vector information is ignored, but the vector read by the Itanium architecture-based processor will be 0x2.
101 (INIT)	Deliver the signal on the Initialization interrupt (INIT) signal of the local processor; vector information is ignored.
111 (ExtINT)	Deliver the signal to the local processor as an interrupt originating from an Intel 8259A compatible external interrupt controller. A software-initiated INTA transaction that corresponds to this ExtINT delivery will be routed to the external controller that is expected to supply the vector. A delivery mode of ExtINT requires an edge trigger mode. SAPIC architecture supports only one ExtINT source in a system. Vector information is ignored, but the vector read by the Itanium architecture-based processor will be 0x0.

- **Interrupt Input Pin Polarity (P):** Specifies the polarity of each interrupt signal connected to the local interrupt pins of the processor.

0 (High)	Means the signal is active high.
1 (Low)	Means the signal is active low.

- **Trigger Mode (T):** Indicates the type of signal on the interrupt pin that triggers an interrupt.

0 (Edge)	Indicates edge sensitive. Assertion of the LINT pin pends an external interrupt for the specified vector.
1 (Level)	Indicates level sensitive. Assertion of the LINT pin pends an external interrupt for the specified vector; deassertion of the LINT pin removes the pending interrupt.

- **Mask (M):** Masks the delivery of this interrupt.

0 (Unmasked)	Indicates that occurrences of this interrupt are pended.
1 (Masked)	Indicates that occurrences of this interrupt are not pended. Edge-sensitive interrupts signaled on a masked Interrupt Input pin are simply ignored (i.e. it is not delivered and is not held pending). Level asserts or deasserts occurring on a masked level-sensitive pin are also ignored and have no side effects. Changing the Mask bit from unmasked to masked after the interrupt was received by a processor has no effect on that interrupt. This behavior is identical to that in which the device withdraws the interrupt before that interrupt is posted to the processor. It is the software's responsibility to deal with the case where the Mask bit is set after the interrupt message has been received by a processor but before the interrupt is dispatched to the processor.

2.3.2 I/O xAPIC

The I/O xAPIC unit consists of a set of interrupt input pins, an interrupt RT, and a message unit for sending SAPIC messages on the I/O bus. The I/O xAPIC unit is where I/O devices inject their interrupts. The I/O xAPIC unit selects the corresponding entry in the RT and uses the information in that entry to format an interrupt request message. The message unit then sends this message over the I/O bus. For the purpose of this guide, PCI is used as an example of the I/O bus. The content of the RT is under software control and is assigned default values upon reset.

2.3.2.1 I/O xAPIC Registers

Software interacts with the I/O xAPIC by reading and writing to its registers, which are memory-mapped into the physical address space. In a multiprocessor (MP) system, the register sets of I/O xAPIC units would be globally accessible by any processor that has access to the address range. If a system contains multiple I/O xAPIC units, then each of the I/O units would be located at a different address.

All registers are accessed using 32-bit uncacheable loads and stores to a reserved memory location in system memory. This implies that to modify a field (e.g., a bit or a byte) in any register, the whole 32-bit register must be read, the field modified, and the 32 bits written back. Partial register access, or non-aligned register access, are implementation-defined by the I/O xAPIC and will not be compatible across different implementations. Also, registers that are described as 64 bits wide are accessed as multiple independent 32-bit registers.

There are a set of external registers (directly accessible via memory-mapped addresses) and a set of internal register (indirectly accessible via two external registers) as described in Table 2-2.

Table 2-2. External and Internal Register Description

Register	Access	Offset	Register Description
I/O Register Select Register	Direct	Base+0x00	This 32 -bit register is used to determine which read/writable internal register within the I/O xAPIC is manipulated when the I/O Window register is accessed. There are a maximum of 256 internal registers residing in the I/O xAPIC.
I/O Window Register	Direct	Base+0x10	This 32-bit register is mapped by the value in the I/O Register Select register to the designated internal register.

Table 2-2. External and Internal Register Description (Continued)

Register	Access	Offset	Register Description
IRQ Assertion Register (optional)	Direct	Base+0x20	This register allows several devices to generate interrupts without increasing the number of interrupt pins to the I/O xAPIC. I/O devices supporting this register can generate an interrupt by writing an interrupt vector to it.
I/O EOI Register (optional)	Direct	Base+0x40	This 32-bit register is used by software to communicate to the I/O xAPIC that it has serviced a level-triggered interrupt initiated by the I/O xAPIC along with the vector number. If the interrupt line is still held high, the I/O xAPIC should generate a new interrupt message for the vector.
I/O xAPIC Version Register	Indirect	0x001	This 32-bit read-only register identifies different I/O xAPIC implementations and their versions. It also contains the maximum number of RT entries supported by the I/O xAPIC.
Redirection Table Entry X {31:0}	Indirect	0x010 through 0x0FE by 2	This I/O RT has entries dedicated to each interrupt source. The lower 32-bits of each entry specifies the interrupt vector, delivery mode, status, trigger mode, and mask mode.
Redirection Table Entry X {63:32}	Indirect	0x011 through 0x0FF by 2	This I/O RT has entries dedicated to each interrupt source. The upper 32 bits of each entry specifies the destination processor ID.

When accessing the I/O xAPIC RT, care must be taken to ensure that stores do not occur out of order. The first step is to do a store to the I/O Register Select register and second, do a load/store to the I/O Window register. If the two accesses occur out of order, the result is that the data being read/written belongs to the entry that was previously selected by the I/O Register Select register. To ensure the order of the load and store operations, the I/O xAPIC registers must be accessed with uncacheable attributes.

2.3.2.2 I/O xAPIC Redirection Table

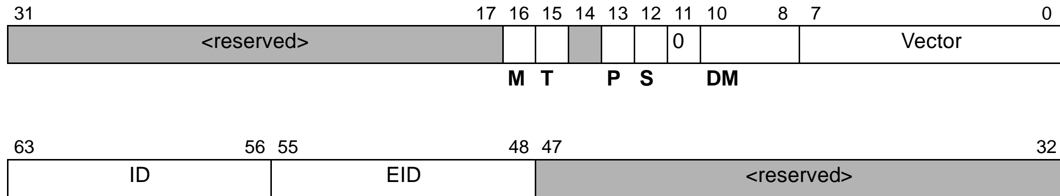
The I/O RT has entries dedicated to each interrupt source of the I/O xAPIC. Unlike the Intel 8259 interrupt controller, there is no concept of interrupt priority depending on the pin for which the interrupt was asserted. Software can decide the vector (and hence the priority) dedicated to an individual pin. The information in the I/O RT is used to translate an incoming interrupt signal into the appropriate interrupt message.

Once an interrupt is detected on an input pin, a delivery status bit within the I/O xAPIC is set. The interrupt remains pending until the interrupt message associated with that interrupt has been successfully delivered on the bus.

Only after the delivery of the interrupt message has occurred will a new occurrence of an edge-triggered interrupt on that interrupt input pin be recognized. If a new interrupt message is issued for a new occurrence at the interrupt input pin, the recognition of that interrupt will occur if, and only if, the previous interrupt pending has already been serviced by the processor (i.e. if the processor's Interrupt Request bit for that interrupt vector is already set to a '1', the new interrupt will not be recognized).

The number of entries in the RT is implementation dependent, and that number is recorded in the I/O xAPIC Version register. Because the RT registers are accessed indirectly via the 8-bit offset stored in the I/O Register Select register, there can be a maximum of 120 RTEs (each entry consists of two 32-bit registers) per I/O xAPIC. See Figure 2-4 for the I/O xAPIC RTE definition.

Figure 2-4. I/O xAPIC Redirection Table Entry Definition



Note: Certain delivery modes will only operate as intended when used in conjunction with a specific trigger mode. These restrictions are indicated in the list for each delivery mode.

- **Vector:** An 8-bit field containing the interrupt vector.
- **Delivery Mode (DM):** The delivery mode is a 3-bit field that specifies how the processors listed in the destination field should act upon receipt of this signal.

000 (INT)	Deliver the signal to the processor listed in the destination. Trigger mode can be edge or level. The vector value can range between 0x10 and 0xFF.
001 (INT with Redirectable hint)	The same as "INT" delivery, except that the platform is allowed to redirect the interrupt to another processor on the same processor system bus as the destination (including the destination processor listed in the destination). See Section 2.4.2.
010 (PMI)	Deliver the interrupt on the PMI signal of the processor listed in the destination. The interpretation of the vector field is implementation defined: the vector ranges from 0x1-0xF (0x1 - 0x3 are useable by SAL; all other values are Intel reserved). The trigger mode must be edge. This delivery mode is compatible with the System Management interrupt (SMI) delivery mode of the APIC when the vector is 0x0.
100 (NMI)	Deliver the signal on the NMI signal of the processor listed in the destination; vector information is ignored, but the vector read by the Itanium architecture-based processor will be 0x2. NMI is always delivered as an edge triggered interrupt; the trigger mode field is ignored.
101 (INIT)	Deliver the signal on the INIT signal of the processor listed in the destination; vector information is ignored.
111 (ExtINT)	Deliver the signal to the INTR signal of the processor listed in the destination as an interrupt that originated in an 8259 compatible external interrupt controller. A software-initiated INTA transaction that corresponds to this ExtINT delivery will be routed to the external controller that is expected to supply the vector. A delivery mode of ExtINT requires an edge trigger mode. SAPIC architecture supports only one ExtINT source in a system. Vector information is ignored, but the vector read by the Itanium architecture-based processor will be 0x0.

- **Delivery Status (S):** A 1-bit field that contains the current status of the delivery of this interrupt. This is a software read-only field; writes to this field do not affect this bit. This field is only used for debugging purposes.

0 (Idle)	Indicates that there is currently no activity for this interrupt.
1 (Pending)	Indicates that the interrupt has been injected, but it has yet to be delivered.

- **Interrupt Input Pin Polarity (P):** Specifies the polarity of each interrupt signal connected to the interrupt pins of the I/O xAPIC.

  0 (High) Means the signal is active high.

  1 (Low) Means the signal is active low.

- **Trigger Mode (T):** Indicates the type of signal on the interrupt pin that triggers an interrupt.

  0 (Edge) Indicates edge sensitive.

  1 (Level) Indicates level sensitive.

- **Mask (M):** Masks the delivery of this interrupt.

  0 (Unmasked) Indicates that delivery of this interrupt is not masked. An edge or level on an interrupt pin that is not masked results in the delivery of this interrupt message to the destination processor.

  1 (Masked) Indicates that delivery of this interrupt is masked. Edge-sensitive interrupts signaled on a masked Interrupt Input pin are simply ignored (i.e. it is not delivered and is not held pending). Level asserts or deasserts occurring on a masked level-sensitive pin are also ignored and have no side effects. Changing the mask bit from unmasked to masked after the interrupt was accepted by a processor has no effect on that interrupt. This behavior is identical when a dev mask bit is set after the interrupt message has been received by a processor but before the interrupt is dispatched to the processor.

- **Destination ID and EID:** Specifies a processor ID and EID. Processors receiving an interrupt message compare these fields to the corresponding fields in their LID registers to determine if the interrupt is the interrupt to be received by them.

Because the RTEs are accessed indirectly in 32-bit chunks, it is necessary to make two write accesses to fully program one RTE. The proper sequence to do this is to:

1. Program the Mask bit to mask any interrupts (after a reset, the Mask bit is set automatically, so it is not necessary to set it if the RTE is being programmed for the first time after reset).
2. The destination field half of the RTE is written.
3. The Mask bit is cleared to enable the RTE to generate interrupt messages.

2.3.2.3 I/O xAPIC Version Register

Each I/O xAPIC unit contains a Version Register (VR) that identifies different implementations of I/O SAPIC and their versions. Software can use this to provide compatibility between different I/O SAPIC implementations and their versions. The VR also contains the Max RTE. These fields are described in Figure 2-5.

Figure 2-5. I/O SAPIC Version Register

31 24	23 16	15 8	7 0
<APIC Reserved>	Max. Redir. Entry	<APIC Reserved>	Version

- **Version:** This is a version number that identifies the implementation version. This field is hardwired and is read-only. the version numbers are assigned for 82489DX, APIC, and SAPIC as follows. “X” means a don’t care 4-bit value.

  0x0X = 82489DX

  0x1X = APIC

0x2X = SAPIC

0x30-0xFF = Reserved

- **Max Redir Entry:** This is the entry number (0 being the lowest entry) of the highest entry in the I/O RT. This field is hardwired and is read-only.

2.3.3 Interprocessor Interrupts

IPIs are generated by software using a 64-bit uncacheable store of an interrupt command into an 8-byte aligned address within the PIB. See Section 2.7.1 for a detailed discussion of the IPI generation process.

2.4 Interrupt and Processor Priorities

Each interrupt is associated with a vector number which determines the priority of the interrupt. Typically, the higher the vector number, the higher the priority of the interrupt. When interrupts are enabled in the processor (i.e. PSR.i is set to 1), a higher priority interrupt can interrupt the processor even when it is already servicing a lower priority interrupt, but a lower priority interrupt cannot interrupt the processor which is servicing a higher priority interrupt.

Within the processor Local SAPIC unit, there is a Task Priority Register (TPR), which is used by software to mask interrupts based on "priority class." There are 256 individual interrupt vectors in the SAPIC architecture; these are divided into 16 classes of 16 vectors each. The lowest 16 vectors are reserved (class 0). Classes 1 - 15 are maskable by the TPR.mic, which contains four bits, corresponding to 16 classes.

When TPR.mic contains 0x5, then classes 1-5 are masked (vectors 0x10-0x5F). When the TPR contains 0xE, then all interrupts below 240 are masked, and so on. Even when the TPR contains zero, the lowest priority class (interrupt vectors 0x00 to 0x0F) is still masked; therefore, these vectors are not usable for normal interrupts.

TPR.mmi is a single bit that can mask maskable interrupts. If TPR.mmi is 1, then all interrupts other than NMI (vector 0x2) are disabled, including class 0. If TPIR is 0, then TPR.mic determines interrupt masking.

Refer to the *Intel® Itanium® Architecture Software Developer's Manual*, Section 5.8.1 for additional information

Any interrupt that is received by the processor is kept pending and recorded in the Interrupt Request Register (IRR). If the processor is not servicing an interrupt, then the contents of the TPR determines whether the processor will accept a pending interrupt depending on the priority of the interrupt compared to the current TPR. If the interrupt has a higher priority, then the processor is interrupted, otherwise the interrupt is kept pending. However, when the processor is already servicing an interrupt (which must have been accepted when the TPR was at a value lower than the priority of the interrupt being serviced), then the priority level of the interrupt being serviced is also used to determine whether to accept another pending interrupt. If the new interrupt has higher priority, then it is serviced immediately. See Section 2.4.3 for more information.

This temporary processor priority is necessary to ensure that only higher priority interrupts are allowed to disrupt the processing of another interrupt. Although the TPR only has 16 levels of priority and interrupts are masked by class, it really has 256 individual levels of priority when the

processor is servicing an interrupt. As an example, vector 0x51 is allowed to disrupt the processor while the processor is servicing interrupt vector 0x41. See Chapter A for differences between the SAPIC and APIC architectures.

2.4.1 Batch Processing of Interrupts

The SAPIC architecture supports batch processing of interrupts. When an interrupt occurs in an APIC system, the processor and OS first perform a context switch from the application to the OS in order for the device driver to handle an interrupt. Upon completion of this ISR, a context switch must occur again to return to the application. If a second (lower priority) interrupt occurs prior to the completion of the initial ISR, then the second interrupt cannot be processed until the initial ISR completes and returns control to the application. At this point, the application must context switch again to the OS. Each context switch is time consuming because the processor states must be saved. With the SAPIC architecture, when the second interrupt occurs prior to the completion of the initial ISR, this second interrupt can be processed along with any other pending interrupts by reading the Interrupt Vector Register (IVR) before context switching back to the application.

The SAPIC architecture allows the ISR of the OS to process all pending interrupts within the current priority level. This batch processing of interrupts eliminates the need for repeated interruptions of the processor and the associated time consuming context switching that is necessary to service the interrupts.

As discussed in Section 2.4.4, there is a range between the priority of the interrupt currently being serviced and the contents of the TPR. Any pending interrupts within this range can be serviced as a batch without exiting the ISR. Obviously a higher priority interrupt that arrives during this time will disrupt the ISR and cause the interrupts to nest if interrupts are enabled.

The conceptual ISR in the OS would look similar to the following pseudo code fragment:

```
Save processor state;
PSR.ic = 1;
srlz.d;
// Interrupt is disabled upon entry
Loop {
    Read IVR register into variable VECTOR;
    if (VECTOR == 0x0F) {  // Spurious interrupt, no more interrupts
        Enable Interrupts;
        Return from Interrupts;
    }
    Enable Interrupts;  // So higher priority interrupts can come in
    Service (VECTOR);  // 8259 handler will send EOI directly to 8259
    Disable Interrupts;
    if (external interrupt) {  // includes 8259 interrupts
        if (Level_Triggered (VECTOR))
            Write to I/O xAPIC I/O EOI register;
        Write to EOI register in Local SAPIC;
    }
}
```

Note: Any read of the IVR is destructive if there is an interrupt pending that is within the priority range. Once the read of the IVR is done, the pending interrupt is removed from the IRR and must be processed or else the interrupt is lost. The interrupt can be pended again by sending an IPI with the same vector number to any processor, including itself.

2.4.2 Redirectable Interrupt Delivery

The lowest priority interrupt delivery methodology increases the probability of delivering an interrupt request to a processor that is executing a lower priority task. To implement the lowest priority delivery method over the system bus requires costly bandwidth. This is in addition to the unnecessary complexity that would need to be added on the bus interface unit.

Instead of the lowest priority delivery of the APIC, the Itanium architecture implements an interrupt redirection mechanism that is controlled by system firmware. This mechanism is an optional part of the platform architecture. The bridge controller oversees the interrupt redirection as opposed to the processor Local SAPIC unit. A set of External Task Priority (XTP) registers, one for each Local SAPIC unit, can optionally be integrated in the chipset. These memory-mapped registers keep track of each processor's current priority level. They do not need to be a true reflection of the actual TPR registers of the processors. Update of the XTPRs can be performed by the OS or the System Abstraction Layer (SAL) firmware before OS boot.

The SAL Platform Features Descriptor Table can be used by the OS to determine if the platform supports either I/O interrupt redirection or IPI redirection. The firmware can implement different heuristics to control delivery of the interrupts to the processors. As an interrupt is delivered to the system bus, the interrupt destination may be altered by modifying the destination ID of the interrupt transaction based upon the XTP registers. The interrupt transaction may be diverted from its original target processor to another processor with a lower priority on the same bus.

Because there is a one-to-one correspondence between the XTP registers and the processor ID (as set by the hardware at system reset time), software should not alter the processor ID field that identifies the processor on the system bus. The proper sequence to program the LID register is to read the register, modify only the portions that identify the processor bus for interrupt routing purposes (which may include parts of the ID and EID fields), then write the new information back into the LID register.

Of course, the software can always reprogram the I/O xAPIC unit RT to change the destination ID/EID fields to send an interrupt to another processor, but that operation is both time consuming and inflexible. The redirectable interrupt delivery mechanism in the chipset is relatively inexpensive to change (in terms of time) because the chipset is much closer to the processor. Also, the destination processor can be changed dynamically at execution time depending on the workload being executed and the frequency of the interrupts, allowing for a more flexible solution in response to the changing system environment.

However, platform level interrupt redirection is optional, therefore software must be aware of whether the platform supports this capability or not. Because the agent that does the redirection is the chipset, it can only redirect interrupts that are delivered through the chipset; processor local interrupts cannot be redirected. Since IPIs can also be placed on the system bus with the redirectable hint, other processors cannot receive the interrupt message until the chipset has a chance to process the redirection arbitration. Therefore, the minimum requirement on any chipset is to turn off the Redirectable Hint bit in any interrupt message, even if it does not actually implement any priority arbitration to change the destination ID. If the chipset does not implement redirection for IPIs, any IPI placed on the system bus with the Redirectable Hint bit set will not be received by any agents and the interrupt will be lost.

If the platform chipset does not implement the redirection, then any XTP register updates will not be useful and will only consume bus bandwidth. Therefore arbitrarily updating the XTP registers when the platform does not implement redirection may hurt total system performance. It is highly recommended that software be made aware of whether this capability is supported by the platform.

2.4.2.1 Software Heuristics for Interrupt Redirection

Since the updating of the XTP registers is completely under the control of software, different heuristics can be used to accomplish various modes of operation. The following are several alternatives for programming the XTP registers to achieve different effects. Software can also determine the best trade-offs between the accuracy of delivering the interrupt to the lowest priority processor versus the amount of bus traffic needed to update the XTP registers. However, if two processors have the same XTP priorities, then which processor receives the interrupt is completely dependent on the chipset implementation; software must not rely on intelligent or deterministic behavior from the chipsets.

2.4.2.1.1 In Sync with TPR

The simplest and most straightforward method is to update the XTP registers whenever the TPR registers are updated. However, if the TPR registers are being changed frequently, this may take up some of the processor system bus bandwidth.

2.4.2.1.2 Fewer Buckets

One optimization is to divide the processor priority into a smaller number of buckets ranging from 2 to 16 (the TPR supports 15 priority levels). For example, if the priority is divided into three buckets: *high*, *medium,* and *low*. Software will have complete flexibility in deciding when a processor is in a high level, a medium level, etc. And since this is completely controlled by software, both the number of buckets and when to transition from one bucket to another are dynamically adjustable at run time. A system can start out with two buckets and then add a third bucket as the system load changes.

2.4.2.1.3 All for One

Another method of distributing interrupts is to send them all to one processor until that processor becomes saturated. This can be achieved by setting the XTP value of the chosen processor very low, regardless of the actual value of the TPR register. Then all interrupts that can be redirected will be sent to that chosen processor. When the chosen processor is saturated, the XTP register of another processor will be reprogrammed to have a lower value and the XTP of the former chosen processor will be raised. This avoids the need to reprogram the individual RTE in the I/O xAPIC.

2.4.2.1.4 Round Robin

It is also possible to program the XTP registers to simulate the distribution of the redirectable interrupts in a round robin manner. The XTP registers are initialized so that only one processor is at a low priority while all the others are at the same high level. Then when a redirectable interrupt is delivered to the low priority processor, it will notify the next processor to lower its XTP register value and raise its own XTP value.

2.4.3 Interrupt Nesting

When the processor is servicing an interrupt of a certain priority, the priority of the processor is increased momentarily from whatever is set in the TPR to the priority of the interrupt being serviced; this is done to prevent a new interrupt, with priority between the TPR and the current interrupt, from interrupting the current ISR. However, if an interrupt of higher priority than the current interrupt arrives, then it must be serviced immediately. This condition results in the nesting of interrupt servicing.

The SAPIC architecture fully supports nested interrupts by keeping track of the priority of all the interrupts currently being serviced. When a nested interrupt occurs, the new priority is automatically noted, and when the new ISR is finished, the priority automatically reverts back to the priority of the previous interrupt being serviced.

For example, suppose the processor is servicing an interrupt with vector 0x50, and a new interrupt with vector 0x80 arrives; since vector 0x80 is of higher priority than vector 0x50, the ISR that is servicing vector 0x50 will be interrupted and the new interrupt 0x80 will be serviced first, if interrupts are enabled. When that is finished, servicing for vector 0x50 will resume until it is finished.

2.4.4 Interrupt Masking

It is possible to prevent interrupts from interrupting the task the processor is performing. The process is called interrupt masking, and there are two ways of accomplishing this, with varying implications and interrupt masking coverage. One way is to prevent collected interrupts from being accepted and keep them pending within the processor; the other way is to prevent the interrupts from reaching the processor itself.

The most complete “masking” of all interrupts is via the Processor Status Register (PSR) Interruption Enable bit, or the PSR.i. Using this bit, all interrupts are disabled (including NMI, but excluding PMI, INIT, and MCA) and kept pending until this bit is set. The processor will automatically clear this bit when an interrupt is taken so that another interrupt of higher priority will not interrupt the ISR until it is ready to be interrupted. An ISR should do all the necessary state saving before it enables interrupts again, and it should do so only when it is ready to be interrupted.

Modifications of the PSR and Control registers may require serialization to ensure that new values written to a register are observed by subsequent instruction reads of the same register.

The PSR.ic (interrupt collection) bit supports a nested interruption model and additional masking capability. When an interruption event occurs, the various interrupt resources (i.e. Interruption Processor Status Register [IPSR]) are overwritten with information pertaining to the PSR in effect at the time of the interruption. To prevent the current set of resources from being overwritten on a nested fault, the PSR.ic bit is cleared upon any interrupt. This will suppress writing of critical interrupt resources if another interrupt occurs while it is disabled. For example:

```
// Save processor state;
PSR.ic = 1;
srlz.d;
PSR.i = 1;
```

A more common method of masking interrupts is via the Mask Maskable Interrupts (MMI) field of the TPR. This will mask all interrupts other than the NMI. By toggling this single field of the TPR, it is not necessary to save and restore the current processor priority that is kept in the TPR if the processor priority is not changed. For example:

```
CR.TPR = ...;
srlz.d;
//effect of new priority is seen
```

When the MMI field of the TPR is not set, the processor priority is determined by the Mask Interrupt Class (MIC) field of the TPR. The TPR supports 15 classes of interrupt levels. When the TPR is set at a particular level, all interrupts of a priority equal to and lower than that level (i.e. having a vector number lower than the TPR class value times 16) are masked, and will be kept

pending until the processor's priority is lowered to below the priority of the interrupt. For example, when the MIC field is 5, interrupt classes 1-5 are masked, which corresponds to vectors 0x10 to 0x5F.

Finally, interrupts can be masked at the source. In the I/O xAPIC unit (or the equivalent LVR/LRR registers within the processor), each RTE has a Mask bit. When the Mask bit is set, any activities of the interrupt line corresponding to the RTE are prevented from generating an interrupt message. However, accessing the RTE of the I/O xAPIC can be a slow process, therefore it should not be used as a normal method of masking interrupts. Depending on the arrival, interrupts that occur while they are masked at the RTE may or may not be latched when the RTE is unmasked.

Besides the explicit masking of interrupts via the above methods, the Local SAPIC unit in the processor also does implicit priority masking. When an interrupt of a certain priority (i.e. vector number) is being serviced, the priority of the processor is implicitly asserted to the priority level of the interrupt, thereby masking all interrupts of a an equal or lower priority.

2.5 Interrupt Handling

Interrupts can be viewed in two ways: as a discrete event or as a condition. The implementation of these two views is through the edge-triggered interrupt and the level-triggered interrupt, respectively. The processor Local SAPIC unit, however, can only handle interrupts as discrete events; therefore, the I/O xAPIC units must convert conditions to a series of discrete events that need servicing if the condition still exists.

Additionally, there are a few special interrupts that require special handling, such as the NMI or the PMI.

Refer to the *Intel® Itanium® Architecture Software Developer's Manual, Volume 2* for further details on edge and level-triggered interrupt handling on Itanium Architecture-based processors.

2.5.1 Edge-Triggered Interrupts

A single edge-triggered interrupt counts as a single occurrence of an event. A new occurrence of an interrupt that is already pending cannot be distinguished from the previous occurrence. All occurrences to a given vector number are recorded in the same Internal Interrupt bit in the IRR. They are therefore treated as the "same" interrupt occurrence. If the second occurrence happens after the processing of the first occurrence has started (i.e. a read of the IVR has removed the pending interrupt bit from the IRR), then the second occurrence is treated as a new interrupt and will be recorded as pending in the IRR.

With edge-triggered interrupts, it is not necessary for the software to inform the I/O xAPIC that the interrupt has been serviced. The only action required of software is to write to the End of Interrupt (EOI) register of the Local SAPIC unit to inform it that servicing of the interrupt is completed and the processor is available to handle the next lower priority interrupt that is pending.

2.5.2 Level-Triggered Interrupts

Level-triggered interrupts are handled differently from edge-triggered interrupts because (1) level-triggered interrupts are conditions that exist as opposed to events for an edge-triggered interrupt, and (2) level-triggered interrupts can be shared by multiple I/O devices.

When a level-triggered interrupt occurs, the I/O xAPIC sends an interrupt assert message to the destination processor on the rising edge. The I/O xAPIC notes that a message has already been sent to the RT. When the ISR of the OS (or the device driver) has finished servicing the interrupt, it must write to the I/O EOI register of the I/O xAPIC that sent the original interrupt. When the I/O xAPIC receives the I/O EOI data, which should be the vector of the level-triggered interrupt that was just serviced, it will compare I/O EOI data against the entries in the RT. If the EOI vector matches one of the entries, the I/O xAPIC will resample the input interrupt line of that entry to see if it is still being held high (i.e. the condition still exists). If the interrupt line is still high, then the I/O xAPIC will send another interrupt assert message to the destination processor.

Note: Because this reassertion message is a new instance of an interrupt, it may be redirected to another processor if the platform-level redirectable interrupt delivery mechanism is being used.

It is possible that after the I/O xAPIC has sent the interrupt assert message to the processor that the I/O device connected to the interrupt line has deasserted the line to indicate that the condition no longer exists. In this case, the original interrupt message will be received by the processor. When the ISR queries the I/O device, it will receive an indication that the condition no longer exists and there is no interrupt to service. This situation is also called a spurious interrupt, although no special message is used in this case to indicate the change in condition. ISRs should be written to handle such situations properly. See Section 2.5.2.2 for more discussion on spurious interrupts.

An advantage of implementing conditions as level-triggered interrupts is that multiple I/O devices can share the same interrupt line even though individually the interrupts from each device are discrete events. Since the interrupt line is shared, multiple I/O devices can all assert the line and it will register at the I/O xAPIC unit as a single condition. The process of handling these shared interrupts is that the ISR must be aware of the sharing, and when a level-triggered interrupt occurs, the ISR must poll all the devices that are sharing the interrupt line (i.e. the same vector, although they could be using different I/O xAPIC units or different entries in the RT of the same I/O xAPIC unit.).

After a level-triggered interrupt is serviced, the software must also write to the EOI register of the Local SAPIC unit to inform it that the servicing of the interrupt is completed and the processor is available to handle the next lower priority interrupt that is pending.

2.5.2.1 EOI Handling

EOI is a mechanism for the software to inform the interrupt controller hardware that the servicing of an interrupt is done and that the processor is ready to handle the next interrupt. In the SAPIC architecture, there are two different levels of EOI, one is at the Local SAPIC unit and the other is at the I/O xAPIC unit.

The Local EOI register in the Local SAPIC unit is used to tell the local interrupt controller that the servicing of the current interrupt is done, thus restoring the priority of the processor. When the interrupt servicing is done, the write to the Local EOI register will revert the priority of the processor back to its previous level, which could be the priority of a lower priority interrupt being serviced (if there is a nested interrupt) or the setting of the TPR if there was no nested interrupt.

At the I/O xAPIC unit, there is an I/O EOI register. SAPIC does not support broadcast of the write to the Local EOI register to the I/O xAPIC units, therefore software must explicitly write to the I/O EOI register of the I/O xAPIC units. The write to the I/O EOI register is necessary only for level-triggered external interrupts. The function of the I/O EOI write is to inform the I/O xAPIC unit that servicing of the specific interrupt vector is done, and if the line is still held high by the I/O device, then another interrupt should be sent to the processor (this is the process of converting a condition into a series of events or edges). For edge-triggered interrupts, it is not necessary to write to the I/O EOI register because there is no resampling involved.

In order to do the directed I/O EOI write, the ISR of the OS (or the device driver) must be aware of which I/O xAPIC unit is programmed to generate a particular level-triggered interrupt vector. Since the OS programmed the appropriate I/O xAPIC unit, it knows exactly which I/O xAPIC unit is to generate the appropriate level-triggered interrupt vectors. The simplest way to do this is to keep a table of vectors and their respective I/O xAPIC units when the original programming of the RT of the I/O xAPIC unit is done.

I/O xAPIC units must be able to handle back-to-back I/O EOI writes if the I/O xAPIC is used in an MP platform. The back-to-back I/O EOI writes could be coming from two (or more) processors that are handling level-triggered interrupts originating from the same I/O xAPIC unit.

2.5.2.2 Spurious Interrupts

A spurious interrupt is an interrupt that is sent to the processor, but the condition that caused the interrupt is removed before the processor services the interrupt. Because servicing interrupts are time consuming activities, spurious interrupts should be avoided. The following discussion explains some of the circumstances where spurious interrupts can occur and how they can be avoided.

One source for spurious interrupts is from level-triggered devices. The process for servicing a level-triggered interrupt is as follows:

1. Processor is interrupted and enters ISR for the specific vector.
2. ISR polls all I/O devices that could generate that vector until it finds one that needs service. If no device needs service, then it is a spurious interrupt.
3. ISR tells the I/O device that its request is being serviced (this is typically a read or write to some device register). It continues to service the interrupt.
4. If the I/O device has no other reason to keep the interrupt line high, it will deassert the line.
5. ISR finishes servicing the interrupt and writes to the I/O EOI register of the I/O xAPIC.
6. Upon receipt of the I/O EOI write, the I/O xAPIC resamples the interrupt line connected to the RT that had the specific vector. If the line is still high, then another interrupt is sent to the processor; otherwise nothing happens.
7. ISR writes to the Local SAPIC EOI register to indicate that servicing of the interrupt is done.

Since there is a window of time between when the I/O device is indicated to deassert its line and when the interrupt line is resampled by the I/O xAPIC, there is a possibility for the I/O xAPIC to send an interrupt to the processor just before the line is deasserted by the I/O device. This is the spurious interrupt window. In order to minimize this window, the ISR tells the I/O device to deassert its line as soon as possible before it starts to service the interrupt. By the time the I/O EOI is written to the I/O xAPIC and the I/O xAPIC resamples the line, the I/O device will have deasserted the line, so that there is no chance for a spurious interrupt.

If the time to service the interrupt is shorter than the time required by the I/O device to deassert its interrupt line, then by the time the line is resampled by the I/O xAPIC it would still be held high. Keep in mind that the mechanism to tell an I/O device to deassert its line (typically either a read or write to an I/O register on the device) could be blocked or delayed en route to the I/O device, so it may take longer for the device to receive such notice. To ensure that the I/O device has received the command, it may be necessary for the ISR (or device driver) to query the device before it writes to the I/O EOI register.

A spurious interrupt can occur if an I/O device sends an interrupt and decides it does not need service from the processor after all. There is nothing software can do to eliminate spurious interrupts. An example would be if a condition existed causing an interrupt to be sent, and the condition being removed before the processor is interrupted. There is nothing for the processor to do in response to this kind of interrupt.

When a level-triggered spurious interrupt is received by the processor, the ISR must still send a directed I/O EOI back to the I/O xAPIC unit because when it sent the original interrupt to the processor, it recorded that an interrupt had been sent but not yet serviced. Therefore the I/O xAPIC unit will not resample the interrupt line again until the I/O EOI is received. If the ISR fails to send the directed I/O EOI write to the I/O xAPIC unit, then the interrupt line will forever be unsampled and the device is effectively hung, with no further interrupts serviced.

2.5.3 Greater than 240 Devices

There are 256 individual interrupt vectors, but since the lower 16 are reserved, only 240 interrupt vectors can be used by interrupts sources (or devices). However, There are several ways to have more then 240 sources in the system.

One method is to share level-triggered interrupts. With a single interrupt vector, all level-triggered interrupts may be routed through a single RTE. The ISR/device driver must be aware that there is more than one I/O device sharing the level-triggered interrupt line and thus polls all the devices when an interrupt is received. With this method, it does not matter how many processors are in the system, since they can all share the memory and OS services.

When there is more than one node in the system, another method can be used. In this case the processors, I/O devices, and I/O xAPIC units are partitioned into smaller domains. Then, within the domains, the I/O xAPIC units and the processors can reuse the same vector numbers. But any interrupt that can be routed to a processor outside of the domain must have a unique vector number. With this method, the processors in different domains cannot share the same ISRs or device drivers because the ISR must deal directly with the I/O xAPIC units or I/O devices.

A third method is to share the same vector number with multiple RTEs and even across multiple I/O xAPIC units. The two different triggering methods will also make a significant difference in the way vectors are reused.The ISR must be fully aware of the sharing and must poll all the possible devices that can generate the interrupt vector to determine if it needs servicing. Beyond that, handling of edge-triggered and level-triggered interrupts that reuse the same vector numbers are quite different.

Each interrupt is a separate instance of an event with edge-triggered interrupts. If two devices exist that are both using the same interrupt vector (but different RTEs), then if they both send an interrupt message to the processor simultaneously, only one interrupt is registered at the Local SAPIC unit's IRR. Therefore, it is up to the ISR or device driver to poll all the devices to ensure those that need service are handled. If this is not done, then the interrupt from the other device will be lost and will never be serviced.

With level-triggered interrupts, the situation is slightly different. All level-triggered ISRs (or device drivers) are already required to poll all possible devices that can generate the same interrupt vector, regardless of whether they share the same interrupt line or not. But because level-triggered interrupts require an explicit directed EOI write to the I/O xAPIC unit, the software must ensure that all I/O xAPIC units capable of generating the reused interrupt vector are informed that the servicing of the interrupt has finished and the line should be resampled.

2.5.4 NMI Handling

NMI is used for platform-level machine check (typically parity error) on IA-32 platforms. However, Itanium architecture-based platforms signal CMCs using the Corrected Platform Error Interrupt (CPEI). Machine check aborts are signaled using BERR#, BINIT#, or the hard-fail bus response. Note that NMI can be masked in the Itanium architecture (via the PSR.i field, but not via the TPR). NMI can be sent either through an I/O xAPIC unit using an IPI, or through the NMI pin of the processor. An OS should always be ready to handle interrupt vector 0x02, which is the NMI vector in Itanium architecture-based systems.

2.5.5 ExtINT Handling

ExtINT is used to support legacy interrupt controllers (i.e. 8259 support) in an Itanium architecture-based platform. ExtINT should always be programmed as edge-triggered. ExtINT is maskable by the MMI field of the TPR or the PSR.i. The external interrupts do not have any priority relationship with the other interrupts, therefore IRR bits are not maintained for them. If more than one ExtINT is directed to a Local SAPIC unit, it treats them as only one ExtINT; therefore no more than one interrupt should be programmed as ExtINT for each destination in a system. However, there can be more than one ExtINT type of interrupt in a system if each is directed to a different destination processor. In that case, the routing of the INTA cycle must be governed by the chipset so that the proper 8259 interrupt controller will receive the INTA and deliver up the interrupt vector number.

2.5.6 Timer Interrupt Handling

If the timer is programmed for intervals that are too short, then all the processor cycles are consumed in handling the timer interrupts and no forward progress can be made. The time to modify a register must be less than the programmed timer interval.

2.5.7 PMI Handling

PMIs are different from other interrupts because they do not interrupt the executing software and pass control to a software ISR. When a PMI is received by the processor, control is first passed on to the PAL firmware, then control is passed on to the SAL.

There are several ways of generating a PMI: from the processors using IPI, from an I/O device, or by asserting the PMI pin. When the PMI is delivered via an interrupt message, a parameter can be passed in a vector field to ensure those needing service are handled. But when the PMI is delivered via the pin, there is no parameter (parameter of zero is assumed). Vector values 0x0 to 0xF are used by system firmware and they have specific meanings to PAL/SAL. All other values are reserved.

2.6 8259 Interrupt Controller Support (Optional)

Note: The 8259 interrupt controller interrupt signal can be connected to the processor's external interrupt input pin. All interrupt lines that are connected to the 8259 interrupt controller must be connected to an I/O xAPIC. Itanium architecture-based operating systems must handle all interrupts via the SAPIC and should never program the 8259 interrupt controller for interrupt delivery.

Itanium architecture-based processors do not generate the INTA transaction automatically upon receipt of the interrupt from the 8259 interrupt controller. Software will read the IVR to find the vector number assigned to the ExtINT interrupt, then it is up to the software to generate the INTA

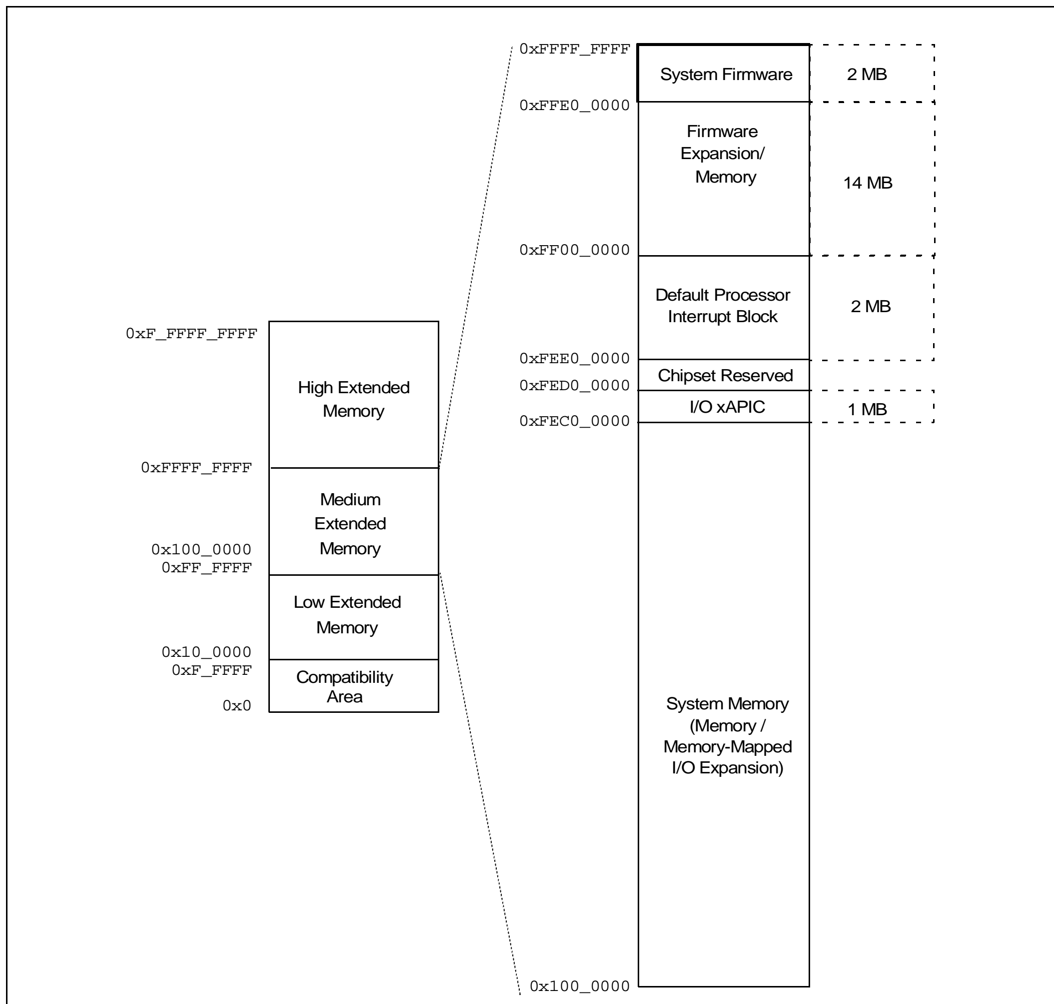
special transaction on the bus to ask the 8259 interrupt controller to return the interrupt vector number. Within the Itanium architecture-based processor address space, there is a byte within the PIB that controls the generation of the INTA transaction (see Section 2.7.1). Byte loads from the INTA address will cause an Itanium architecture-based processor to emit the INTA transaction onto the processor system bus. External interrupt controllers, such as the 8259 interrupt controller, must respond with the actual interrupt vector, which will become the data loaded by the load instruction.

After servicing an interrupt, the software must issue an EOI directly to the 8259 interrupt controller to clear the interrupt. Then the software must write to the processor EOI register to clear the interrupt in the processor SAPIC. This is the same as servicing any other external interrupt.

2.7 SAPIC Memory Map

An Itanium architecture-based system has up to 2^{63} bytes of addressable physical memory. The memory is divided into compatibility, low extended, medium extended, and high extended memory. The area of focus is the medium extended memory area (see Figure 2-6) which covers the 16 MB to 4 GB range (0x0100_0000 to 0xFFFF_FFFF). The Default PIB and the Default I/O xAPIC Configuration Space are located in this area.

Figure 2-6. Expanded View of the Medium Extended Memory Range

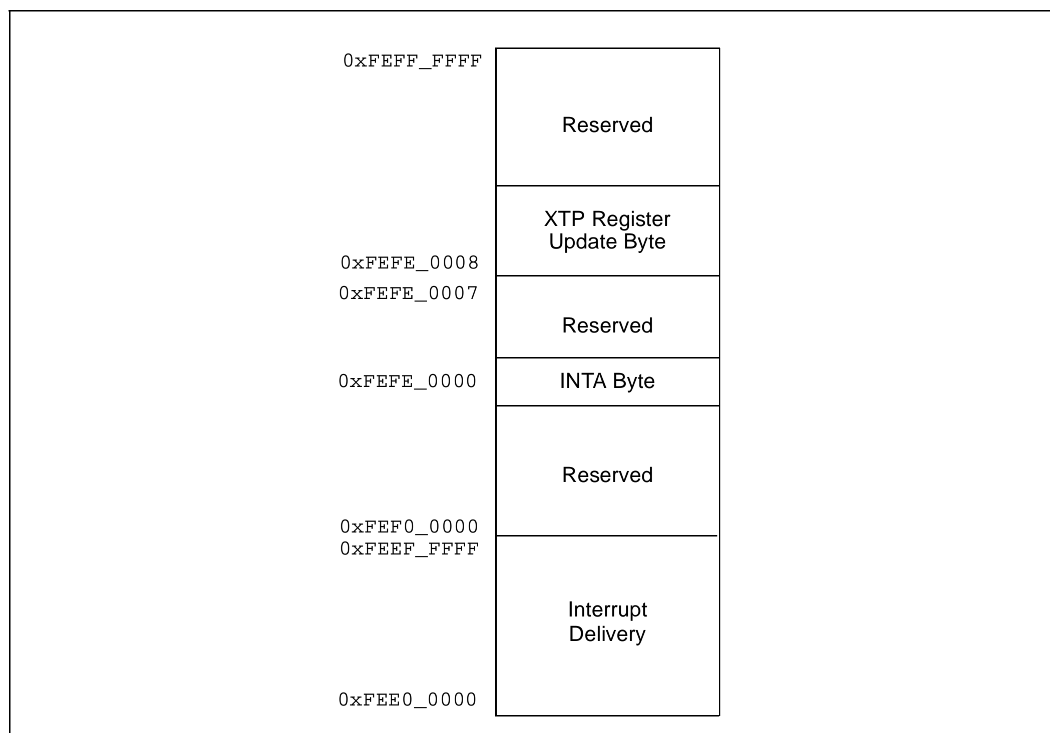


2.7.1 Processor Interrupt Block

The PIB defines the location of the Interrupt Delivery regions. This 2 MB block is relocateable in the processor, but corresponding changes are required in the PAL and chipset; the address given is the default address if it is not relocated. Although PAL services are provided to relocate this block, this block should not be relocated by SAL or software for compatibility. This address block must have the uncacheable attribute regions (shown in Figure 2-7).

Note: No memory resources can be mapped to the address space where this PIB is located. Reads into this space from the I/O bus will result in a target abort.

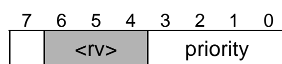
Figure 2-7. Default Processor Interrupt Block



2.7.1.1 XTP Register Update (Optional)

The XTP register is used for platform level interrupt redirection. This capability is explained in Section 2.4.2. The format of the eight bits of data to be stored is shown in Figure 2-8.

Figure 2-8. XTP Register Format



1 = Disable

0 = Enable

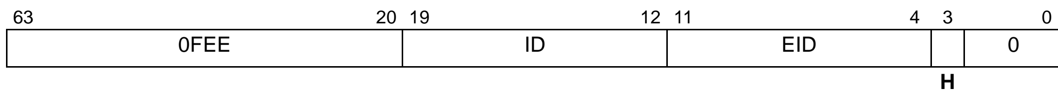
2.7.1.2 INTA Byte (Optional)

Byte loads from the INTA address will cause an Itanium architecture-based processor to emit the INTA cycle onto the processor system bus. External interrupt controllers, such as the 8259 interrupt controller, will respond with the actual interrupt vector, which will become the data loaded by the load instruction.

2.7.1.3 Interrupt Delivery Area

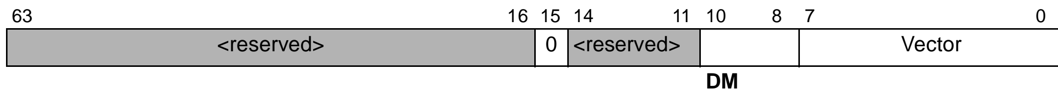
Stores into this memory space are directed as processor system bus interrupt transactions. A processor generates IPIs by storing a 64-bit interrupt command to an 8-byte aligned address in this region. The address specifies the target processor to receive the interrupt message, as shown in Figure 2-9. The format of the 64-bit interrupt command is shown in Figure 2-10. All IPIs are treated as edge-triggered. Writing to an improperly aligned address within this region or storing less than 64 bits, is an invalid operation. Reading from this region will result in an invalid operation. See the *Intel® Itanium® Architecture Software Developer's Manual* for more detail about IPIs.

Figure 2-9. Format of Address to Generate IPIs



- **Hint (H):** When this bit is set, it allows the platform to redirect the interrupt to another processor as the destination identified by the ID/EID fields. See Section 2.4.2.
- **Destination ID and EID:** Specifies a processor ID and EID. Processors receiving an interrupt message compare these fields to the corresponding fields in their LID registers to determine if the interrupt is the interrupt is to be received by them.

Figure 2-10. Format of IPI Data Being Stored



Note: Certain delivery modes will only operate as intended when used in conjunction with a specific trigger mode. These restrictions are indicated in the list for each delivery mode.

- **Vector:** An 8-bit field containing the interrupt vector.
- **Delivery Mode (DM):** The delivery mode is a 3-bit field that specifies how the processors listed in the destination field should act upon reception of this signal.

000 (INT) Deliver the signal to the processor listed in the destination. Trigger mode can be edge or level. The vector value can range between 0x10 and 0xFF.

010 (PMI) Deliver the interrupt on the PMI signal of the processor listed in the destination. The interpretation of the vector field is implementation defined: the vector ranges from 0x1-0xF (0x1 - 0x3 are useable by SAL; all other values are Intel reserved). The trigger mode must be edge. This delivery mode is compatible with the SMI delivery mode of the APIC when the vector is 0x0.

100 (NMI) Deliver the signal on the NMI signal of the processor listed in the destination; vector information is ignored, but the vector read by the Itanium architecture-based processor will be 0x2. NMI is always delivered as an edge triggered interrupt; the trigger mode field is ignored.

101 (INIT) Deliver the signal on the INIT signal of the processor listed in the destination; vector information is ignored.

111 (ExtINT) Deliver the signal to the processor listed in the destination as an interrupt that originated in an 8259 compatible external interrupt controller. Be aware that the software-initiated INTA transaction that corresponds to this ExtINT delivery will be routed to the 8259 compatible external controller that is expected to supply the vector and not the processor that originated the IPI. A delivery mode of ExtINT requires an edge trigger mode. SAPIC architecture supports only one ExtINT source in a system. Vector information is ignored, but the vector read by the Itanium architecture-based processor will be 0x0.

2.7.2 I/O xAPIC Configuration Space

The I/O xAPIC base address defines where the individual I/O xAPIC device configuration registers are located. The I/O xAPIC may be relocated to any address depending on the chipset implementation.

The base addresses for each I/O xAPIC are reported to the operating system by the platform using the ACPI Multiple APIC Descriptor Table. Please refer to the *Advanced Configuration and Power Interface Specification v2.0* for more information.

Platform Level Implementation 3

3.1 Platform Interrupt Delivery Overview

This chapter describes how various interrupt messages are delivered to the Local SAPIC units. As described in Section 2.2, interrupts can originate from remote I/O devices (I/O xAPIC, MSI), other processors (IPI), local I/O devices (PMI, NMI, LINT0/1 pins), and from the processor itself. Since interrupts originating from the local processor do not involve the platform, they are not discussed in this chapter.

3.2 External I/O Interrupt Delivery

The following section describes how an interrupt originating from an external I/O device is delivered to a single system bus for service by a processor.

3.2.1 I/O xAPIC Duties

All external pin-based I/O interrupts must be routed through the I/O xAPIC. When a signal is asserted on an I/O xAPIC input interrupt pin, the signal must remain asserted for a minimum of one bus clock cycle in order for it to be recognized. The delivery status bit within the I/O xAPIC is set when this occurs and remains pending until its interrupt message is sent on the I/O bus. The I/O xAPIC accepts level- and edge-triggered interrupts as pin input, but converts the level-triggered interrupts into an interrupt message (identical to an edge-triggered interrupt) that gets delivered to the processors.

Upon receiving an interrupt signal at the pin, the I/O xAPIC selects the corresponding entry in the I/O RT and uses the information in the I/O RTE to translate an incoming interrupt signal into the appropriate interrupt request message. This message is then sent across the I/O bus as a memory write transaction.

The ID and EID fields of the interrupt transaction uniquely identify a particular processor in the platform. The two fields can be considered as one 16-bit field.

For platforms that have multiple processor system buses, the Itanium architecture-based operating systems must be used to program both the ID and EID fields of the RTEs for routing interrupt transactions to the appropriate system bus. In this case, the concatenation of the ID/EID fields can be thought of as a single 16-bit identifier. One potential implementation would be to use EID as the system bus identifier and ID as the processor identifier on the bus.

3.2.2 Interrupt Transaction on the I/O Bus

Interrupts are signaled on the I/O bus with a 32-bit default address at 0xFEEx_xxxx, where **x_xxxx** is the encoding of the destination processor. This default address is the beginning of the PIB. The bridge controllers can use the leading bits in the address as a signature for interrupt transactions.

When the I/O xAPIC sends interrupt messages on the I/O bus, the messages are implemented as 32-bit transactions. This ensures that the I/O bus maintains ordering rules for memory write cycles that were generated prior to the interrupt message. The address of the memory write cycle is

determined by the destination ID fields located in the I/O xAPIC's RT. In addition to the destination ID, the Redirectable Hint bit is also transmitted. The Redirectable Hint bit informs the bridge controller that it may optionally change the destination ID. The vector, delivery mode, and trigger level are transferred during the data phase of the memory write transaction. The delivery mode is always specified as a fixed delivery mode and the Redirectable Hint bit from the RT is sent when the I/O xAPIC sends the interrupt. The 32-bit write cycle data is defined also by entries in the RT.

The address (AD[31:00]) for the 32-bit memory write cycle is defined in Figure 3-1. The data of the 32-bit write is defined by the lower 32-bits of the corresponding RTE in the I/O xAPIC, as shown in Figure 3-2.

Figure 3-1. Format of Interrupt Address on I/O Bus

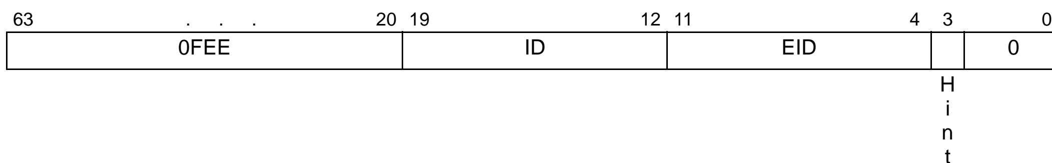
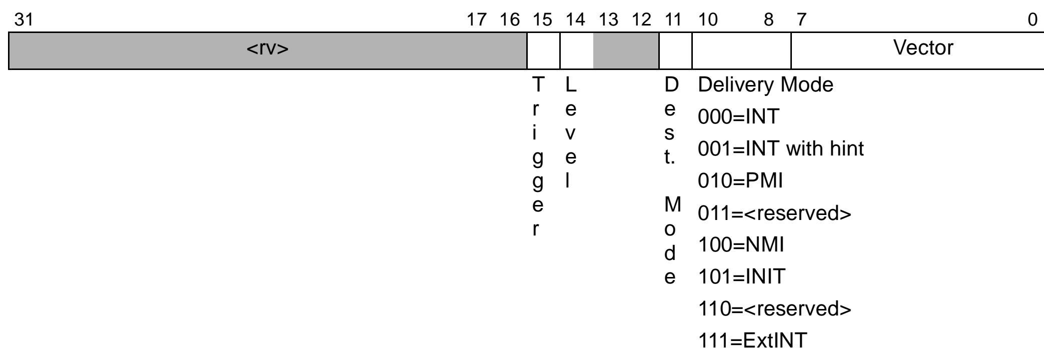


Figure 3-2. Format of Interrupt Data on I/O Bus



3.2.3 Bridge Controller Duties

The implementation of the platform-level lowest priority bridge controller interrupt delivery is optional. The Redirectable Hint bit is only a hint to the bridge controller.

On a bridge controller that does not implement the XTP registers, it will pass the interrupt transaction through to the processor system bus unmodified (except for clearing the Redirectable Hint bits and converting them to processor system bus transactions when necessary). The bridge controller does not need to examine the interrupt transaction.

When the bridge controller of the system bus with the target processor receives the interrupt message, it detects that the Redirectable Hint bit is set and can deliver the interrupt transaction to the lowest priority processor on that system bus. The bridge controller uses the XTP registers located in the chipset to determine which processor the interrupt is to be redirected to. After the lowest priority processor is identified, the bridge controller will modify the ID to identify the new destination processor. There is an XTP register for each processor on the system bus which is used to reflect the lowest priority processor. These registers are placed in the interrupt delivery path of the platform and are used to redirect an interrupt transaction to the lowest priority processor. Lowest priority delivery is optional at this platform level.

The processors on the same system bus with their corresponding XTP registers enabled are able to accept redirected interrupt transactions. If the XTP register is disabled through software control, then the corresponding processor will not be considered as a redirected destination. The interrupt transaction will not be redirected if all of the XTP registers are disabled.

If the Redirectable Hint bit is set, the bridge controller must always clear this bit within the bus transaction before placing the transaction on the system bus. This will allow the redirection of IPIs. Thus, only after the chipset has intercepted an interrupt transaction and cleared the Redirectable Hint bit, can processors accept any interrupt transactions.

3.2.4 Interrupt Transactions on the System Bus

The bridge controller issues an interrupt transaction to interrupt a processor on the system bus. With the SAPIC architecture, the ID and EID fields specify a specific processor as the destination for an interrupt transaction. When a processor recognizes the interrupt transaction, it compares the ID/EID fields with its LID register to determine if it is indeed the targeted destination.

3.3 IPI Delivery

Local SAPIC units can exchange interrupt information via the system bus. The message format is not visible to software. Software generates IPIs by storing a 64-bit interrupt command to an 8-byte aligned address in the interrupt delivery region of the processor I/O block. The default address range for the Interrupt Delivery region is 0xFEE0_0000 to 0xFEEF_FFFF. The address being stored into designates the target processor which will receive the interrupt message. All IPIs are treated as edge-triggered. Writing to an improperly aligned address within the Local SAPIC address range or storing less than 64 bits, is an invalid operation. Reading from the Local SAPIC address range will result in an invalid operation. Refer to the *Intel® Itanium® 2 Processor Hardware Developer's Manual* for the format of the address store.

3.4 Ordering Issues

Although delivering interrupts through the system bus has eliminated many of the potential ordering issues with regard to data and interrupt synchronization, it does not completely eliminate the possibility that data traffic and interrupt traffic can get out of sync with each other. For example, in a distributed system where there are multiple paths for data and interrupt traffic, an I/O device can send the data to the memory system via one path, and after the data transfer is completed, the interrupt is sent to the processor via a different path. The platform must ensure that the order of the data transfer and the interrupt message is maintained.

3.5 Platform Topology and Routing Issues

The SAPIC architecture uses two separate elements to identify the destination processor of each interrupt, although both of these elements are packaged together in the ID and EID fields. One element uniquely identifies the processor on one system bus, while the remaining bits forming the other element uniquely identify the system bus within the whole platform. The architecture does not specify the width of these different elements because the width is implementation dependent. For example, in an implementation where only two processors are supported on one system bus,

only one bit is required to uniquely identify the processor on the bus. In an implementation where eight processors are supported on one system bus, three bits are required to uniquely identify the processor.

The SAPIC architecture uses a memory-mapped interface for interrupt routing. Interrupt messages outside of the system bus are just normal memory transactions. In a system with distributed memory, consideration must be given to the way these interrupt transactions are routed throughout the system.

If the memory traffic routing agent in a system is not intelligent with regard to interrupt messages, then all interrupt messages are routed the same as normal memory transactions. If the distributed memory in the system is interleaved, then the granularity of the memory interleave will have a direct impact on the distribution of the interrupts. If the memory in such a system is not interleaved and exists as contiguous blocks, then the processors on the same bus as the memory system that contains the PIB physical address will be the destination for all interrupt messages.

In order to route interrupt messages independently of the normal memory traffic, there must be some intelligence in the system routing agent to recognize interrupt messages and treat them differently.

Differences from APIC A

The SAPIC architecture differs from the APIC architecture in several ways, but these differences are only visible to software running in an Itanium architecture system environment.

A.1 Addressing

APIC supports Logical Mode addressing, along with destination format and ID registers and the destination shorthand field of the Interrupt Command Register (ICR), while SAPIC only supports physical addressing.

All APIC units on one private serial APIC bus must have a unique ID, including all Local APIC units and I/O APIC units, for a maximum of 15 units per APIC bus (an ID of 0x0F indicates broadcast in physical mode). The SAPIC has no such restrictions — the I/O xAPIC units do not need an ID, and there are no similar constraints on the ID/EID fields of the Local SAPIC units.

A.2 Interrupts

A local APIC is designed to handle a local timer interrupt. This 32-bit programmable timer generates periodic interrupts and a one-shot interrupt when the timer count reaches zero. The SAPIC architecture does not implement this feature.

The vector number for spurious interrupts is programmable in the APIC, but it is fixed at 15 with the SAPIC.

The Local APIC unit, in conjunction with the processor, automatically vectors an interrupt to different ISRs. The Local SAPIC unit does not provide hardware vectoring of interrupts. A new IVR is added to the Local SAPIC unit.

APIC interrupts are typically processed one at a time by the processor, but SAPIC interrupts can be processed in batches by reading the IVR.

Local APIC units generate the INTA cycle automatically when the ExtINT interrupt is received. Local SAPIC units generate the INTA cycle only when the INTA byte is read.

The lowest 16 interrupt vectors are not used in the APIC; these vectors are reserved for pre-assigned non-vectored interrupts (such as ExtINT and NMI) in the SAPIC.

A.3 Interrupt Delivery

The APIC architecture relies on a three-wire serial APIC bus to deliver interrupts to the processor as well as for other services. The SAPIC architecture eliminates this private bus and uses the I/O and system bus for interrupt delivery. Without the APIC serial bus, interrupts can be delivered much faster and the platform can be simplified because there is not another bus to be routed around the platform.

A processor generates IPIs to pass messages or interrupts to other processors. In an APIC system, this was initiated by writing to an ICR within the APIC. The SAPIC no longer uses a command register but uses a memory store to generate an IPI. For delivery of the interrupt to its correct destination, the address of the store as well as the information being stored contains the 32-bit routing information. With the APIC, it takes two 32-bit writes to completely program the ICR; with the SAPIC, a single 64-bit store is used (unaligned or shorter accesses to the IPI block are undefined).

The APIC architecture supports a lowest priority delivery mode by implementing this capability through the arbitration mechanism in the APIC bus. Since the SAPIC architecture does not use a private bus, the implementation of delivering an interrupt to the lowest priority processor is done at the platform level with control by platform firmware. See Section 2.4.2 for more information.

The APIC supports the SMI delivery mode; the SAPIC does not support this delivery mode, but it supports the equivalent PMI delivery mode.

The APIC supports the start-up delivery mode; the SAPIC does not support this delivery mode.

A.4 Messages

Software must poll the ICR to ensure that the previous interrupt message was sent before writing into the ICR with the APIC; such polling is not necessary with the SAPIC.

An APIC design allows for remote read messages. A remote read message is a request to read the contents of another processor’s local APIC register. This remote capability is not implemented in a SAPIC design.

The APIC architecture allows processor messages to be broadcast over the APIC bus to a specific agent or multiple agents on the APIC bus. Message broadcasting is not supported in a SAPIC design due to the implementation complexity in a scalable fashion across multiple system buses.

A.5 Processor Support

The SAPIC architecture supports 65536 processors while the IA-32 APIC architecture is able to support 15 clusters of 4 processors each (60 total). For an IA-32 APIC-based system, external interrupt logic would be required to support greater than 60 processors.

A.6 Registers

A processor compares its processor ID against the destination ID of the interrupt to determine if it is the processor to service the interrupt. This processor ID is a 4-bit ID in an APIC design, and a 16-bit ID in a SAPIC design. This 16-bit field consists of the destination ID and EID. Operating systems running on platforms implementing multiple processor nodes must program the ID and EID fields to ensure interrupt routing to the appropriate system bus.

Local APIC registers are memory-mapped, while Local SAPIC registers are control registers.

There are many more registers visible to the OS with APIC. Only a subset of the equivalent function control registers are OS visible with SAPIC.

The TPR of the SAPIC has been extended to mask the non-vectored interrupts.

Writing into the EOI register of the Local APIC unit will broadcast an EOI message on the private serial APIC bus to send a deassert message to the I/O APIC that initiated the level-triggered interrupt. Writing into the EOI register of the Local SAPIC unit does not send any messages out. This change may result in the processor receiving fewer spurious interrupts than with the APIC.

The APIC handles errors detected in the private serial APIC bus or the remote read delivery mode. The SAPIC does not handle errors because these features are not supported. SAPIC does not include support for the error interrupt or the error status register.

A.7 Support Disabling

With both a SAPIC and APIC, the local controller is integrated into the processor. The APIC support can be disabled at reset based on the state of the APIC-enable signal. The interrupts are then processed using 8259 interrupt controllers. The Local SAPIC unit is an integral part of the processor and cannot be disabled.