AP-732

APPLICATION NOTE

I/O APIC Emulation Software for the i960® RP Microprocessor

Warren Gilbert
Intel Technical Marketing Engineer

Intel Corporation
Mail Stop CH6-319
5000 W. Chandler Blvd.
Chandler, Arizona 85226

May 31, 1996

Order Number: 272905-001

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

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

or call 1-800-879-4683

I/O APIC Emulation Software for the i960® RP Microprocessor

FIGURES

TABLES

EXAMPLES

1.0 INTRODUCTION

This application note describes how to program the I/O APIC (Advanced Peripheral Interrupt Controller) port integrated on the i960® RP microprocessor. The i960 RP processor provides a hardware port that requires emulation software to achieve full I/O APIC functionality.

This application note supplements existing information available on APIC and the i960 RP processor's implementation of I/O APIC. To implement I/O APIC on the i960 RP processor, refer to the documents listed in Section 10 of this document.

Before getting into the details of programming the I/O APIC on the i960 RP processor, an overview of APIC architecture is presented. After the APIC overview the application note presents details on the I/O APIC implementation on the i960 RP processor. Detail information is provided on the emulation software, the hardware port and initialization requirements, and pseudo code for the Interrupt Service Routine.

1.1 Terminology Used in this Document

- 80960RP - the i960 RP processor
- Core processor - the i960 Jx processor integrated into the 80960RP
- APIC, I/O APIC - the Advanced Peripheral Interrupt Controller port on the 80960RP

2.0 APIC OVERVIEW

The APIC is based on a distributed architecture in which interrupt control functions are distributed between two basic functional units, the local unit and the I/O unit, as shown in Figure 1. The local and I/O units communicate through a bus — the APIC bus — as shown in the figure. In a multiprocessor system, multiple local and I/O APIC units operate together as a single entity, communicating via the 3-wire APIC bus. The APIC units are collectively responsible for delivering interrupts from sources to interrupt destinations throughout the multiprocessor system.

APICs exist in many forms; it is integrated in Intel processors such as the Pentium 735/90, 815/100 and i960 RP processors. APICs are also available in discrete form in the Intel 82489DX APIC.

This application note concerns itself with the I/O APIC as implemented on the 80960RP.

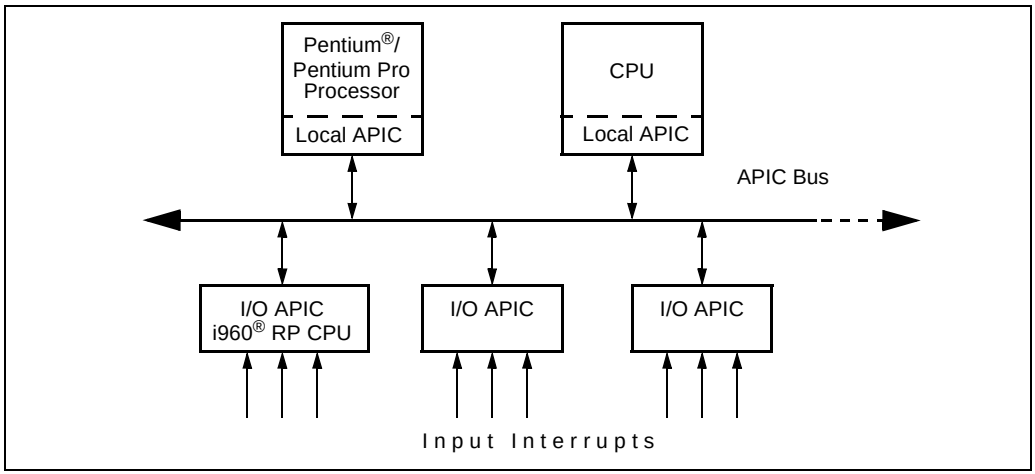


Figure 1. APIC Architecture

The APICs help achieve the goal of scalability by:

- Off-loading interrupt-related traffic from the memory bus, making the memory bus more available for processor use.
- Helping processors share the interrupt processing load with other processors.

The APIC architecture's main features are:

- APIC provides multiprocessor interrupt management for Intel Architecture CPUs such as the Pentium® Processors, providing both static and dynamic symmetric interrupt distribution across all processors.
- Dynamic interrupt distribution includes routing of the interrupt to the lowest-priority processor.
- APIC works in systems with multiple I/O subsystems, where each subsystem can have its own set of interrupts.
- APIC provides inter-processor interrupts, allowing any processor to interrupt any processor or set of processors, including itself.
- Each APIC Interrupt Input pin is individually programmable by software as either edge or level triggered. The Interrupt Vector and Interrupt Steering information can be specified per pin.
- APIC supports a naming/addressing scheme that can be tailored by software to fit a variety of system architectures and usage model.
- APIC supports system-wide processor control functions related to Non-Maskable Interrupts (NMI), INIT, and System Management Interrupt (SMI).
- APIC co-exists with the 8259A Programmable Interrupt Controller (PIC) to maintain PC compatibility.
- APIC provides programmable interrupt priority (vectors) for each Interrupt Input pin. Since the APIC programming interface consists of two 32-bit memory locations, I/O APIC functionality can be emulated by the i960 core processor in the 80960RP.

3.0 I/O APIC ON THE i960 RP PROCESSOR

As described in the *i960® RP Microprocessor User's Manual* (272736), the 80960RP is a single-chip intelligent Input/Output subsystem. It consists of the following functional units:

- i960 JF microprocessor core (the "core processor")
- PCI to PCI Bridge Unit (BU)
- Address Translation Unit for direct access between PCI and the 80960 local bus
- Memory Controller
- Messaging Unit
- PCI arbiter for secondary PCI bus
- DMA Controller
- I/O APIC Interface (BIU)
- I^2C Interface
- Interrupt Routing

The I/O APIC unit provides interrupt pins on which I/O devices send interrupts into the system. The 80960RP interrupt pins (S_INT[D:A]#/ XINT7:0#) require level-sensitive interrupt detection. Because of this, APIC Delivery Modes requiring edge-sensitive Input Interrupts are unavailable. When an application requires Delivery Modes that use edge-sensitive Input Interrupts, external hardware is required. The external hardware must latch the edge-sensitive Input Interrupt so it appears as level-sensitive at the XINT7:4# pins. The external hardware must also provide the capability for software to clear this interrupt after it has been serviced. As is the case with all 80960RP input level-sensitive interrupts, software must clear the interrupt's bit in the IPND register.

The I/O APIC port on the 80960RP is a Bus Interface Unit (BIU), which provides for communication between the local bus and the 3-wire APIC bus. It provides two basic functions:

- It gives the core processor the ability to send an interrupt message out onto the APIC bus and optionally be interrupted when the message has been sent. The i960 core processor then reads the resulting status of the message transmission to check for errors.
- It can also receive End Of Interrupt (EOI) messages from the APIC bus and optionally interrupt the i960 core processor to inform it that an EOI vector is available.

The 80960RP is a dual-function PCI device, and informs PCI configuration software of its dual-functionality by providing two PCI device Configuration Headers. One header is for the 80960RP's PCI-to-PCI bridge and the other header is for its Address Translation Unit (ATU). To make the I/O APIC's existence on a PCI bus known to PCI initialization and operating system software, the APIC has been given a PCI Class Code. To use the APIC PCI Class Code, the I/O APIC "pirates" the ATU's Configuration Header. It does this by writing the I/O APIC's PCI Class Code into the Class Code field of the ATU's Configuration Header. Now PCI configuration software sees the 80960RP as a dual-function device:

- A PCI-to-PCI Bridge
- An APIC

The overall control of the I/O APIC is handled by emulation software executing within the 80960RP.

4.0 I/O APIC EMULATION

A basic I/O APIC unit is emulated by the 80960RP with the APIC BIU and emulation software. The I/O APIC unit consists of:

- A set of interrupt pins
- Interrupt Redirection Table
- An I/O APIC BIU for sending and receiving APIC messages from the APIC bus

The I/O APIC BIU is a dedicated hardware unit in the 80960RP and acts as an interface from the 80960RP local bus to the APIC bus. The I/O APIC BIU provides the Interrupt Input pins on which I/O devices inject interrupts into the system. The I/O APIC also contains a Redirection Table (RDT) with an entry for each Interrupt Input pin. Each entry in the RDT can be individually programmed as to what vector and priority the interrupt has, which of all possible processors should service the interrupt, and how to select that processor (statically or dynamically). The information in the table is used to broadcast a message to all Local APIC units. I/O APIC software disables the Input Interrupt on the 80960RP that caused the APIC message to be sent. After the processor services the interrupt it clears the source of the interrupt and sends an APIC End Of Interrupt (EOI) message to the I/O APIC. When EOI interrupts are enabled, reception of an EOI message causes an interrupt on the 80960RP. Software then reads the EOI vector from the APIC BIU's EOI Vector Register (EVR). When the EVR vector matches the vector for the message just sent, software re-enables servicing of that PCI interrupt. Thus, overall control of the I/O APIC is handled by emulation software executing within the 80960RP.

The I/O APIC has two registers implemented in the 80960RP Messaging Unit. These registers are the APIC Register Select Register (ARSR) and the APIC Window Register (AWR). These are the only I/O APIC registers directly visible to host software and are only accessible from the primary PCI bus. A primary PCI based host can initialize the RDT, read selected I/O APIC registers for status, and enable I/O APIC servicing of Input Interrupts using these two registers.

When an ARSR in the Messaging Unit is written from the primary PCI bus, an interrupt is asserted to the 80960RP and the Messaging Unit locks out all other PCI accesses to the two Messaging Unit APIC registers. The emulation software then reads the logical APIC register at the offset contained in the ARSR and stores the value back into the AWR. The emulation software then clears the interrupt and the Messaging Unit releases the interlock mechanism to allow additional accesses to the APIC registers. The emulation software must also keep the value of the AWR updated when the Redirection Table changes due to interrupt activity.

When the AWR is written, an interrupt is asserted to the i960 core processor and the Messaging Unit locks out all other PCI accesses to the two Messaging Unit APIC registers. The emulation software reads the values of the ARSR and AWR, updates the appropriate register, and then clears the interrupt to release the interlock.

5.0 I/O APIC EMULATION SOFTWARE

The scope of this application note is limited to an I/O APIC implementation that supports up to four PCI Input Interrupts. To handle more than four Input Interrupts, hardware external to the 80960RP is required. For information on the issues of implementing more than four Input Interrupts, refer to Appendix A of this document. The example emulation software confines itself to an environment in which the 80960RP is dedicated to the I/O APIC application. In such an application, the 80960RP may be configured as a PCI Bridge with the i960 core processor executing only I/O APIC emulation software.

The example emulation software is layered on the MON960 monitor. MON960 handles i960 core processor initialization

that sets up an environment for the emulation software. When your implementation does not use MON960, it is necessary to perform initialization functions done by MON960, such as setting up the Initial Memory Image and the Process Control Block. Refer to the *i960® RP Microprocessor User's Manual* (272736) for initialization details.

The diagram in Figure 2 shows the 80960RP's I/O APIC in a system environment, with Primary and Secondary PCI buses, and the APIC Bus as interfaced to other system components. The host processor, shown on the primary PCI bus, is a generalization of functions residing on the primary PCI bus that need access to the I/O APIC. The host may be PCI configuration software in the form of the system BIOS, it may be part of the operating system, or it may be custom system software. As shown in Figure 2, the 80960RP's Messaging Unit provides the host access to the I/O APIC.

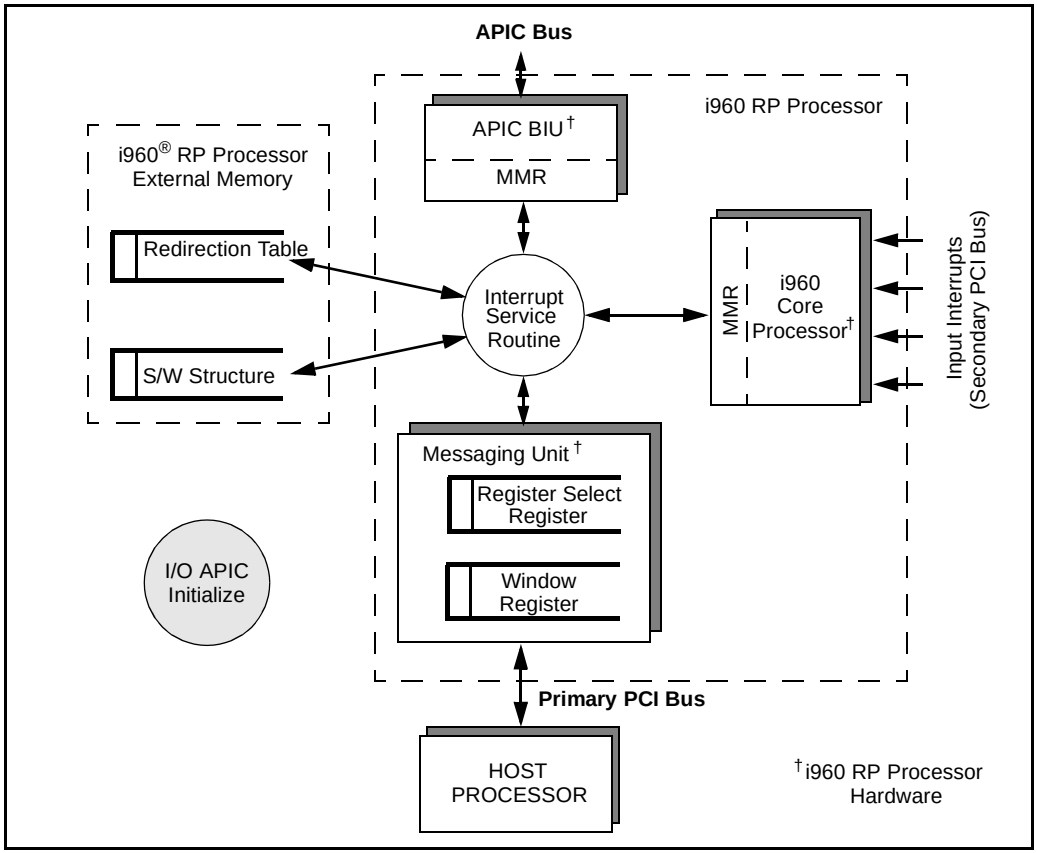


Figure 2. i960 RP Processor's I/O APIC Emulation Diagram

PCI configuration software needs to know of the existence of the I/O APIC. The PCI configuration software does this by reading the PCI Class Code from the 80960RP's PCI Configuration Header and determining that the PCI class code is that of the I/O APIC. I/O APICs are mapped to give shared access from all processors. I/O APIC devices are mapped in system memory between addresses FEC0 0000-FECF FFFFH. The default base address of the first I/O APIC is FEC0 0000H with subsequent I/O APIC addresses assigned in 4 Kbyte increments. The host also needs to know where the Messaging Unit exists in the PCI address space and the size of it. It can determine this by

using a Type 0 configuration command to read Base Address 0 (Primary Inbound ATU Base Address) from the 80960RP's Configuration Header. The first 4 Kbytes of the inbound address translation window of the primary ATU are reserved for the Messaging Unit. The two APIC registers that are accessible to the host are located in the Messaging Unit address space. Also, the PCI configuration software needs to know the mapping of Input Interrupts to Redirection Table entries. For the emulation software, this is a compile time definition and therefore must be part of an overall system definition that is made visible to the PCI configuration software (Refer to the Intel *Multiprocessor Specification* for details on creating a multiprocessor configuration table).

Another I/O APIC system interface is the Secondary PCI bus. This is the source of PCI interrupts. In the application note these interrupts are S_INT[D:A]#/ XINT3:0#. Before being configured by the host to handle these PCI interrupts, the 80960RP routes these interrupts to the Primary PCI bus (P_INT[D:A]#). In this state, the host is able to handle S_INT[D:A]#/ XINT0:0# interrupts until it enables the I/O APIC.

The remaining 80960RP interface is to the APIC bus. The APIC bus is a 3-wire synchronous bus connecting all APICs (all I/O APIC Units and all Local APIC Units). Two of these wires are used for data transmissions, and one wire is the clock. For bus arbitration, the APIC uses only one of the data wires. The maximum APIC bus speed is 16 MHz.

The 80960RP's architecture lends itself to an interrupt driven implementation of APIC emulation software. Interrupts, in general, allow the i960 core processor to execute other software, only stopping to service interrupts when told to do so by the device itself. When servicing is complete, the 80960RP resumes executing code exactly where it left off, prior to the interrupt.

The example solution is implemented as a single Interrupt Service Routine (ISR). The ISR services only I/O APIC related interrupts such as the following:

- S_INT[D:A]#/ XINT3:0# (IPND.xip3-0)
- XINT7# (IPND.xip7)

The above interrupts are reported in the IPND register along with other 80960RP interrupts. S_INT[D:A]# are used to receive PCI interrupts from the secondary PCI bus. XINT7# receives internal interrupts from the APIC BIU and the Messaging Unit. In the example solution, all I/O APIC interrupts have the same priority and do not interrupt each other. To avoid unwanted side effects, the ISR must take care to affect only those shared register bits that are peculiar to the operation of the I/O APIC.

The 80960RP's I/O APIC emulation software consists of an Initialization Routine and an Interrupt Service Routine (ISR). Upon reset APIC hardware and software components are set to their initialized state. XINT7# interrupts are enabled and S_INT[D:A]#/ XINT3:0# interrupts are bypassed to the host. A host on the Primary PCI Bus, through the Messaging Unit, initializes the Redirection Table (RDT) thus enabling the servicing of Input Interrupts. The RDT contains a number of entries (RDTE) which are equal to the number of Input Interrupts, which in the example solution is four (4). Each RDTE contains all the information necessary to send a message on the APIC bus.

After initialization, control of the I/O APIC is handled by the ISR. The ISR accesses Memory-Mapped Registers when handling interrupts and controlling the APIC BIU. The software data structures and the RDT, located in the 80960RP's external memory, are accessed by the ISR during normal operation. The Interrupt Input pins are either just the 80960RP input pins S_INT[D:A]#/ XINT3:0#, XINT7:4#, and NMI# or are expanded through external hardware, feeding into the 80960RP. The APIC Bus Interface Unit is dedicated hardware in the 80960RP and acts as an interface from the 80960 local bus to the APIC bus.

When the i960 core processor receives an Input Interrupt that it determines should be sent as an APIC message, it vectors to the ISR. The ISR looks up the information related to that interrupt in the Redirection Table, and writes that information to the APIC Bus Interface Unit which then sends the correct message on the APIC bus.

5.1 Input Interrupts

The Emulation Software must handle the S_INT[D:A]#/ XINT3:0# (IPND.xip3-0) Input Interrupts. For correct operation of the 80960RP, the i960 core processor external interrupt pins (S_INT[D:A]#/ XINT3:0# AND XINT7:4#) must be programmed for Dedicated Mode of operation, level-sensitive interrupts, and fast Sampling Mode. With the i960 core processor in the Dedicated Mode, each external interrupt pin can be assigned a vector number. This vector number, not to be confused with the vector number in the Redirection Table, is used to invoke the Interrupt Service Routine associated with that interrupt. In

the example solution, all PCI Input Interrupts are programmed to invoke the same ISR. The ISR reads the IPND register to get the event(s) that caused the interrupt. It then uses the event to select the information stored in the Redirection Table to generate an APIC message. S_INT[D:A]#/ XINT3:0# and XINT7# are programmed as level-sensitive Input Interrupts and, as such, their signal remains asserted until the source of the interrupt is serviced and the interrupt signal is ultimately cleared. S_INT[D:A]#/ XINT3:0# interrupts are cleared by the host. XINT7# interrupt sources are cleared by the ISR. To ensure proper the operation of the I/O APIC, the emulation software must selectively disable S_INT[D:A]#/ XINT3:0# interrupts until the host has handled and cleared the S_INT[D:A]#/ XINT3:0# interrupt. The ISR disables level-sensitive interrupts by masking their respective bits in the IMSK register. **Failure to mask these level-sensitive interrupts could cause the 80960RP to spend all its time jumping in and out of the ISR**.

The core processor must be programmed to map Input Interrupts to the ISR that processes the interrupt. With level-sensitive interrupts the ISR can read IPND to determine which input caused the interrupt. Input Interrupts must be associated with Redirection Table entries. Table 1 shows the association of Input Interrupts with Redirection Table entries in our example solution. To associate Input Interrupts with RDT entries, the programmer must have knowledge of the overall interrupt architecture.

Table 1. PCI Interrupt Mapping

Input Interrupt	RDT Entry
S_INTA#/ XINT0#	0
S_INTB#/ XINT1#	1
S_INTC#/ XINT2#	2
S_INTD#/ XINT3#	3

NOTE: The interrupt mapping in Table 1 is for demonstration purposes only, and does not necessarily represent a practical implementation. The mapping implies four devices are on the secondary PCI bus, and that each device has a dedicated interrupt line. The probability of each PCI device having a dedicated interrupt line is not likely, since in a PCI environment all devices on the PCI bus could be single-function devices. The PCI specification requires that single-function PCI devices must use the INTA# interrupt (never INTB#, INTC#, or INTD#). Therefore, when there are, for example, four single-function devices on the PCI bus they all share INTA# interrupt and are not distributed across INT[D:A]#. Refer to the *PCI Local Bus Specification* for additional information on PCI interrupt related issues.

Table 2 lists initialize and control interrupt registers.

Table 2. Relevant i960 RP Processor Interrupt Registers

Register Name	Description	Local Bus Address
IMAP0	Interrupt Map Register 0	FF00 8520H
IMAP1	Interrupt Map Register 1	FF00 8524H
IMAP2	Interrupt Map Register 2	FF00 8528H
ICON	Interrupt Control Register	FF00 8510H
IPND	Interrupt Pending Register	FF00 8500H
IMSK	Interrupt Mask Register	FF00 8504H
X7ISR	XINT7 Interrupt Status Register	0000 1704H
PIRSR	PCI Interrupt Routing Select Register	0000 1050H

NOTE: Addresses 0000 1000H through 0000 17FFH are Peripheral Memory-Mapped Registers (PMMR) and provide full accessibility from the Primary and Secondary ATU, and the i960 core processor. Addresses FF00 0000H through FFFF FFFFH are reserved for specific Memory-Mapped Registers.

The following descriptions are for Table 2 register names:

ICON - a Memory-Mapped Control Register that sets up the Interrupt Controller. It is used to select Interrupt Mode, Signal-Detection Mode, enable/disable interrupts, mask operations, vector cache enable, and set Sampling Mode. It is automatically loaded at initialization time, from the control table, in external memory. Software can also manipulate this register with load/store type instructions.

IMSK - this Interrupt Mask Register selectively masks hardware-requested interrupts. It provides a mechanism for masking individual bits in the IPND register. **Software should use atomic-modify type instructions when accessing this register**.

IPND - the Interrupt Pending Register posts hardware-requested interrupts. In the Dedicated Mode, interrupts are those originating from the eight external dedicated sources and the two timer sources. **Software must use the ATMOD instruction when accessing this register**.

IMAP - used to program the vector associated with the interrupt source, when the source is connected to a Dedicated Mode input. In Dedicated Mode, IMAP0 and IMAP1 contain mapping information for the external interrupt pins.

PIRSR - PCI Interrupt Routing Register used to route PCI inputs to either the i960 core processor or to the primary PCI bus.

X7ISR - XINT7 Interrupt Status Register shows current pending XINT7 interrupts (e.g., Messaging Unit, APIC CSR, etc.).

5.2 Internal Interrupts

The emulation software must handle I/O APIC events that cause XINT7 interrupt to be asserted in the IPND register.

APIC Message Sent Interrupt - an event notifying emulation software that an APIC message was sent (APIC CSR Bit 6).

APIC EOI Received Interrupt - an event notifying emulation software that a level-sensitive interrupt has been serviced by the host; it signifies the EOI (not all EOIs Received will be for this I/O APIC [APIC CSR Bit 15]).

APIC Register Select Interrupt - an event notifying emulation software that the host wants to READ selected APIC registers or Redirection Table data (Inbound Interrupt Status Register [IISR] Bit 7).

APIC Window Interrupt - an event notifying emulation software that the host wants to WRITE selected APIC registers or Redirection Table data (IISR Bit 8).

The order of accessing the interrupt registers is significant. The interrupt structure is hierarchal; IPND is at the top level as it contains all external interrupts. IPND.xip3:0 are interrupts XINT3:0 which correspond to PCI interrupts S_INT[D:A]#/ XINT3:0#. IPND.xip7 is XINT7# which, when set, means that software must read X7ISR to determine the event that caused the interrupt. Applicable X7ISR events are APIC CSR and Message Unit interrupts. For APIC events, the APIC CSR is read to determine if the interrupt is due to an APIC Message Sent or EOI Received. On the other hand, if the XINT7# interrupt was due to Message Unit Interrupt, the Inbound Interrupt Status Register (IISR) is read to determine if the event was a Window Register Interrupt and Register Select Register Interrupt. The emulation software pseudo code shows the order in which interrupt registers are read to determine the event(s) causing the interrupt.

5.3 APIC Messaging Unit Registers

The 80960RP Messaging Unit contains two registers accessible to the host via the primary PCI bus. These registers are the APIC Register Select (ARSR) and the APIC Window Register (AWR). All of the Redirection Table and the following APIC registers are accessible to host software through the ARSR and AWR.

- I/O APIC ID Register
- I/O APIC Version Register
- I/O APIC Arbitration ID Register

The ARSR is used to select which I/O APIC register or Redirection Table Entry (RDTE) appears in the AWR register. A write to the ARSR causes an interrupt to be generated to the i960 core processor. The APIC emulation software is responsible for updating the contents of the AWR register. This includes updating the AWR when the Redirection Table is changed due to interrupt activity.

As described above, a register number must be written to the ARSR prior to access. With a read access, the contents of the specified register can then be read by the host in the AWR. Once a register number is placed in the ARSR, that register number remains valid for subsequent read or write

operations. So the value in the ARSR is valid for several consecutive operations on the same register.

To prevent multiple accesses to APIC registers before the I/O APIC emulation software has a chance to update the register contents, the Messaging Unit implements a hardware interlock for PCI accesses to the APIC Registers. When the interlock is set, all subsequent PCI accesses to **either** of the APIC Registers are signalled a PCI Retry until the interlock is cleared. Emulation software clears the interlock when clearing the source of the interrupt, by writing to the IISR. Note that the hardware interlock mechanism is disabled when the APIC BIU is disabled. **Also, failure of the software to clear the interlock may result in a deadlock**.

APIC-related Messaging Unit Registers are those listed in Table 3.

Table 3. Relevant i960 RP Processor's Messaging Unit Registers

Register	Description	Local Bus Address
IIMR	Inbound Interrupt Mask Register	0000 1328H
IISR	Inbound Interrupt Status Register	0000 1324H
ARSR	APIC Register Select Register	0000 1300H
AWR	APIC Window Register	0000 1308H

NOTE: The two APIC registers provide an external PCI interface for accessing the I/O APIC Registers. They are part of the Messaging Unit which occupies the first 4 Kbytes of the ATU Primary Inbound PCI Address Space.

The following descriptions are for Table 3 registers:

IIMR - the **Inbound Interrupt Mask Register** selectively masks Messaging Unit Interrupts. Each bit in this register corresponds to an interrupt in the IISR. In our example, usage bits [8:7] are set to zero to unmask the ARSRs and AWRs.

IISR - the **Inbound Interrupt Status Register** posts interrupts generated by the Messaging Unit, Doorbell Registers, and Circular Queues. I/O APIC is only concerned with Messaging Unit interrupts in the IISR. Emulation software reads this register to determine when the interrupt is due to a write to the ARSR or AWR. All IISR interrupts are routed to XINT7# interrupt of the i960 core processor.

ARSR - APIC Register Select Register selects which APIC register or RDTE data appears in the APIC Window Register. While this is implemented as a 32-bit register, I/O APIC uses only bits [7:0] to hold the index. A write to this register from the PCI bus causes an interrupt to be generated to the i960 core processor, when interrupts are enabled.

AWR - APIC Window Register contains the value of the register selected by the Register Select Register. A write to this register from the PCI bus causes an interrupt to be generated to the i960 core processor, when interrupts are enabled.

The IIMRs and IISRs are used by emulation software for handling interrupts caused by the host software accessing the RSRs and WRs. In general, when an interrupt bit is read-only, the interrupt must be cleared through another register.

5.4 I/O APIC Logical Registers Accessed via Messaging Unit

The host, residing on the Primary PCI bus, uses the 80960RP Messaging Unit to initialize the I/O APIC and to read I/O APIC status. Bits [7:0] of the ARSR are used for selecting the data to be displayed in the AWR. These eight bits allow the host to access I/O APIC registers and the Redirection Table listed in Table 4. Using the 8-bit index provides the mapping of the **logical registers**, which can be selected by the ARSR. In implementing this interface, emulation software must not allow the host to write any read-only (RO) fields. When the host attempts to write a RO field the emulation software must not comply. In addition to not writing the RO field, emulation software must correct the value written by the host to the AWR. Note that logical register counterparts may be implemented on the 80960RP as either Memory-Mapped Hardware Registers, as is the APIC ArbID Register, or as software data structures, as is the Redirection Table.

Table 4. I/O APIC Logical Registers

Register Offset Written to the I/O Register Select Register	Logical Register Name	Register details
00H	I/O APIC ID Register	See section 5.4.1 Logical I/O APIC ID Register
01H	I/O APIC Version	See section 5.4.1.1 APIC Version Register
02H	I/O APIC Arbitration ID Register	See section 5.5 APIC Bus Interface Unit (BIU) Registers
03H-0FH	Reserved	Reads to these registers should always return zero. Writes have no effect.
10H	Redirection Table Entry 0 Bits 31:0	See section 5.4.1.2 Redirection Table Entries (RDTE)
11H	Redirection Table Entry 0 Bits 63:32	See section 5.4.1.2 Redirection Table Entries (RDTE)
12H	Redirection Table Entry 1 Bits 31:0	Does the same as RDTE 0 Bits 31:0 but for entry 1
13H	Redirection Table Entry 1 Bits 63:32	Does the same as RDTE 0 Bits 63:32 but for entry 1
14H - FFH	Redirection Table Entries 2-120	These **logical registers** allow access to the rest of the **software data structure** Redirection Table entries. Reads to unimplemented entries return 0 and writes have no effect.

The **Physical** ID Register is different from the **Logical** ID Register, which is visible to the host. Therefore, emulation software must make necessary changes when exchanging data between these two registers. Logical Registers represent the Host's viewpoint of APIC registers and Redirection Table entries. The Physical Registers are those as seen by the emulation software. In some cases these registers have the same physical as logical implementation, but in some cases they are different. Since the emulation software deals with both types, it needs to understand the differences and make adjustments as required.

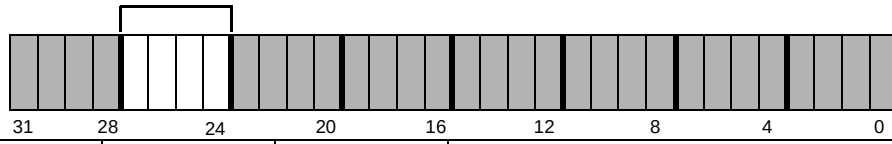
5.4.1 Logical I/O APIC ID Register

The Physical I/O APIC ID Register is implemented differently from the Logical APIC ID Register. Therefore, emulation software must make these differences transparent to the host. It does this by making the necessary changes when moving data between the two ID registers.

The Logical Register (Table 5), consists of a 4-bit read/write field in (bits 27:24), for the APIC ID. The rest of the register contains reserved bits, which always return zero when read. The four defined bits are cleared on reset.

The Physical APIC ID Register is implemented as a **Memory-Mapped Register** on the 80960RP and the APIC ID field is in (bits 3:0). All other bits are reserved and return zero when read.

Table 5. Logical APIC ID Register



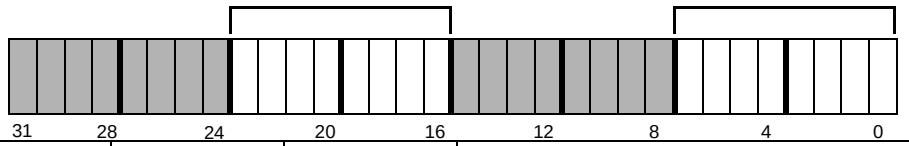
Bit	Default	R/W	Description
31:29	000_2	RO	Reserved Bits
27:24	0000_2	RO	4-bit APIC ID
24:00	0000000_H	RO	Reserved Bits

5.4.1.1 APIC Version Register

Each I/O APIC Unit contains a version register that identifies different implementations of APIC and their versions. The host reads this register to determine the maximum number of RDTEs allowed on the I/O APIC.

Table 6 shows the APIC Version Register. All fields are Read-Only (RO).

Table 6. APIC Version Register



Bit	Default	R/W	Description
31:24	00_H	RO	Reserved Bits
23:16	00_H	RO	Maximum entry in Redirection Table (#interrupts -1)
15:08	00_H	RO	Reserved Bits
07:00	17_H	RO	Version Number (17H)

The following are for Table 6 descriptions:

Version Number - The version number is 17H for I/O APIC on the 80960RP.

Max Redir Entry - This is the entry number of the highest entry in the Redirection Table. Entry numbers are zero based. This number is equal to the number of Input Interrupts minus one. The entry number range is 0 through 239.

5.4.1.2 Redirection Table Entries (RDTE)

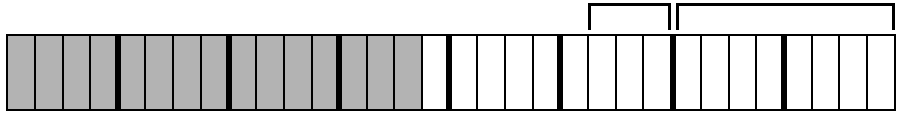
The **Physical** Redirection Table is implemented differently from the **Logical** Redirection Table known to the host. Emulation software implements Physical Redirection Table entries as single 32-bit entity. Bits [17:00] of an entry have a 1-to-1 correspondence in both implementations. The emulation software however, "pirates" reserved bits [31:24] for the 8-bit Destination field. Therefore, physical implementation basically overlays one logical register on the other.

The host sees a Redirection Table entry consisting of two 32-bit entities, as shown in Table 7 and Table 8.

The I/O APIC treats an entry as the single 32-bit entity, as shown in Section 5.6.2.

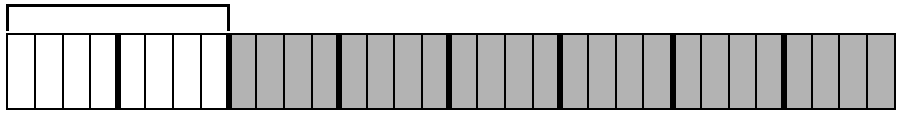
Emulation software must make the differences between the Physical and Logical Redirection Table transparent to the host processor.

Table 7. Logical Even RDTE Register



Bit	Default	R/W	Description
31:17	00000_H	RO	Reserved
16	1_2	RW	Mask Bit
15	0_2	RW	Trigger Mode
14	0_2	RW	Remote IRR Bit
13	0_2	RW	Interrupt Input Pin Polarity
12	0_2	RW	Delivery Status
11	0_2	RW	Destination Mode
10:08	000_2	RW	Delivery Mode
07:00	00_H	RW	APIC Vector

Table 8. Logical Odd RDTE Register



Bit	Default	R/W	Description
63:56	00_H	RW	8-bit Destination Field
55:32	000000_H	RO	Reserved

The following descriptions are for Table 7 and Table 8:

APIC Vector - The vector field is an 8-bit field containing the Interrupt Vector for this interrupt. Vector values range between 10 and FEH.

Delivery Mode - is a 3-bit field that specifies how the APIC listed destination field should act upon reception of this signal. Note that certain Delivery Modes only work when used in conjunction with a specific Trigger Mode. The Delivery Modes are: 000 (Fixed), 001 (Lowest Priority), 010 (SMI), 100 (NMI), 101 (INIT), and 111 (ExtINT).

Destination Mode - This field determines the interpretation of the Destination Field. The destination modes are "0" (physical) and "1" (logical).

Delivery Status - Delivery Status is a 1-bit field that contains the current status of the delivery of this interrupt. Two states are defined: "0" (Idle) and "1" (Send Pending). Send Pending indicates that the interrupt has been injected, but its delivery is temporarily held up due to APIC bus being busy or the inability of the receiving APIC unit to accept that interrupt at this time.

Interrupt Input Pin Polarity - This bit specifies the polarity of each interrupt signal connected to the input pins of the I/O APIC. A value of "0" means the signal is active high and a value of "1" means the signal is active low.

Remote IRR - This bit is used for level-triggered interrupts; its meaning is undefined for edge-triggered interrupts. For level-triggered interrupts, this bit is set when the Local APICs accept the level interrupt sent by the I/O APIC. Remote IRR bit is reset when an EOI message is received from a Local APIC.

Trigger Mode - This field indicates the type of signal on the interrupt pin that triggers an interrupt. Level "0" indicates edge sensitive, and "1" indicates level sensitive.

Mask - Use this bit to mask injection of this interrupt. Level "0" indicates injection of this interrupt is **not masked**, while "1" indicates injection of this interrupt **is masked**.

Destination - When the Destination Mode of this entry is Physical Mode, then bits [59:56] contain an APIC ID. When Logical Mode, then the destination field potentially defines a set of processors. Bits [63:56] of the Destination field specify the logical destination.

Table 9 lists Valid APIC programming combinations.

Table 9. Valid APIC Programming Combinations

Trigger Mode	Destination Mode	Delivery Mode
Edge or Level	Physical or Logical	Fixed
Edge or Level	Physical or Logical	Lowest Priority(LP)
Edge	Physical or Logical	NMI
Edge	Physical or Logical	INIT
Edge	Physical or Logical	SMI
Edge	Physical or Logical	ExtINT

NOTE: External hardware is required to implement edge-triggered interrupts.

5.5 APIC Bus Interface Unit (BIU) Registers

This section describes the programmer's interface to the I/O APIC BIU, on the 80960RP. Full details of the BIU registers are described in the *i960® RP Microprocessor User's Manual* (272736) and are not repeated here. Table 10 lists APIC BIU registers and their memory-mapped addresses.

Table 10. i960 RP Processor's APIC BIU Registers

Register Name	Description	Local Bus Address
APIC ID	APIC ID Register	0000 1780H
APIC ArbID	APIC Arbitration ID Register	0000 1784H
APIC EVR	APIC EOI Vector Register	0000 1788H
APIC IMR	APIC Interrupt Message Register	0000 178CH
APIC CSR	APIC Control and Status Register	0000 1790H

The following descriptions are for Table 10 registers:

APIC ID Register - contains the I/O APIC's unique 4-bit ID number. The ID servers as the physical name of the APIC unit. The APIC ID is read-write by emulation software and must be programmed by the host to a valid ID value before the APIC bus is used to transmit messages.

APIC ArbID Register - contains the current bus arbitration priority for the APIC BIU. This register is written by the 80960RP when the emulation software writes the APIC ID Register. This ID is used by the BIU when arbitrating for the APIC bus. Its value changes along with the activity on the APIC bus.

APIC EVR - the EOI Vector Register contains the APIC vector of received EOI message. This read-write register is normally not written by software. EOI Flow Control should be enabled when vector information in this register is important.

APIC IMR - the Interrupt Message Register provides the data to be sent as the APIC interrupt message. When the RDTE is compressed, as in the example solution, the IMR has a 1-to-1 correspondence with the fields defined in the RDTE and the bits sent in the APIC message.

APIC CSR - the BIU control and status register controls and monitors the status of the APIC BIU. Before enabling the BIU, software must read this register to verify that the PICCLK Alive is set. Without this clock the BIU cannot send or receive messages on the APIC bus. The CSR is also used for enabling Message Sent and EOI Received interrupts to the i960 core processor and clearing those interrupts once the interrupt is serviced by emulation software. Transmit status is checked by reading the ACSR status field. EOI Flow Control can be controlled through the ACSR.

5.6 Data Structures and Redirection Table

Two important software data structures used include:

- Global Data Structure
- Redirection Table

5.6.1 Global Data Structure

The following example shows an array of 32-bit entries and related data structures.

Example 1. Global Data Structure

```
typedef struct
     {
     bit32       PndIntrpt;     /* pending pci interrupts */
     bit32       CrntIntrpt;    /* pci interrupt currently servicing */
     bit32       RdtOff;        /* current interrupt * sizeof(entry) */
     bit32       MsgRetry;      /* send apic msg retry flag */
     bit32       RSIndex;       /* register select register index */
     BOOLEAN     EnblInput;     /* enable i/o apic input interrupts */
     }tISRGlobal;
```

The following are structure field descriptions for Example 1:

PndIntrpt - initially a snapshot of input PCI interrupts. With multiple interrupts in PndIntrpt, its contents change as interrupts are serviced and cleared by the emulation software.

CrntIntrpt - is the interrupt currently being serviced. It provides quick find capability when accessing the RDTE associated with current Message Send operation.

RdtOff - is an offset into the Redirection Table (CurntInt times the sizeof an RDTE [4 bytes]).

MsgRetry - is used to track any retrys required during message transmission (if retrys are implemented).

RSIndex - is a local copy of the last index value written by the host to the Register Select Register. It is saved to provide quick access on Window Register writes.

EnblInput - used to indicate whether PCI Interrupt Inputs are enabled. Also indicates where PCI inputs are routed.

5.6.2 The Redirection Table

The Redirection Table is implemented as a software data structure in 80960RP's external memory. The RDT contains an array of 32-bit entries (RDTE). Each RDTE is represented by the Example 2 data structure:

Example 2. Redirection Table Entries

```
typedef union
   {
   struct
      {
      bit32     Vector:8;          /* vector */
      bit32     DlvryMode:3;       /* delivery mode */
      bit32     DestMode:1;        /* destination mode */
      bit32     DlvryStatus:1;     /* delivery status */
      bit32     iiPinPol:1;        /* interrupt input pin polarity */
      bit32     RIRR:1;            /* remote IRR */
      bit32     TrgrMode:1;        /* trigger mode */
      bit32     Mask:1;            /* mask */
      bit32     BufFlshEnbl:1;     /* buffer flush enable */
      bit32     rsrvd:6;           /* not used */
      bit32     Dest:8;            /* destination field */
      } f;
      bit32     all;               /* access entire register */
   }t_apicRDT;
```

The field definitions shown here are identical to those specified for the logical RDTE described see Section 5.6.2, The Redirection Table (pg. 5-14). As explained previously, the RDT is an array of entries represented by the **t_apicRDT** data type. The size (RDT_SIZE) of the array is equal to the number of Input Interrupts and is implemented as a compile-time variable. In Example 3, the solution RDT_SIZE is set to four (4) because there are four Input Interrupts (S_INT[D:A]#/ XINT3:0#). The host is provided this information when the emulation software writes the maximum entry size (RTDE_SIZE -1) to the APIC Version Register. During initialization, the host reads the Version Register to get this information.

Example 3. Redirection Table Size

```
#define RDTE_SIZE 4       /* maximum number of expanded RDTEs allowed is 120 */
t_apicRDT rdt[RDT_SIZE];  /* Redirection Table */
```

5.7 Initialization

Several units within the 80960RP require initialization before system operation. These are the PCI to PCI bridge, Address Translation Unit, i960 core processor, Memory Controller, and the Secondary PCI Bus Arbiter. The order in which they are initialized is dependent on how the 80960RP is utilized in the system. The initialization process is generally controlled through either a host processor or the i960 core processor. Initialization of the 80960RP for system operation is beyond the scope of this document. Refer to the *i960 RP Microprocessor User's Manual* for initialization details.

Certain steps must be taken to install the I/O APIC ISR and to control interrupt behavior. Also, the 80960RP must be initialized before a Primary PCI bus module can use the Messaging Unit to access the APIC Register Select (ARSR) and Window Registers (AWR). Steps must also be taken to enable reception of interrupts from the Secondary PCI bus. Initialization of the 80960RP's Bridge is not covered in this application note. Refer to the *i960 RP Microprocessor User's Manual* when initializing the bridge. The following I/O APIC related initialization is required to enable I/O APIC emulation:

- Initializing core processor for Input Interrupts
- Initializing the Address Translation Unit (ATU)
- Initializing Messaging Unit for APIC interrupts
- Initializing BIU interrupts
- Initializing Redirection Table and Software Data Structures

5.7.1 Initializing Core Processor for Input Interrupts

To use the processor's interrupt handling facilities, the user software must provide the following items in memory:

- Interrupt Table
- Interrupt Service Routines
- Interrupt Stack

User software must also:

- Initialize the Initial Memory Image (IMI) data structures. The Initial Boot Record (IBR) must point to the Process Control Block (PRCB) and the PRCB must point to the Control Table, the Interrupt Table, the System Procedure Table, etc.
- Program the IMAP(0-2) to assign each external interrupt pin to a vector.
- For each ISR, place an instruction pointer to ISR in the Interrupt Table at the offset of its Interrupt Vector.
- Program the ICON Register to select (00) Dedicated Mode, (0) low-level activated Signal Detection mode, and Global Interrupts Enable. This data can be placed in the Control Table or programmed directly by software. Note that ICON also allows control of the Mask operation and Vector Cache Enable.
- Program the Interrupt Mask (IMSK) Register to selectively mask any of the Dedicated Mode interrupts.

When an interrupt is programmed to be level-sensitive detected, the pin's bit in IPND remains set as long as the input pin is asserted (low). The 80960RP attempts to clear the IPND bit on entry into the Interrupt Service Routine; however, when the active level on the pin is not removed at the same time, the bit in the IPND Register remains set

until the source of the interrupt is deactivated and the IPND bit is explicitly cleared by software. After an interrupt signal is deasserted, the ISR then clears the interrupt pending bit for that source before the return from the ISR is executed. When the pending bit is not cleared, the interrupt is reentered after the return is executed.

The example assembly language code below demonstrates how a Level Detect Interrupt is typically handled. The example assumes the **ld** from address "timer_0" deactivated the Interrupt Input.

Example 4. Return from Level-detect Interrupt

```
#Clear level-detect interrupts before return from handler
    lda IPND_MMR, g1      # Get address of IPND Memory-Mapped Register
    ld  timer_0, g0       # Get timer value and clear TMR0
    lda 0x1000, g2
wait:
    mov 0, g3
    atmod g1, g2, g3
    bbs 0xC, g3, wait
    ret                   # Return from interrupt service routine
```

The interrupts must be configured such that the PCI interrupts call the I/O APIC ISR and map Input Interrupts to RDTE entries. The following steps are performed:

- The Interrupt Table must point to the I/O APIC Interrupt Service Routine. In the Dedicated Mode, this is done by programming the interrupt map (IMAP) Registers. Doing this maps the Interrupt Input pins (S_INT[D:A]#/ XINT3:0#) to the Interrupt Service Routine.
- Each ISR is mapped to a specific entry in the Redirection Table. Mapping the Input Interrupt to an RDTE is done by the host software prior to enabling Input Interrupts.
- Write **0000 0000H** to IPND to prevent unwanted interrupts. This must be done before any mask register bits are set.
- Write IMAP1:0 with the four most-significant bits of the vector number for each external interrupt. In the example solution IMAP1:0 are programmed with the same vector number because all interrupts are handled by a single ISR.
- Write **0000 0001H** to PIRSR (interrupts routed to P_INT[D:A]# pins).
- Write **0000 008FH** to IMSK to enable S_INT[D:A]#/ XINT3:0# and XINT7# external interrupts to the i960 core processor.
- Write **0000 4000H** to ICON register (fast Sampling Mode).

The ICON and IMAP2:0 control registers are loaded from the control table in external memory when the processor is initialized or reset.

Refer to the *i960 RP Microprocessor User's Manual* for more information on core initialization.

5.7.2 Initializing the Primary Address Translation Unit (PATU)

The following sequence of events occur when initializing the ATU for use as an I/O APIC.

- APIC software programs the ATU Class Code Register (ATUCCR) with APIC Class Code
- APIC software programs the Primary Inbound ATU Limit Register (PIALR) with I/O APIC window size
- PCI configuration software reads the PCI Class Code (ATUCCR) to verify APIC function
- PCI configuration software reads I/O APIC window size from Primary ATU Base Address Register (PIABAR)
- PCI configuration software writes I/O APIC PCI base address to Primary ATU Base Address Register (PIABAR)

APIC software programs ATUCCR with a value of 080010H to set APIC class code. APIC class code fields are:

- Base Class is 08H - Base system peripherals
- SubClass is 00H
- Programming Interface value is 10H - APIC

According to the *Multiprocessor Specification*, the I/O APIC window size is fixed at 4 Kbytes, so the value the 80960RP's initialization software must program into the PIALR is FFFF F000H.

The Messaging Unit uses the first 4 Kbytes of the primary inbound translation window in the PATU. So when the window size is set by 80960RP initialization software to 4 Kbytes, only the Messaging Unit is accessible to the PCI configuration software. After initialization is complete, the PCI configuration software can access to the ARSRs and AWRs which are located in the 80960RP Messaging Unit.

The PCI configuration software uses Type 0 configuration commands on the primary PCI interface, to access the 80960RP Primary ATU (PATU) registers. Configuration software should specify a function number of one (1) and a register number of two (2) when reading PCI Class Code. PCI Class Code field occupies the upper three (3) bytes of this four (4) byte register. When accessing the PIABAR, PCI configuration software should specify a function number of one (1) and a register number of four (4).

PCI configuration software should read the ATU's Configuration Header to get the size of the primary inbound translation window so it can map the I/O APIC into the PCI address space. To get the window size, the PCI configuration software writes FFFF FFFFH to the PIABAR. It then reads the PIABAR and gets back a binary-weighted value that indicates the window size. For a 4 Kbyte window the value read back should be FFFF F000H. Next the PCI configuration software determines the starting address for the I/O APIC and writes this value to the PIABAR.

The PCI base address for the I/O APIC window is now contained in the PIABAR. The 80960RP uses the contents of the PIABAR and the PIALR to determine whether to claim a PCI address on the primary PCI bus.

Table 11. Relevant Address Translation Unit Registers

Register	Description	Local Bus Address	PCI Address
PATUCMD	Primary ATU Command Register	0000 1204H	04H
ATUCCR	ATU Class Code Register	0000 1209H	08H
PIABAR	Primary Inbound ATU Base Address Register	0000 1210H	10H
PIALR	Primary Inbound ATU Limit Register	0000 1240H	40H
ATUCR	ATU Configuration Register	0000 1288H	88H

The PATUCMD register bit "1" must be set to enable the ATU to respond to PCI memory addresses.

NOTE: The above discussion describes the minimal settings of the Table 11 registers required for I/O APIC operation on the 80960RP. Based on your implementation, it may be necessary to program other bits and/or ATU registers.

5.7.3 Initializing the Messaging Unit (MU)

The four Messaging Unit registers used by I/O APIC are IISR, IIMR, ARSR, and AWR. All four registers default to a value of zero. Therefore, the only I/O APIC Messaging Unit register that requires initialization is the IIMR.

Write **7FH** to IIMR to mask all Messaging Unit interrupts, except for the ARSR and AWR interrupts. Of course, when other Messaging Unit Registers are used in your application, their interrupts must also be unmasked.

To set the other three APIC Messaging Unit Registers to a known state:

- Write **0000 0000H** to ARSR
- Write **0000 0000H** to AWR
- Write **0000 0180H** to IISR

5.7.4 Initializing the APIC BIU

The default value is zero for the APIC ID Register, ArbID Register, EOI Vector Register, Interrupt Message Register, and the APIC Control/Status Register is zero. Therefore, the only BIU register that requires initialization is the APIC Control/Status Register.

Write 0000 F260H to the APIC BIU CSR Register to:

- Set EOI Flow Control bit to “1” (Wait) - to enable flow control.
- Write “1” to EOI Received bit (EOI message received) - to clear this bit.
- Set EOI Interrupt Enable bit to “1” (EOI message generates an interrupt).
- Set APIC Bus Interface Enable bit to “1” (Enabled) - to enable BIU.
- Set APIC Bus Interface Reset bit to “0” (Not Reset) - to not reset the BIU.
- Set Send Message bit to “0” (Do not send message) - to not send a message.
- Write “1” to Message Sent bit (Message sent) - to clear this bit.
- Set Message Sent Interrupt Enable bit to “1” (Enable interrupt after message sent)
- Write “0” to APIC Message Status field bits[4:0] - to clear status.

5.7.5 Initializing RDT & Software Data Structures

All entries in the Redirection Table must be initialized. For each entry all bits with the exception of the Mask bit must be set to "0". The Mask bit must be set to “1” to mask Input Interrupts until they are initialized by the PCI configuration software. To initialize the RDT, write **0001 0000H** to each Redirection Table entry.

The APIC Version Register must be initialized. Because it is implemented in software, software must set its default value. To initialize, write **00nn 0017H** to the APIC Version Register (nn = number of Input Interrupts minus one).

- Set the Version Number field to 17H.
- Write the maximum entry value in the 8-bit Maximum Entry field in Redirection Table field. This value differs from application to application, but its value should always equal the number of Input Interrupts minus one.

5.7.6 Initialization Checklist

- S_INT[D:A]#/ XINT3:0# are routed to Primary PCI bus
- APIC clock is present
- APIC Register Select and Window Register interlock is cleared
- Input Interrupts are masked
- APIC EOI interrupt is enabled
- EOI Flow control is enabled (when desired)
- APIC Message Unit Register Interrupts are enabled
- All RDTE are set to 0x00010000 (masked)
- APIC ID is zero
- Version Register has maximum entry number and version number fields set

5.8 APIC Interrupt Service Routine (ISR)

Example 5 shows top-level pseudo code for the I/O APIC ISR. The top-level pseudo code entails handling all events and shows the order in which interrupts are serviced. This is followed by pseudo code for handling: (1) Input Interrupts, (2) message sent, (3) EOI received, (4) register select, and (5) window register interrupts.

5.8.1 APIC ISR Top Level Pseudo Code

A snapshot of all pending interrupts is taken and saved on entering the ISR. All pending interrupts are addressed prior to exiting the ISR. PCI Input Interrupts are handled first, followed by Message Sent, EOI Received, Register Select, and the Window Register interrupt.

Example 5. APIC ISR Top Level Pseudo Code

```
Read and save unmasked IPND interrupts;
IF ipnd event is xip0-3 THEN
       Process pci input interrupts;
ENDIF
IF ipnd event is xip7 THEN
       Read x7isr;
       IF x7isr event is APIC interrupt THEN
              Read APIC BIU Register;
              IF APIC BIU event is message sent THEN
                    Process message sent interrupt;
              ENDIF
              IF APIC BIU event eoi received THEN
                     Process eoi received interrupt;
              ENDIF
       ENDIF
       IF x7isr event is Msg Unit interrupt THEN
              Read iisr;
              IF iisr event is register select THEN
                     Process register select;
              ENDIF
              ELSEIF iisr event is window register THEN
                     Process window register interrupt;
              ENDIF
       ENDIF
ENDIF
Clear IPND Register;
Exit isr;
```

5.8.2 Handling PCI Input Interrupt Pseudo Code

Example 6 shows PCI Interrupt Pseudo Code. PCI interrupts are disabled, by masking them in IMSK, to prevent further interrupts until current Input Interrupts are handled. This is important with level-triggered Input Interrupts, especially the PCI interrupts (S_INT[D:A]#/ XINT3:0#). The source of the PCI interrupts must be cleared by the host handling the APIC interrupt message for that Input Interrupt. Only after the host has handled and cleared the source of the interrupt should that Input Interrupts be unmasked.

When the BIU is idle, the emulation software programs the BIU to send the APIC message.

Example 6. PCI Interrupts Pseudo Code

```
Mask new pci interrupts;
Update interrupt pending list;
IF biu_idle THEN
     Send next pending apic interrupt on list;
ENDIF
```

5.8.3 Handling Message Sent Interrupt

Example 7 shows Message Sent Interrupt pseudo code. The Message Sent Interrupt indicates that the BIU transmitted the APIC message. The status of the transmit must be checked to determine if the transmit was successful. The software checks transmit status by reading the APIC CSR status field. When transmit status is successful, the Delivery Status field of the active Redirection Table entry is changed from **Send Pending** to **Idle** and when the Input Interrupt is level-sensitive, the Remote IRR bit for that RDTE is set. When transmit status is failed, the Delivery Status is left unchanged, and the next pending interrupt is transmitted. The next pending interrupt may be the same interrupt that just failed or it may be another pending interrupt. It may be practical to implement a retry count and discontinue trying to transmit APIC messages that have failed the last *n* number of times.

Example 7. Message Sent Interrupt Pseudo Code

```
Clear message sent interrupt;
IF transmit status success THEN
       IF level_sensitive_interrupt THEN
              Set RIRR in RDTE;
       ENDIF
       Clear delivery status;
ENDIF
IF interrupt pending THEN
       Send next pending apic interrupt;
ENDIF
```

5.8.4 Handling EOI Received Interrupt Pseudo Code

Example 8 shows EOI Received pseudo code. An End Of Interrupt (EOI) message has been received and it may or may not be for this I/O APIC. To determine if the EOI is for this APIC, the software reads the EOI Vector Register to get the vector associated with the EOI. The vector is used to scan the Redirection Table looking for all vectors that match. When there is a match, the software clears the Remote IRR field of all RDTEs with matching vectors. This indicates that the interrupt has been serviced by the host and the source of the interrupt has been cleared. It is now OK to unmask that Input Interrupt in the IMSK Register. When that Input Interrupt remains present at this time, it is treated as a new interrupt. The EOI received bit of the APIC CSR is written to clear the EOI interrupt. When interrupts are pending and the BIU is idle, the next pending Input Interrupt is transmitted. Otherwise all PCI Input Interrupts are enabled.

Example 8. EOI Received Interrupt Pseudo Code

```
Clear EOI received interrupt;
Read vector from eoi vector register;
IF eoi vector register matches rdte vector THEN
       Clear rirr bit in rdte;
       Clear interrupt bit in IPND;
       Clear pci interrupt mask(s) of interrupt just serviced;
ENDIF
IF int_pnd THEN
       IF apic is idle THEN
              Send next pending apic interrupt;
       ENDIF
ENDIF
```

5.8.5 Handling Register Select Interrupt Pseudo Code

Example 9 shows the pseudo code for handling Register Select Interrupts. A Register Select Interrupt is most likely to occur during initialization time, when the host is programming the I/O APIC with Redirection Table entries and enabling I/O APIC Input Interrupts. The index written to the Register Select Register is saved, as a global variable, to reduce the number of times the register must be read. The index specifies what APIC Register or RDTE the host wants to access. It is also used to see when a recently modified RDTE is in focus by the host. When an RDTE is modified by the emulation software and it is in focus (index is in the Register Select Register), then the emulation software must also update the contents of the Window Register. The index is also used by the emulation software when an APIC Window Interrupt occurs.

The code in Example 9 expands the RDTEs, stored as 32-bit data on the I/O APIC, to 64-bit RDTE expected by the host. The first 10 (hexadecimal) indexes are allocated to 32-bit registers, three of which are currently used. These registers are not compressed and require no expansion when being read. The remaining indexes are used to access RDTEs which are compressed on the I/O APIC. (You may prefer to not compress RDTEs and to deal with 64-bit RDTEs on the I/O APIC.) The property of an index being either odd or even is used when expanding compressed RDTEs.

A Register Select Interrupt implies a READ, by the host, of the data specified by the index. The data located at the index is written, by the emulation software, to the AWR. Then the ARSR Interrupt is cleared to re-enable the ARSR interrupt and to release the interlock on the ARSRs and AWRs. This is done by writing a “1” to bit “7” of the IISR.

Example 9. Register Select Interrupt Pseudo Code

```
Read register select register into index data structure;
IF index < 10h THEN
       Write indexed register to window register;
ELSE
       IF index is odd THEN
              Put bit 31:24 of indexed rdte in window register;
              Zero bits 23:0 in window register;
       ELSE
              Zero bits 31:24 of window register;
              Put bits 23:0 indexed rdte in window register;
       ENDIF
ENDIF
Clear register select interrupt;
```

5.8.6 Handling Window Register Interrupt Pseudo Code

Example 10 shows the pseudo code for handling Window Register interrupts. This interrupt occurs when the host wants to WRITE data to the I/O APIC in the location specified by the contents of the ARSR. Index values less than 10 (hexadecimal) are written as 32-bit entries which are handled directly with no data compression required. An index value of 10 (hexadecimal) or greater is written to an RDTE which requires data compression on the write. The property of an index being even or odd is used when performing data compression. After data is written to the indexed location on the I/O APIC, the AWR interrupt is cleared. The Window Register interrupt is cleared by writing a "1" to bit "8" of the IISR.

Example 10. Window Register Interrupt Pseudo Code

```
Read data from window register;
IF index < 10h THEN
       Write window register to index register;
ELSE
        IF index is odd THEN
              Put window register bits 31:24 in indexed rdte;
       ELSE
              Put window register bits 23:0 in indexed rdte;
       ENDIF
ENDIF
Clear window register interrupt;
```

5.9 Design Decisions

The following decisions were made in the example APIC emulation software.

Next Active Interrupt - The I/O APIC usually, continuously, scans its Input Interrupts stored in a software data structure. Interrupts are scanned sequentially (from top to bottom) in a circular fashion. When it sees an interrupt active, its delivery status bit is set (i.e., Send Pending), and software transmits that interrupt on the APIC bus. When transmission is not successful for any reason, the I/O APIC **does not reset** that interrupt's Delivery Status bit to Idle. I/O APIC goes on to the next Input Interrupt and repeats the above process. Thus, the I/O APIC goes back to the interrupt which requires a Retry only when its turn comes up in the sequence again. It may be appropriate for the I/O APIC to maintain a Retry Count and discontinue trying to send an interrupt after a predetermined number of Retrys has been reached.

Redirection Table Compression/Expansion is required when the Redirection Table in the I/O APIC is to consist of a single 32-bit data entity per RDTE, as opposed to two 32-bit data entities per RDTE accessed by host software. Handling a single 32-bit entity provides a quicker solution at run time. The time consumed compressing the Redirection Table during initialization, is prior to enabling Input Interrupts and therefore is not considered critical.

The index of the ARSR determines the accessing capability of the host when accessing APICs and the Redirection Table.

Redirection Table Access time can be improved when searching for the entry that caused the Message Sent Interrupt. A quick solution is to maintain a pointer to the current entry, thus avoiding the need to search the Redirection Table. Since only one message can be transmitted at a time only one pointer is needed.

When accessing the Redirection Table due to an EOI Received event, the received vector is stored by the BIU in the EVR. A vector match search is performed on all Redirection Table entries and all entries with a matching vector have their Remote IRR bit cleared.

When storing Redirection Table entries as a single 32-bit entity, the emulation software must direct two index values to a single Redirection Table entry. For example, index values "0" and "1" refer to RDTE0; index values "1" and "2" refer to RDTE1, and so on. This can be accomplished by using a simple lookup table method.

6.0 APIC ISR PERFORMANCE

The 80960RP employs four methods to specifically reduce interrupt latency:

- Caching Interrupt Vectors on-chip
- Caching interrupt handling procedure code
- Reserving register frames in the local register cache
- Caching the interrupt stack in the data cache

Refer to the *i960 RP Microprocessor User's Manual* for implementation details on any of the above four methods.

The example code is written in the C Programming language. If performance is unacceptable in your application, you can use your development tools to generate assembly language code from the C source code. You can then investigate tuning the assembly language code to obtain better performance.

The time-space trade-off may also be helpful. Where appropriate, substitute either MACROS or Inline functions in place of functions. Doing this makes the code size larger, but avoids the overhead associated with function calls.

Decisions on implementation details of the ISR affect performance. The ISR must check all event bits that could be set. While code executes this ISR, other required interrupts could potentially be blocked. It is possible for unmasked interrupts to be pending immediately following an exit from the ISR. It is more efficient to check all interrupts at the end of the ISR, to save the overhead of leaving and reentering the ISR soon after, but this would increase worst-case latency. The priority when ISR executes impacts performance, since priority determines when the ISR executes in relationship to other system software.

Locating the RDT in Local RAM, internal to the 80960RP, reduces access time to the RDT.

7.0 A CHECKLIST FOR WRITING APIC ISR

The following operations could be overlooked, but are important for proper operation of the I/O APIC on the 80960RP microprocessor.

- The APIC emulation software must keep track of the register or RDTE being displayed in the AWR, and whenever that data is updated, the data in the AWR must also be updated.
- While the host may, via Messaging Unit, request to write data to all register and Redirection Table fields, the emulation software must not permit the host to write RO fields. When the host attempts to write a RO field, the emulation software responds by writing only fields that are R/W to the host. The emulation software immediately updates the contents of the AWR to display the actual data written to the selected register.
- The emulation software should not route PCI interrupt pins to the i960 core processor until the Host enables the I/O APIC processing, by unmasking at least one entry in the Redirection Table.
- Even when disabled by the host from processing Input Interrupts, the I/O APIC should enable the processing of EOI and INIT interrupts.
- A write to the ARSR is synonymous with a host READ. The emulation software must write the selected data to the Window Register and clear the interrupt register to release the interlock.
- While the Host views RDT entries as 64-bit entities it is recommended that the emulation software compress them to 32-bit entities. This eases entry handling by making an entry essentially a one-to-one correspondence with the IMR fields.
- The 80960RP has register bits specified as READ/CLEAR. This means that the bit is readable. To clear it, software writes a "1" to that bit. All READ/CLEAR bits in a register can be cleared by reading the register and writing the read data value back to the same register. In registers containing just Read/Clear and read-only bits, you can just write a "1" to the bit you are clearing and not affect other bits.
- Remember the interlock that occurs when a PCI transaction writes to either the AWR or the ARSR. All subsequent PCI accesses to either of the APIC Registers are signalled a PCI retry until the interlock is cleared. To minimize Retrys, the software should prudently clear the interlock. On write transactions, the interlock can be cleared immediately after the data is copied from the AWR to internal storage. With a read transaction, software has to wait until it writes the requested data to the AWR before clearing the interlock.
- EOI received interrupts occur for **all** EOI messages seen on the APIC bus, not just for messages resulting from APIC messages sent by your I/O APIC. Also, EOI received messages are only generated for level-sensitive Input Interrupts.

- To immediately retry the last message unsuccessfully sent, simply clear the Message Sent Interrupt bit in the CSR and exit the ISR. IMR and CSR contents are still valid from previous message transmit request.
- Only one message can be sent on the APIC bus at a time. The emulation software must wait for the current message transmission to complete, usually resulting in a Message Sent Interrupt, before attempting to send another message on the APIC bus.
- EOI Flow control allows the software to throttle the receipt of EOI Received interrupts. When set to **Wait** and an EOI occurs while processing an EOI Received, then the sender is returned a Retry. When the contents of the EOI Vector Register is not important in your application, you may disable EOI Flow control.
- The **PICCLK Alive** bit in the CSR should be checked before requesting the APIC BIU to send a message over the APIC bus. When the clock is not present and software programs the BIU to send a message, the message is not sent and the software does not receive a Message Sent or EOI Received interrupts. In a reliable system it should be sufficient to check for the presence of the APIC bus clock only at initialization time.
- To ensure proper clearing of IPND when dealing with level-sensitive interrupts, clear the interrupt source before clearing IPND. You must guarantee interrupt source clear has occurred before clearing the event in the IPND register. To do this, clear the interrupt, then read it back to check that it cleared, or put as much time as possible from the time you clear the interrupt source to the time you clear IPND. **When software clears the interrupt source immediately prior to clearing its bit in IPND, the i960 core processor might clear IPND before clearing the interrupt source. This might result in behavior where the ISR is invoked with an IPND value of zero.**

8.0 SOFTWARE DEVELOPMENT TOOLS

The APIC emulation software described in this document was developed using the GNU960 development tools. The GNU960 C compiler has an **interrupt pragma** that is used to indicate that a function is used as an Interrupt Service Routine. Using this pragma informs the compiler to perform operations necessary for a function to perform as an Interrupt Service Routine. The **align pragma** is another valuable function that is used to force alignment of data required by the 80960RP. Other tool features include code optimization for size and performance considerations.

9.0 SUMMARY

The 80960RP can be programmed to provide I/O APIC functionality in the APIC architecture. To implement I/O APIC functionality, software must be written to control the 80960RP's I/O APIC Bus Interface Unit. The 80960RP hardware nicely lends itself to a totally interrupt driven software solution. Level-sensitive Input Interrupts such as PCI S_INT[D:A]#/ XINT3:0# can be routed to either the Primary PCI bus or to the i960 core processor. Interrupts routed to the core processor result in messages being sent on the APIC bus to the Local APIC device that service the interrupt and ultimately clear the source of the interrupt. In the APIC environment, Local APIC functionality is provided in an integrated form on Intel processors such as the Pentium/Pentium PRO processors. Using the APIC in this manner helps off-load interrupt related traffic from the memory bus, making the memory bus more available for processor use. In the multiprocessor environment, the APIC helps processors share the interrupt processing load with other processors.

10.0 For More Information

For more information, refer to the following documents:

- *i960 RP Microprocessor User's Manual,* Intel order number 272736-001
- *Multiprocessor Specification,* Intel order number 242016-004
- *82489DX Advanced Programmable Interrupt Controller* (data book), Intel order number 290446
- *PCI Local Bus Specification*, Product Version, Revision 2.1, June 1, 1995, can be obtained from:

  PCI Special Interest Group
  PO Box 10470
  Portland, OR 97214
  (800) 433-5177

Call (800) 548-4725 to order Intel documents. The software sources are available on Intel's home page on the World Wide Web.

APPENDIX A
Issues on Handling More than Four Input Interrupts

When the application requires more than the four (4) **standard** Input Interrupts (S_INT[D:A]#/XINT3:0), then some external hardware is required. The external hardware is envisioned to be an FPGA or ASIC device residing on the local bus and implementing an interrupt register. This hardware would receive interrupts and provide five independent interrupts on the XINT lines to the 80960RP. The Interrupt Inputs are mapped into XINT3:0, as defined by the *PCI Bridge Architecture Specification* at all times. The 80960RP then enables these through S_INT[D:A]#/XINT3:0 outputs of the 80960RP when the APIC is not in use. Those interrupts that have their enable bit in the External Interrupt Register set are all **OR**ed together and presented on XINT4. Once XINT4 interrupt is received, software must poll external hardware to find out which Input Interrupts are active. Figure A-1 illustrates this concept.

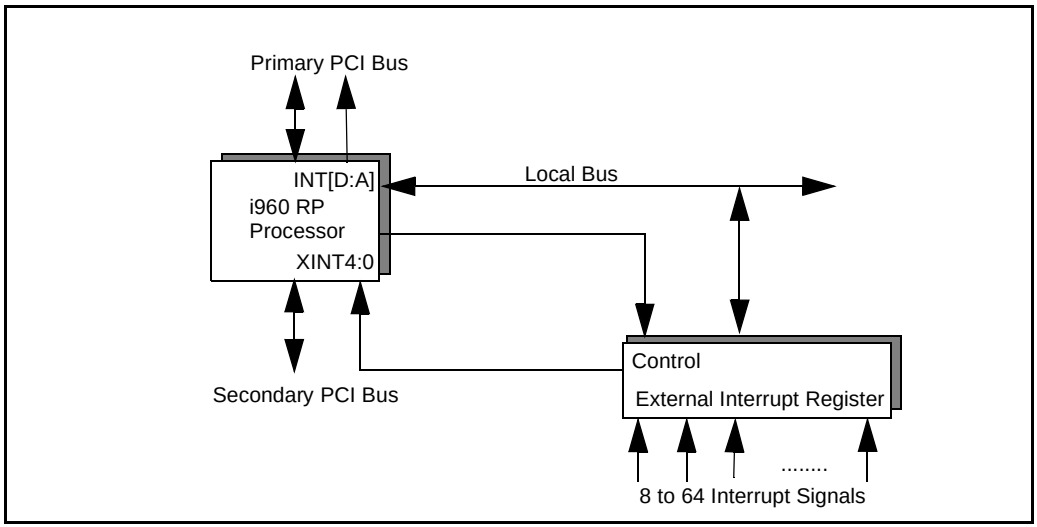


Figure A-1. External Hardware Implementation

The programming for an FPGA implementing the External Interrupt Register is very simple. It must "OR" the proper Interrupt Inputs into the four outputs, as well as **AND**ing each input with its associated enable bit and **OR**ing these registers together to be presented as XINT4. The External Interrupt Register must also support parallel read (8 bits wide is probably sufficient) of N Interrupt Inputs. An optimization, to minimize the number of reads of the register, is to map the Interrupt Inputs that are routed to a single interrupt output into a single byte, so that the proper byte can always be read by determining which XINT was asserted.

To ensure that interrupts, which are disabled by the MASK bit in their Redirection Table entry, do not cause the 80960RP to spend all of its time jumping in and out of the Interrupt Service Routine, the External Interrupt Register must also implement a read/write Enable bit for each Input Interrupt. When the mask bit is set in the Redirection Table, the emulation software also sets the corresponding enable bit in the External Interrupt Register, so that no 80960RP interrupt is generated when the Input Interrupt is active. The current state of the Input Interrupt should be visible when the register is read, but no interrupt to the 80960RP generates when the mask bit is set. This same bit should be set for all interrupts with their Redirection Table mask bit set as well.