DEC 7000/10000 AXP KN7AA CPU Technical Manual

Order Number EK—KN7AA—TM.001

The KN7AA is an Alpha AXP CPU module designed for the LSB platform. It is based on the DECchip 21064 microprocessor and is used in the DEC 7000 and DEC 10000 RISC systems. It supports up to seven MS7AA memory modules in a uniprocessor configuration and one IOP module per system. A multiprocessor system can be configured by either loading additional modules in empty backplane slots or replacing a memory module with a CPU module.

digital equipment corporation
maynard, massachusetts

First Printing, July 1993

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation.

Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software, if any, described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license. No responsibility is assumed for the use or reliability of software or equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

FCC NOTICE: The equipment described in this manual generates, uses, and may emit radio frequency energy. The equipment has been type tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such radio frequency interference when operated in a commercial environment. Operation of this equipment in a residential area may cause interference, in which case the user at his own expense may be required to take measures to correct the interference.

Contents

Chapter 5 Cache Memory

Chapter 6 LSB Bus Interface

Chapter 7 Console Overview

Chapter 8 I/O Operations

Chapter 9 CPU Module Registers

Chapter 10 Privileged Architecture Library Code

Chapter 11 OpenVMS AXP System Support

Chapter 12 DEC OSF/1 AXP System Support

Chapter 13 Initialization

Chapter 14 Error Handling

Examples

Figures

Tables

Preface

Intended Audience

This manual discusses the processor module of Digital's Alpha AXP computer systems designed for the LSB platform. It is intended for developers of system software and for Digital service personnel. It discusses the functions and operations of the KN7AA CPU module at register level. The manual assumes programming knowledge at machine language level and familiarity with the OpenVMS AXP and DEC OSF/1 AXP operating systems.

Document Structure

The material is presented in 14 chapters.

Chapter 1, **KN7AA CPU Module Overview,** presents an overall introduction to the KN7AA CPU module.

Chapter 2, **Address Space,** discusses the address space, memory and I/O, supported by the DECchip 21064.

Chapter 3, **Alpha AXP Architecture Overview,** discusses data types and instructions of the Alpha AXP architecture to prepare the user for the rest of the document.

Chapter 4, **DECchip 21064 Overview,** describes the organization of the central processor of the KN7AA CPU module. It discusses such topics as functional units, internal cache, instruction pipeline, exceptions and interrupts, and internal processor registers.

Chapter 5, **Cache Memory,** describes the elements and operations of the two-level cache hierarchy, which includes the primary cache and the backup cache.

Chapter 6, **LSB Bus Interface,** describes the functions and operations of the LEVI gate arrays that provide the CPU module interface to the LSB bus. It discusses processor-initiated and LSB bus-initiated transactions, LEVI address and data paths, and the LEVI controllers.

Chapter 7, **Console Overview,** gives a brief description of the various elements that comprise the console. It also describes the Gbus registers, which perform console control, diagnostic, and interrupt-related functions.

Chapter 8, **I/O Operations,** describes the mailbox data structure, the operation of the mailbox, interrupt handling, and I/O registers.

Chapter 9, **CPU Module Registers,** lists the LSB required and CPU-specific registers, and provides bit-level functional descriptions of each register.

Chapter 10, **Privileged Architecture Library Code,** describes the essentials of the PALcode and discusses the PALmode environment.

Chapter 11, **OpenVMS AXP System Support,** discusses memory management performed by the OpenVMS AXP operating system and gives the structure of a process within the OpenVMS AXP environment.

Chapter 12, **DEC OSF/1 AXP System Support,** discusses memory management performed by the DEC OSF/1 AXP operating system and gives the structure of a process within the DEC OSF/1 AXP environment.

Chapter 13, **Initialization,** gives an overview of the CPU module initialization, describes the methods and process of initialization, system configuration, and bootstrapping of the operating system.

Chapter 14, **Error Handling,** describes how the KN7AA module handles various types of errors. It discusses the three categories of errors from the viewpoint of error handling routines: processor-detected hard errors, module-detected and processor-recognized hard errors, and processor-corrected soft errors. Error isolation parse trees and individual fault discussions are intended to assist the error routine programmer.

Conventions Used in This Document

Unpredictable Results and Undefined Operations

Results of operations termed UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software must never use UNPREDICTABLE results.

Operations termed UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. UNDEFINED operations may halt the processor or cause it to lose information. However, they do not cause the processor to hang, that is, reach a state from which there is no transition to a normal state of instruction execution. Nonprivileged software cannot invoke UNDEFINED operations.

Register and Bit Designations

Certain conventions are followed in register descriptions and in references to bits and bit fields:

- Registers are referred to with their mnemonics, such as **LCNR register.** The full name of a register (for example, **Module Error Register**) is spelled out only at the top of the register description page, or when the register is first introduced.
- Bits and fields are enclosed in angle brackets. For example, **bit <31>; bits <31:16>.** For clarity of reference, bits are usually specified by their numbers or names enclosed in angle brackets adjacent to the register mnemonic, such as **LMERR<3:0>** or **LMERR<PMAPPE>,** which are equivalent designations.
- When the value of a bit position is given explicitly in a register diagram, the information conveyed is as follows:

Bit Value	Meaning
0	Reads as zero; ignored on writes.
1	Reads as one; ignored on writes.
X	Does not exist in hardware. The value of the bit is UNPREDICTABLE on reads and ignored on writes.

- Acronyms are used in register description tables to indicate the access type of the bit(s). The entry in the Type column of a register description table may include the initialization values of the bits. For example, entry "R/W, 0" indicates a read/write bit that is initialized to zero.

Acronym	Access Type
RC	Read to clear. The value is written by hardware and remains unchanged until read by software or PALcode.
R	Read only. May be read by software, PALcode, or hardware. Written by hardware. Software or PALcode writes are ignored.
R/W	Read/write. May be read and written by software, PALcode, or hardware.
R0	Reads as zero. Read only. Writes are ignored.
W	Write only. May be written by software or PALcode. It is read by hardware. Reads by software or PALcode return an unpredictable value.
W1C	Write 1 to clear. The value may be read by software or PALcode. Software or PALcode writes of 1 to the position cause hardware to clear the bit. Software or PALcode writes of 0 do not modify the state of the bit.
W1S	Write 1 to set. May be read and written by software, PALcode, or hardware. Set by software or PALcode with a write of 1.

MBZ. Fields in registers or data structures noted as must be zero (MBZ) must never be filled by software with a nonzero value. If the processor encounters a nonzero value in an MBZ field, an Illegal Operand exception occurs.

SBZ. Fields in registers or data structures noted as should be zero (SBZ) should be filled by software with a zero value. A nonzero value in an SBZ field produces UNPREDICTABLE results and may produce extraneous instruction-issue delays.

RAZ. Fields in registers or data structures noted as read as zero (RAZ) return a value of zero when read.

IGN. Fields (in registers or data structures) noted as Ignore (IGN) are ignored when written.

Documentation Titles

Table 1 lists the books in the DEC 7000/10000 documentation set.

Table 1 DEC 7000/10000 Documentation

Title	7000 Systems Order Number	10000 Systems Order Number
Installation Kit	EK–7000B–DK	EK–1000B–DK
Site Preparation Guide	EK–7000B–SP	EK–1000B–SP
Installation Guide	EK–700EB–IN	EK–100EB–IN
Hardware User Information Kit	EK–7001B–DK	EK–1001B–DK
Operations Manual	EK–7000B–OP	EK–1000B–OP
Basic Troubleshooting	EK–7000B–TS	EK–1000B–TS
Service Information Kit—DEC 7000	EK–7002B–DK	EK–1002B–DK
Platform Service Manual	EK–7000A–SV	EK–1000A–SV
System Service Manual	EK–7002B–SV	EK–1002A–SV
Pocket Service Guide	EK–7700A–PG	EK–1100A–PG
Advanced Troubleshooting	EK–7701A–TS	EK–1101A–TS
Reference Manuals		
Console Reference Manual	EK–70C0B–TM	
KN7AA CPU Technical Manual	EK–KN7AA–TM	
MS7AA Memory Technical Manual	EK–MS7AA–TM	
I/O System Technical Manual	EK–70I0A–TM	
Platform Technical Manual	EK–7000A–TM	
Upgrade Manuals		
KN7AA CPU Installation Card	EK–KN7AA–IN	
MS7AA Memory Installation Card	EK–MS7AA–IN	
KZMSA Adapter Installation Guide	EK–KXMSX–IN	
DWLAA Futurebus+ PIU Installation Guide	EK–DWLAA–IN	
DWLMA XMI PIU Installation Guide	EK–DWLMA–IN	
DWMBB VAXBI Installation Guide	EK–DWMBB–IN	
H7237 Battery PIU Installation Guide	EK–H7237–IN	
H7263 Power Regulator Installation Card	EK–H7263–IN	
BA654 DSSI Disk PIU Installation Guide	EK–BA654–IN	
BA655 SCSI Disk and Tape PIU Installation Guide	EK–BA655–IN	
Removable Media Installation Guide	EK–TFRRD–IN	

Table 1 DEC 7000/10000 Documentation (Continued)

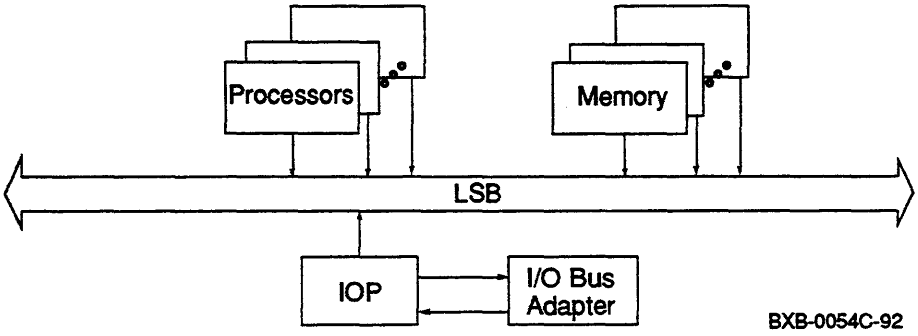
Title	7000 Systems Order Number	10000 Systems Order Number
Related Documentation		
DECchip 21064-AA, -BA Microprocessor Hardware Reference Manual	EC–N0079–72	
Alpha Architecture Reference Manual	EY–L520E–DP	

Chapter 1

KN7AA CPU Module Overview

The KN7AA CPU module is a high performance, dual-instruction issue, RISC (reduced instruction set computer) central processor unit designed around the 64-bit DECchip 21064 microprocessor and is intended for use in midrange compute servers. It operates at a peak clock rate of 200 MHz and communicates with main memory and I/O subsystems by way of the LSB bus. Figure 1-1 shows how the KN7AA CPU module fits in an Alpha AXP computer system that uses the LSB bus.

Figure 1-1 Block Diagram of a DEC 7000 or DEC 10000 System



The CPU module is an Alpha AXP architecture implementation that runs optimized versions of OpenVMS AXP and DEC OSF/1 AXP. It operates in multiple as well as single processor configurations.

All backplane slots except slot 8, which is dedicated to the IOP module, can accept CPU or memory modules. It is strongly recommended, however, that the first CPU module be placed in slot 0 for optimum performance.

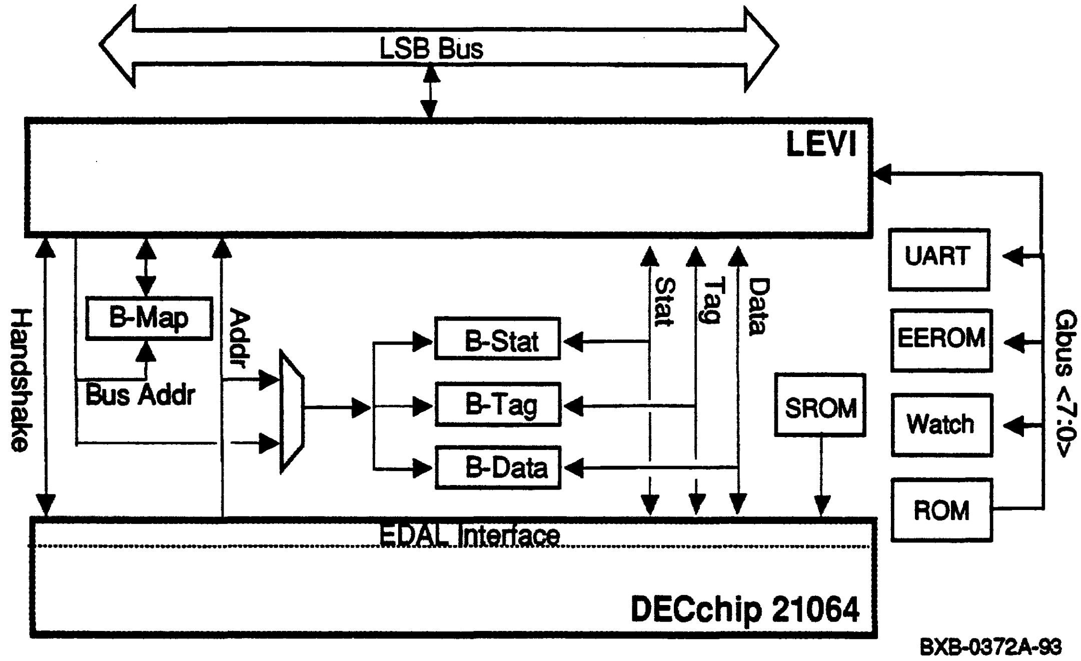
1.1 Module Hardware

The KN7AA CPU module is comprised of three major sections:

- CPU chip (DECchip 21064)
- Backup cache (B-cache)
- LSB interface (LEVI)

Figure 1-2 shows the major sections of the CPU module, which includes on-board ROMs that permit booting from supported devices and provide self-test diagnostics on power-up.

Figure 1-2 KN7AA CPU Module Block Diagram



1.1.1 DECchip 21064

The DECchip 21064 processor is a single-chip, super-scalar, super-pipelined processor with dual-instruction issue. Features include:

- Internal 8-Kbyte data cache (D-cache) and 8-Kbyte instruction cache (I-cache)
- Pipelined floating-point unit
- Demand-paged memory management unit consisting of:
  - A 12-entry I-stream translation buffer with eight entries for 8-Kbyte pages and four entries for 4 Mbyte pages
  - A 32-entry D-stream translation buffer with each entry able to map a single 8-Kbyte, 64 Kbyte, 512 Kbyte, or 4-Mbyte page (see discussions of granularity hint in Sections 11.1.4 and 12.1.4).
- Parity and ECC support
- Chip and module level test support

- Cache and memory subsystem interface (EDAL interface)

The macroinstruction pipelined design of the DECchip 21064 allows significant parallel processing. The DECchip 21064 pipelines macroinstruction decode and operand fetch with macroinstruction execution. When the macropipeline is operating smoothly, the instruction unit (Ibox), which parses instructions and fetches operands, is running several macroinstructions ahead of the execution unit (Ebox). Branch predictions allow compilers to generate optimized code flow. Outstanding writes to registers or memory locations are kept in a scoreboard to ensure that data is not read before it has been written.

The DECchip 21064 uses a set of subroutines, called Privileged Architecture Library code (PALcode), that is specific to a particular Alpha AXP operating system implementation and hardware platform. These subroutines provide operating system primitives for context switching, interrupts, exceptions, and memory management. The subroutines can be invoked by hardware or CALL_PAL instructions. PALcode is written in standard machine code with some implementation-specific extensions that provide direct access to low-level hardware functions. PALcode supports optimization for multiple operating systems, flexible memory management implementations, and multi-instruction atomic sequences.

1.1.2 Backup Cache (B-Cache)

The external backup cache (B-cache) is a 4-Mbyte superset of the primary cache (P-cache). It is a physically addressed, direct mapped, write back, mixed I-stream and D-stream cache with a block and fill size of 64 bytes. It consists of three sets of RAMs:

B-data
B-tag
B-stat

Each block of data (B-data) has a tag (B-tag) and three status bits (B-stat) associated with it. The status bits are Valid, Dirty, and Shared.

1.1.3 LSB Interface (LEVI)

The interface to the LSB bus is called LEVI, which consists of two chips: LEVI-A and LEVI-B. LEVI-A contains most LSB required registers, implements all command execution, LSB arbitration, and B-cache manipulation functions. It also contains a P-cache backmap (P-map) to allow the CPU to do invalidate filtering and to make intelligent update vs. invalidate decisions in response to LSB write traffic. LEVI-A uses an external RAM structure to implement a backmap of the B-cache to filter bus traffic from the B-cache while still maintaining cache coherence.

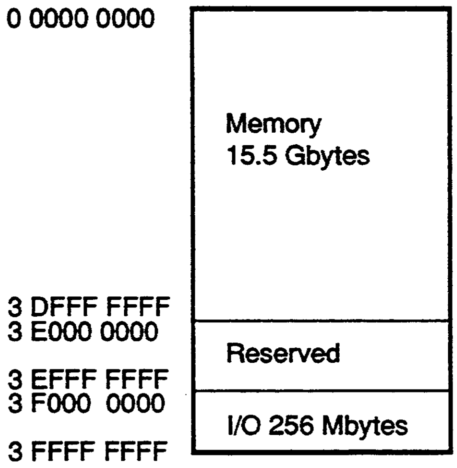
LEVI-B completes the 128-bit data path between the DECchip 21064 and the LSB bus. A 14-bit communication bus between LEVI-A and LEVI-B provides a path that allows look-aside ECC checking on incoming memory traffic.

Chapter 2

Address Space

The DECchip 21064 allows for 34 bits of physical address space. The KN7AA module defines which portion of this space is cacheable or noncacheable. Cacheable address space is commonly referred to as memory space and noncacheable space as I/O space. The KN7AA module segregates I/O space into LSB CSR space, local Gbus space, and broadcast space. Figure 2-1 shows the portion of LSB address space accessible to the DECchip 21064 processor.

Figure 2-1 KN7AA Address Space



BXB-0199B-93

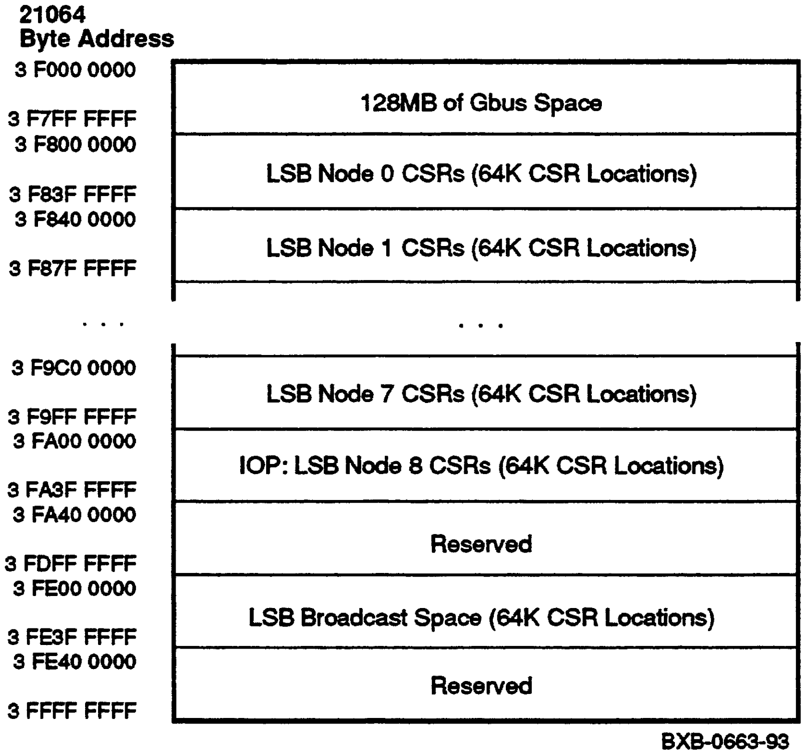
2.1 Memory Space Map

All of memory in an LSB system is accessed as 64-byte blocks. The KN7AA module maps DECchip 21064 address bits <33:5> to LSB address bits <33:5>. LSB address bits <39:34> are always zero during nonCSR LSB command cycles generated by the KN7AA module.

2.2 I/O Space Map

The KN7AA module maps the I/O space into the highest 256 Mbytes of the 34-bit DECchip 21064 physical address space. When DECchip 21064 address bits <33:28> are all ones, the KN7AA module defines these accesses to be noncached. Figure 2-2 shows the KN7AA I/O space map.

Figure 2-2 I/O (Noncacheable) Space Map



2.2.1 LSB CSR Map

All LSB-visible CSRs are defined to be 32 bits wide and aligned on 64-byte boundaries. LSB CSRs are accessed using the Read CSR and Write CSR commands. Bits D <22:1> of the address field in an LSB CSR read/write command cycle are used to specify all LSB CSRs (LSB bits D <33:23> and D <0> are always zero during CSR command cycles).

The KN7AA module maps the 128-Mbyte LSB CSR space into the next to the highest 128 Mbytes of the DECchip 21064 34-bit physical address space. When DECchip 21064 address bits <33:27> are all ones, the KN7AA module uses LSB CSR commands with DECchip 21064 physical address bits <27:6> mapped to LSB command cycle D <22:1>. D <34:23> and D <0> are driven with zeros by the KN7AA module during CSR command cycles. Table 2-1 shows the base addresses of the nodes on the LSB bus.

Table 2-1 KN7AA LSB Node Base Addresses

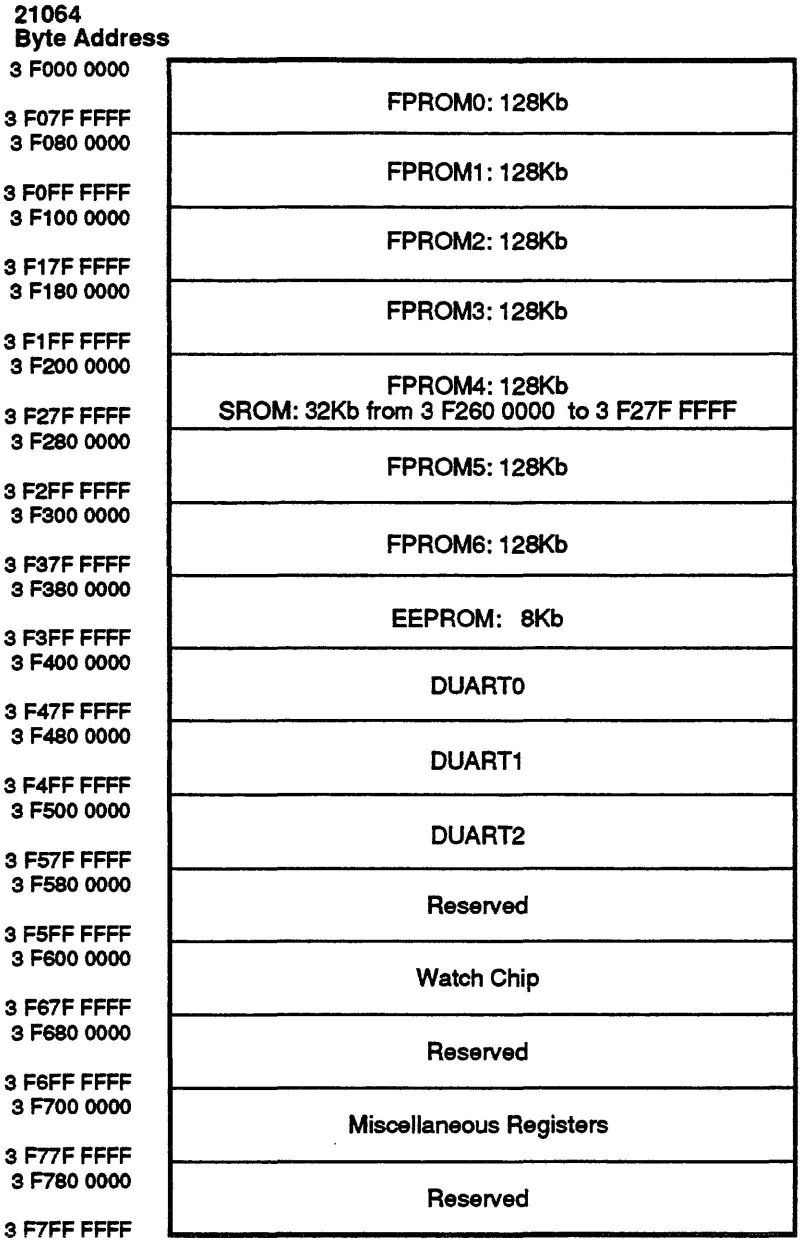
Node Number	Module	21064 Base Address
0	CPU	3 F800 0000
1	CPU/Memory	3 F840 0000
2	CPU/Memory	3 F880 0000
3	CPU/Memory	3 F8C0 0000
4	CPU/Memory	3 F900 0000
5	CPU/Memory	3 F940 0000
6	CPU/Memory	3 F980 0000
7	CPU/Memory	3 F9C0 0000
8	I/O	3 FA00 0000

2.2.2 Gbus Map

The KN7AA module allocates the first 128 Mbytes of the LSB I/O space for local (Gbus) use. This region is called private space. References to this region are serviced by resources local to the module and, therefore, are never accessed with LSB CSR or memory commands.

The KN7AA module provides access to ROM, EEROM, the console UARTs, and the watch chip through the Gbus. All Gbus addresses are located on 64-byte boundaries. Figure 2-3 shows the allocation of the Gbus space segments.

Figure 2-3 Gbus Space Map



21064 Byte Address	
3 F000 0000 3 F07F FFFF	FPROM0: 128Kb
3 F080 0000 3 F0FF FFFF	FPROM1: 128Kb
3 F100 0000 3 F17F FFFF	FPROM2: 128Kb
3 F180 0000 3 F1FF FFFF	FPROM3: 128Kb
3 F200 0000 3 F27F FFFF	FPROM4: 128Kb SROM: 32Kb from 3 F260 0000 to 3 F27F FFFF
3 F280 0000 3 F2FF FFFF	FPROM5: 128Kb
3 F300 0000 3 F37F FFFF	FPROM6: 128Kb
3 F380 0000 3 F3FF FFFF	EEPROM: 8Kb
3 F400 0000 3 F47F FFFF	DUART0
3 F480 0000 3 F4FF FFFF	DUART1
3 F500 0000 3 F57F FFFF	DUART2
3 F580 0000 3 F5FF FFFF	Reserved
3 F600 0000 3 F67F FFFF	Watch Chip
3 F680 0000 3 F6FF FFFF	Reserved
3 F700 0000 3 F77F FFFF	Miscellaneous Registers
3 F780 0000 3 F7FF FFFF	Reserved

BXB-0662-93

2.2.3 Broadcast Space

Broadcast space is used for write-only registers that are written in all nodes in a single bus transaction. This region is used to implement interrupts on the LSB. The base address of the broadcast space is 3 FE00 0000.

Chapter 3

Alpha AXP Architecture Overview

The Alpha AXP architecture is a 64-bit load/store RISC architecture designed with particular emphasis on clock speed, multiple instruction issue, and multiple processors. The architecture has the following characteristics:

- All registers are 64 bits in length, and all operations are performed between 64-bit registers.
- All instructions are 32 bits in length.
- There are 32 integer registers and 32 floating-point registers.
- Memory operations are either loads or stores.
- Memory is accessed by 64-bit virtual byte addresses in conformity with the little-endian format of the LSB bus.

This chapter presents an overview of the Alpha AXP architecture. It focuses on only two of the elements that make up the architecture: data types and instructions. The information given in this chapter is meant to provide insight to the material discussed in this document. The programmer should refer to the *Alpha Architecture Reference Manual* for a thorough discussion of the topics covered in this chapter.

3.1 Data Types

The Alpha AXP architecture provides hardware support to the following subset of data types:

Byte
Word
Longword
Quadword
D_floating (not fully supported by Alpha AXP hardware)
F_floating (32-bit)
G_floating (64-bit)
S_floating (IEEE single, 32-bit)
T_floating (IEEE double, 64-bit)

The remaining data types (octaword, H_floating, D_floating (except load/store and convert to/from G_floating), variable-length bit field, character string, trailing numeric string, leading separate numeric string, and packed decimal string) can be emulated by PALcode. Hardware-supported data types are discussed in detail in the *Alpha Architecture Reference Manual.*

3.2 Instructions

The Alpha AXP architecture supports the following types of instructions:

Memory integer load /store
Control
Integer arithmetic
Logical and shift
Byte manipulation
Floating-point
Memory format floating-point
Branch format floating-point
Floating-point operate format
Miscellaneous
VAX compatibility

These instruction types can be grouped under four instruction format classes that contain 0, 1, 2, or 3 register fields. All formats have a 6-bit opcode. The next section gives brief descriptions of the Alpha instruction classes. Refer to the *Alpha Architecture Reference Manual* for a thorough discussion of instructions supported by the Alpha AXP architecture.

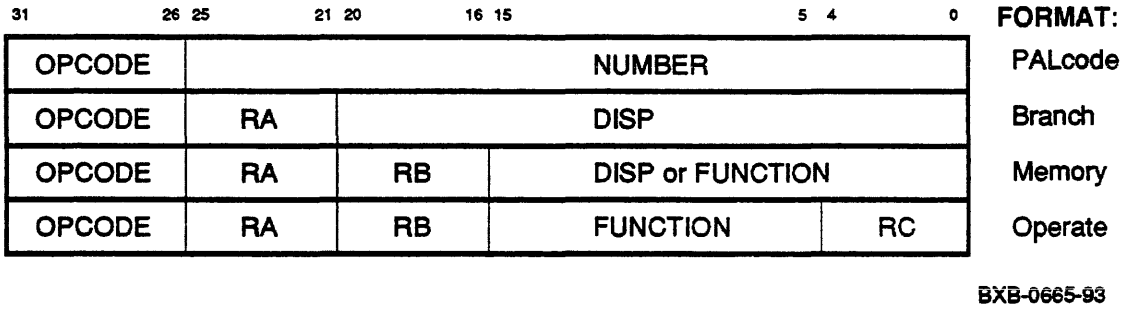
3.2.1 Instruction Format Classes

The Alpha AXP architecture supports the following four instruction formats:

- PALcode
- Branch
- Load/Store (Memory)
- Operate

Figure 3-1 shows the formats for the four classes of Alpha instructions.

Figure 3-1 Alpha AXP Instruction Formats



PALcode instructions specify, in the function code field, complex operations to be performed.

Conditional branch instructions test register Ra and specify a signed 21-bit PC-relative longword target displacement. Subroutine calls put the return address in register Ra.

Load and store instructions move longwords or quadwords between register Ra and memory, using Rb plus a signed 16-bit displacement as the memory address.

Operate instructions for floating-point and integer operations are both represented in Figure 3-1 by the operate format illustration and are as follows:

- Floating operations use Ra and Rb as source registers and write the result in register Rc. There is an 11-bit extended opcode in the function field.
- Integer operations use register Ra and register Rb or an 8-bit literal as the source operand and write the result in register Rc. Integer operate instructions can use the Rb field and part of the function field to specify an 8-bit literal. There is a 7-bit extended opcode in the function field.

3.2.2 Instruction Set Characteristics

The Alpha AXP instruction set has the following characteristics:

- All instructions are 32 bits long and have a regular format.
- There are 32 integer registers (R0 through R31), each 64 bits wide. R31 reads as zero and writes to R31 are ignored.
- There are 32 floating-point registers (F0 through F31), each 64 bits wide. F31 reads as zero and writes to F31 are ignored.
- All integer data manipulation is between integer registers, with up to two variable register source operands (one may be an 8-bit literal) and one register destination operand.
- All floating-point data manipulation is between floating-point registers, with up to two variable register source operands and one register destination operand.
- All memory reference instructions are of the load/store type that move data between registers and memory.
- There are no branch condition codes. Branch instructions test an integer or floating-point register value, which may be the result of a previous compare.
- Integer and logical instructions operate in quadwords.
- Floating-point instructions operate on G_floating, F_floating, IEEE double, and IEEE single operands. D_floating "format compatibility," in which binary files of D_floating numbers may be processed, but without the last 3 bits of fraction precision, is also provided.
- A minimal number of VAX compatibility instructions are included.

3.3 Architecturally Defined OpenVMS AXP IPRs

The Alpha AXP architecture defines OpenVMS internal processor registers (IPRs) that can be accessed by software. These registers are read and written with Move From Processor Register (MFPR) and Move To Processor Register (MTPR) instructions. Many of these registers will be referred to throughout discussions in this document. All architecturally required IPRs are discussed in the *Alpha Architecture Reference Manual.* Table 3-1 lists the Alpha AXP OpenVMS IPRs (not to be confused with the DECchip 21064 IPRs).

Table 3-1 Alpha AXP OpenVMS Internal Processor Registers

Name	Mnemonic	Access[1]
Address Space Number Register	ASN	R
AST Enable Register	ASTEN	R/W*
AST Summary Register	ASTSR	R/W*
Data Align Trap Fixup Register	DATFX	W
Floating-Point Enable Register	FEN	R/W
Interprocessor Interrupt Request Register	IPIR	W
Interrupt Priority Level Register	IPL	R/W*
Machine Check Error Summary Register	MCES	R/W
Performance Monitor Register	PERFMON	W*
Privileged Context Block Base Register	PCBB	R
Processor Base Register	PRBR	R/W
Page Table Base Register	PTBR	R
System Control Block Base Register	SCBB	R/W
Software Interrupt Request Register	SIRR	W
Software Interrupt Summary Register	SISR	R
TB Check Register	TBCHK	R
TB Invalidate All Register	TBIA	W
TB Invalidate All Process Register	TBIAP	W
TB Invalidate Single Register	TBIS	W
TB Invalidate Single Data Register	TBISD	W
TB Invalidate Single Instruction Register	TBISI	W
Kernel Stack Pointer	KSP	None
Executive Stack Pointer	ESP	R/W
Supervisor Stack Pointer	SSP	R/W
User Stack Pointer	USP	R/W
Virtual Page Table Base Register	VPTB	R/W
Who-Am-I Register	WHAMI	R

[1] **Access Types:**

R—Access by MFPR only
R/W—Access by MTPR or MFPR
R/W*—Access by MTPR or MFPR. Read and write by MTPR
W—Access by MTPR only
W*—Read and write access by MTPR
None—Not accessible by MTPR or MFPR. Accessed by PALcode routines as needed.

Chapter 4

DECchip 21064 Overview

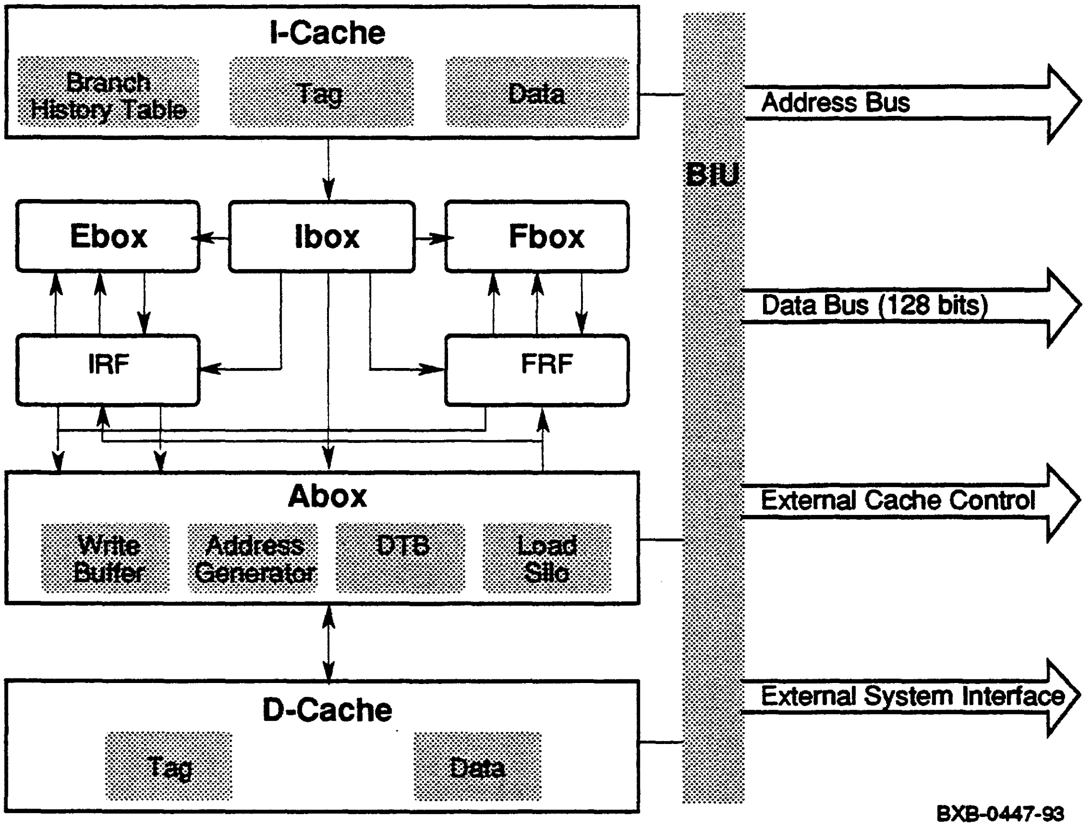
The implementation of the Alpha AXP architecture is defined by a combination of the DECchip 21064 hardware and the Privileged Architecture Library code (PALcode). This chapter presents an overview of the DECchip 21064 micro-architecture. The PALcode is discussed in Chapter 8. Sections in this chapter include:

- Functional Units
- Internal Cache
- Pipeline Organization
- Scheduling and Issuing Rules
- PALcode Instructions
- Exceptions and Interrupts
- Internal Processor Registers

For more information on some of these topics, consult the *DECchip 21064-AA, -BA Microprocessor Hardware Reference Manual* and the *Alpha Architecture Reference Manual.*

Figure 4-1 shows a block diagram of the DECchip 21064.

Figure 4-1 Block Diagram of the DECchip 21064



4.1 Functional Units

Instructions are processed in four functional units or boxes in the DECchip 21064:

- Ibox (central control unit)
- Ebox (integer execution unit)
- Abox (address generation, load/store and bus interface unit)
- Fbox (floating-point unit)

The functional units operate independently of each other. Each unit can accept at most one instruction per cycle; however, if code is correctly scheduled, the DECchip 21064 can issue two instructions to two independent units in a single cycle.

4.1.1 Ibox

The primary function of the Ibox is to issue instructions to the Ebox, Abox, and Fbox. The Ibox implements the following major elements to provide this function:

- Branch prediction logic
- Instruction translation buffers (ITB)
- Interrupt logic

- Performance counters

The Ibox decodes two instructions in parallel and checks that the required resources are available for both instructions. If resources are available, then both instructions are issued. The Ibox does *not* issue instructions out of order. If the resources are available for the second instruction, but not for the first instruction, then the Ibox issues neither. If the Ibox issues only the first of a pair of instructions, the Ibox does not advance another instruction to attempt dual issue again. Dual issue is only attempted on aligned quadword pairs.

4.1.1.1 Branch Prediction Logic

The DECchip 21064 offers a choice of branch prediction strategies selectable through the ICCSR IPR. The I-cache records the outcome of branch instructions in a single history bit provided for each instruction location in the I-cache. This information can be used as the prediction for the next execution of the branch instruction. The prediction for the first execution of a branch instruction is based on the sign of the displacement field within the branch instruction itself.

- If the sign bit is negative, the instruction prefetcher predicts the conditional branches to be taken.
- If the sign is positive, the instruction prefetcher predicts the conditional branches not to be taken.
- Alternatively, if the history table is disabled, branches can be predicted based on the sign of the displacement field at all times.

The DECchip 21064 provides a four-entry subroutine return stack that is controlled by the hint bits in the BSR, HW_REI, and jump to subroutine instructions (JMP, JSR, RET, or JSR_COROUTINE). The chip also provides a means of disabling all branch prediction hardware.

4.1.1.2 Instruction Translation Buffers

The Ibox contains two instruction translation buffers (ITB).

- An eight-entry, fully associative translation buffer that caches recently used I-stream page table entries for 8-Kbyte pages.
- A four-entry, fully associative translation buffer that supports the largest granularity hint option (512 * 8-Kbyte pages) as described further in this manual and more extensively in the *Alpha Architecture Reference Manual.*

The instruction translation buffers—hereafter referred to as the small-page ITB and large-page ITB—use a not-last-used (NLU) replacement algorithm.

In addition, the ITB includes support for an extension called the superpage, which can be enabled by the MAP bit in the ICCSR IPR. Superpage mappings provide one-to-one virtual PC<33:13> to physical PC<33:13> translation when virtual address bits <42:41> = 2. When translating through the superpage, the PTE<ASM> bit used in the I-cache is always set. Access to the superpage mapping is only allowed while executing in kernel mode.

PALcode fills and maintains the ITBs. The operating system, through PALcode, is responsible for ensuring that virtual addresses can only be mapped through a single ITB entry (in the large page, small page, or superpage) at the same time.

The Ibox presents the 43-bit virtual program counter (VPC) to the ITB each cycle while not executing in PALmode. If the PTE associated with the VPC is cached in the ITB, then the Ibox uses the PFN and protection bits for the page that contains the VPC to complete the address translation and access checks.

Each PTE entry in the ITB contains an address space match (ASM) bit. The DECchip 21064 ITB supports a single address space number (ASN) through the PTE<ASM> bit. Writes to the ITBASM IPR invalidate all entries that do not have their ASM bit set. This provides a simple method of preserving entries that map operating system regions while invalidating all others.

The ITB's tag array is updated simultaneously from the TB_TAG IPR when the ITB_PTE IPR is written. Reads of the ITB_PTE IPR require two instructions. The first instruction sends the PTE data to the ITB_PTE_TEMP IPR and the second instruction, reading from the ITB_PTE_TEMP IPR, returns the PTE entry to the register file. Reading or writing the ITB_PTE IPR increments the TB entry pointer corresponding to the large/small page selection indicated by the TB_CTL, which allows reading the entire set of ITB_PTE IPR entries.

4.1.1.3 Interrupt Logic

The DECchip 21064 supports three sources of interrupts:

- Hardware—Six level-sensitive hardware interrupts sourced by the interrupt request pins
- Software—Fifteen software interrupts sourced by an on-chip IPR (SIRR)
- Asynchronous system trap (AST)—Four AST interrupts sourced by a second internal IPR (ASTRR)

All interrupts are independently maskable by on-chip enable registers to support a software controlled mechanism for prioritization. In addition, AST interrupts are qualified by the current processor mode and the current state of SIER<2>.

By providing distinct enable bits for each independent interrupt source, a software controlled interrupt priority scheme can be implemented by PALcode or the operating system with maximum flexibility. For example, the DECchip 21064 can support a six-level interrupt priority scheme through the six hardware interrupt request pins. This is done by defining a distinct state of the Hardware Interrupt Enable IPR (HIER) for each interrupt priority level (IPL). The state of the HIER determines the current interrupt priority. The lowest interrupt priority level is produced by enabling all six interrupts (setting bits <6:1>). The next is produced by enabling five interrupts (setting bits <6:2>), and so on, to the highest interrupt priority level, which is produced by enabling only a single interrupt (setting only bit <6> and clearing bits <5:1>). When all interrupt enable bits are cleared, the processor cannot be interrupted from the HIRR IPR. Each state (<6:1>, <6:2>, <6:3>, <6:4>, <6:5>, <6>) represents an individual IPL. If these are the only states allowed in the HIER IPR, a six-level hardware interrupt priority scheme can be controlled entirely by PALcode.

The scheme is extensible to provide multiple interrupt sources at the same interrupt priority level by grouping enable bits. Groups of enable bits must be set and cleared together to support multiple interrupts of equal priority level. This method reduces the total available number of distinct levels.

Since enable bits are provided for all hardware, software, and AST interrupt requests, a priority scheme can span all sources of processor interrupts. The only exception to this rule is the following restriction on AST interrupt requests:

Four AST interrupts are provided, one for each processor operating mode—kernel, executive, supervisor, and user. AST interrupt requests are qualified such that AST requests corresponding to a given mode are blocked whenever the processor is in a higher mode regardless of the state of the AST Interrupt Enable Register. In addition, all AST interrupt requests are qualified in the DECchip 21064 with SIER<2>.

When the processor receives an interrupt request that is enabled, hardware reports or delivers an interrupt to the exception logic if the processor is not currently executing PALcode. Before vectoring to the interrupt service PALcode dispatch address, the pipeline is completely drained and all outstanding data cache fills are completed. The restart address is saved in the Exception Address IPR (EXC_ADDR) and the processor enters PALmode. The cause of the interrupt may be determined by examining the state of the interrupt request registers.

Hardware interrupt requests are level-sensitive and, therefore, may be removed before an interrupt is serviced. If they are removed before the interrupt request register is read, the register will return a zero value.

4.1.1.4 Performance Counters

The DECchip 21064 contains a performance recording feature. The implementation of this feature provides a mechanism to count various hardware events and cause an interrupt upon counter overflow. Interrupts are triggered six cycles after the event, and therefore, the exception program counter may not reflect the exact instruction causing counter overflow. Two counters are provided to allow accurate comparison of two variables under a potentially nonrepeatable experimental condition. Counter inputs include:

- Issues
- Non-Issues
- Total cycles
- Pipe dry
- Pipe freeze
- Mispredicts and cache misses
- Counts for various instruction classifications

In addition, the DECchip 21064 provides one chip pin input to each counter to measure external events at a rate determined by the selected system clock speed.

4.1.2 Ebox

The Ebox contains the 64-bit integer execution data path, which consists of the following elements:

- Adder
- Logic box
- Barrel shifter
- Byte zapper
- Bypassers
- Integer multiplier

The integer multiplier retires four bits per cycle. The Ebox also contains the 32-entry 64-bit integer register file (IRF in Figure 4-1). The register file has four read ports and two write ports that allow reading operands from and writing operands (results) to the integer execution data path.

4.1.3 Abox

The Abox contains four main elements:

- Data translation buffer
- Bus interface unit (BIU)
- Load silos
- Write buffer

4.1.3.1 Data Translation Buffer

The 32-entry, fully associative, data translation buffer (DTB) caches recently used D-stream page table entries and supports all four variants of the granularity hint option, as described in the *Alpha Architecture Reference Manual.* Superpage mapping modes, selected through ABOX_CTL <5:4>, provide virtual to physical address translation for two regions of the virtual address space. The first mode enables superpage mapping when virtual address bits <42:41> = 2. In this mode, the entire physical address space is mapped multiple times to one quadrant of the virtual address space defined by VA <42:41> = 2. The second mode maps a 30-bit region of the total physical address space defined by PA <33:30> = 0 into a single corresponding region of virtual space defined by VA<42:30> = 1FFE (hex). Superpage translation is only allowed in kernel mode. The operating system, through PALcode, should ensure that translation buffer entries, including those in the superpage regions, do not map overlapping virtual address regions at the same time.

The DECchip 21064 DTB supports a single address space number (ASN) with the PTE<ASM> bit. Each PTE entry in the DTB contains an address space match (ASM) bit. Writes to the DTBASM IPR invalidate all entries that do not have their ASM bit set. This provides a simple method of preserving entries that map operating system regions while invalidating all others.

For load and store instructions, the effective 43-bit virtual address is presented to the DTB. If the PTE of the supplied virtual address is cached in the DTB, the PFN and protection bits for the page that contains the address are used by the Abox to complete the address translation and access checks.

The DTB is filled and maintained by PALcode. Note that the DTB can be filled in kernel mode by setting ICCSR<HWE>.

The DTB's tag array is updated simultaneously from the TB_TAG IPR when the DTB_PTE is written. Reads of the DTB_PTE require two instructions. The first instruction sends the PTE data to the Data Translation Buffer Page Table Entry Temporary IPR (DTB_PTE_TEMP). The second instruction, reading from the DTB_PTE_TEMP IPR, returns the PTE entry to the register file. Reading or writing the DTB_PTE increments the TB entry pointer of the DTB, which allows reading the entire set of DTB_PTE entries.

4.1.3.2 Bus Interface Unit

The bus interface unit (BIU) controls the interface to the DECchip 21064 EDAL interface. The BIU responds to three classes of CPU-generated requests:

- D-cache fills
- I-cache fills
- Write buffer-sourced commands

The BIU resolves simultaneous internal requests using a fixed priority scheme in which D-cache fill requests are given highest priority, followed by I-cache fill requests. Write buffer requests have the lowest priority.

The BIU contains logic to directly access an external cache to service internal cache fill requests and writes from the write buffer. The BIU services reads and writes that do not hit in the external cache with help from external logic.

Internal data transfers between the CPU and the BIU are made through a 64-bit bidirectional bus. Since the internal cache fill block size is 32 bytes, cache fill operations result in four data transfers across this bus from the BIU to the appropriate cache. Also, because each write buffer entry is 32 bytes wide, write transactions may result in four data transfers from the write buffer to the BIU.

4.1.3.3 Load Silos

The Abox contains a memory reference pipeline that can accept a new load or store instruction every cycle until a D-cache fill is required. Since the D-cache lines are only allocated on load misses, the Abox can accept a new instruction every cycle until a load miss occurs. When a load miss occurs, the Ibox stops issuing all instructions that use the load port of the register file or are otherwise handled by the Abox. This includes LDx, STx, HW_MTPR, HW_MFPR, FETCH, FETCH_M, RPCC, RS, RC, and MB. It also includes all memory format branch instructions, JMP, JSR, JSR_COROUTINE, and RET. However, a JSR with a destination of R31 may be issued.

Because the result of each D-cache lookup is known late in the pipeline (stage 6) and instructions are issued in pipe stage 3, there can be two instructions in the Abox pipeline behind a load instruction that misses the D-cache. These two instructions are handled as follows:

- Loads that hit the D-cache are allowed to complete — hit under miss.
- Load misses are placed in a silo and replayed in order after the first load miss completes.
- Store instructions are presented to the D-cache at their normal time with respect to the pipeline. They are siloed and presented to the write buffer in order with respect to load misses.

To improve performance, the Ibox is allowed to restart the execution of Abox-directed instructions before the last pending D-cache fill is complete. D-cache fill transactions result in four data transfers from the BIU to the D-cache. These transfers can each be separated by one or more cycles depending on the characteristics of the external cache and memory subsystems. The BIU attempts to send the quadword of the fill block that the CPU originally requested in the first of these four transfers (it is always able to accomplish this for reads that hit in the external cache). Therefore, the pending load instruction that requested the D-cache fill can complete before the D-cache fill finishes. D-cache fill data accumulates one quadword at a time into a "pending fill" latch, rather than being written into the cache array as it is received from the BIU. When the load miss silo is empty and the requested quadword for the last outstanding load miss is received, the Ibox resumes execution of Abox-directed instructions despite the still-pending D-cache fill. When the entire cache line has been received from the BIU, it is written into the D-cache data array whenever the array is not busy with a load or a store.

4.1.3.4 Write Buffer

The Abox contains a write buffer for two purposes:

- To minimize the number of CPU stall cycles by providing a high bandwidth (but finite) resource for receiving store data. This is required since the DECchip 21064 can generate store data at the peak rate of one quadword every CPU cycle, which is greater than the rate at which the external cache subsystem can accept the data.
- To attempt to aggregate-store data into aligned 32-byte cache blocks to maximize the rate at which data may be written from the DECchip 21064 into the external cache (B-cache).

The write-merging operation of the write buffer may result in the order of off-chip writes being different from the order in which their corresponding store instructions were executed. Further, the write buffer may collapse multiple stores to the same location into a single off-chip write transaction. If strict write ordering is required, or it is desired that multiple stores to the same location result in multiple off-chip write sequences, software must insert a memory barrier instruction between the store instructions of interest.

In addition to store instructions, MB, STQ_C, STL_C, FETCH, and FETCH_M instructions are also written into the write buffer and sent off-chip. Unlike stores, however, these write buffer-directed instructions are never merged into a write buffer entry with other instructions.

The write buffer has four entries, each of which has storage for up to 32 bytes. The buffer has a "head" pointer and "tail" pointer. The buffer puts new commands into empty tail entries and takes commands out of nonempty head entries. The head pointer increments when an entry is unloaded to the BIU, and the tail pointer increments when new data is put into the tail entry. The head and tail pointers only point to the same entry when the buffer has zero or four nonempty entries.

Suppose for a moment that no writes ever merge with existing nonempty entries. In this case the ordering of writes with respect to other writes will be maintained. The write buffer never reorders writes except to merge them into nonempty entries. Once a write merges into a nonempty slot, its "programmed" order is lost with respect to both writes in the same slot and writes in other slots.

The write buffer attempts to send its head entry off-chip by requesting the BIU when one of the following conditions is met:

- The write buffer contains at least two valid entries.
- The write buffer contains one valid entry and at least 256 CPU cycles have elapsed since the execution of the last write buffer-directed instruction.
- The write buffer contains an MB instruction.
- The write buffer contains an STQ_C or STL_C instruction.
- A load miss is pending to an address currently valid in the write buffer that requires the write buffer to be flushed. The write buffer is completely flushed regardless of which entry matches the address.

4.1.4 Fbox

The Fbox is on-chip, pipelined, and capable of executing instructions in both Digital and IEEE floating-point formats. IEEE floating-point data types S_floating and T_floating are supported with all rounding modes except round to +/– infinity, which can be provided in software. F_floating and G_floating Digital floating-point data types are supported fully. Support for D_floating format is limited.

4.1.4.1 Operation

The Fbox contains:

- A 32-entry, 64-bit floating-point register file (FRF in Figure 4-1)
- A user-accessible control register, FPCR, containing:
  - — Round mode controls
  - — Exception flag information

The Fbox can accept an instruction every cycle, with the exception of floating-point divide instructions. The latency for data-dependent, non-divide instructions is six cycles.

For divide instructions, the Fbox does not compute the inexact flag. Consequently, the INE exception flag in the FPCR register is never set for IEEE floating-point divide using the inexact enable (/I) qualifier. To deliver IEEE-conforming exception behavior to the user, DECchip 21064 FPU hardware always traps on DIVS/SI and DIVT/SI instructions. The intent is for the arithmetic exception handler in either PALcode or the operating system to identify the source of the trap, compute the inexact flag, and deliver the appropriate exception to the user. The exception associated with DIV/SI and DIVT/SI is imprecise. Software must follow the rules specified by the Alpha AXP architecture associated with the software completion modifier to ensure that the trap handler can deliver correct behavior to the user.

4.1.4.2 IEEE Floating-Point Conformance

The DECchip 21064 supports the IEEE floating-point operations as defined by the Alpha AXP architecture. Support for a complete implementation of the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754-1985) is provided by a combination of hardware and software as described in the *Alpha Architecture Reference Manual.* The DECchip 21064 supports IEEE floating conformance as follows:

- When operating without the /Underflow qualifier, the DECchip 21064 replaces underflow results with exact zero even if the correct result would have been negative zero as defined in the IEEE Standard. This is an Alpha AXP architecture value-added behavior for improved performance over either hardware or software Denormal handling. When strict IEEE compliance is required, the /Underflow modifier is necessary and the software must provide the correct result (including negative zero).
- The DECchip 21064 supports Infinity operands only when used in the CMPT instruction. Other instructions using Infinity operands cause Invalid Operation (INV) arithmetic traps.
- NaN, Denormal, or Infinity (except when used in CMPT) input operands produce Invalid Operation (INV) arithmetic traps when used with arithmetic operation instructions. CPYSE/CPYSN, FCMOV instructions, and MF_FPCR/MT_FPCR are not arithmetic operations, and will pass NaN, Denormal, and Infinity values without initiating arithmetic traps. Input operand traps take precedence over arithmetic result traps.
- The DECchip 21064 does not produce a NaN, Denormal, or Infinity result.
- The DECchip 21064 supports IEEE normal and chopped rounding modes in hardware. Instructions designating plus infinity and minus infinity rounding modes cause precise exceptions to the OPCDEC PALcode entry point. This implies that the EXC_ADDR IPR will be loaded with the address of the faulting instruction and all following instructions will be aborted.
- The DIVS and DIVT with /Inexact modifier instructions report an Inexact (INE) arithmetic trap on all results of operations that do not involve NaN, Infinity, or Denormal input operands. Operations using NaN, Infinity, and Denormal input operands generate Invalid Operation (INV) arithmetic traps.
- Floating-point exceptions generated by the DECchip 21064 are recorded in two places.
  - — The FPCR register, as defined in the Alpha AXP architecture and accessible by the MT/MF_FPCR instructions, records the occurrence of all exceptions that are detected (except SWC), whether or

not the corresponding trap is enabled (through the instruction modifiers). This register can only be cleared through an explicit clear command (MT_FPCR) so that the exception information it records is a summary of all exceptions that have occurred since the last clear.

— In addition, if an exception is detected and the corresponding trap is enabled, the DECchip 21064 will record the condition in the EXC_SUM IPR and initiate an arithmetic trap. As a special case, in order to support Inexact exception behavior with the DIVS/I and DIVT/I instructions, the FPCR will not record an Inexact exception, although the DECchip 21064 will always set the INE bit in the EXC_SUM register during these instructions. This behavior allows software emulation of the division instruction with accurate reporting of potential Inexact exceptions.

4.2 Internal Cache

The DECchip 21064 includes two on-chip caches, a data cache (D-cache) and an instruction cache (I-cache). These two internal caches are referred to as P-cache in this document.

The D-cache has a size of 8 Kbytes. It is a write-through, direct-mapped, read allocate physical cache and has 32-byte blocks. System components can keep the D-cache coherent with memory by using the invalidate bus.

The I-cache is an 8-Kbyte, direct-mapped physical cache. An I-cache block, or line, contains 32 bytes of I-stream data with associated tag, as well as a six-bit ASN field, a one-bit ASM field, and an eight-bit branch history field. The I-cache does not contain hardware for maintaining coherency with memory and is unaffected by the invalidate bus.

The DECchip 21064 also contains a single-entry I-cache stream buffer that, together with its supporting logic, reduces the performance penalty due to I-cache misses incurred during in-line instruction processing. Stream buffer prefetch requests never cross physical page boundaries, but instead wrap around to the first block of the current page.

4.3 Pipeline Organization

The DECchip 21064 has a seven-stage pipeline for integer operate and memory reference instructions. Floating-point operate instructions progress through a ten-stage pipeline. The Ibox maintains state for all pipeline stages to track outstanding register writes and determine I-cache access results (hit/miss).

Figures 4-2, 4-3, and 4-4 show the integer operate, memory reference, and the floating-point operate pipelines for the Ibox, Ebox, Abox, and Fbox. The first four cycles are executed in the Ibox; the last stages are box specific. There are bypassers in all the boxes that allow the results of one instruction to be used as operands of a following instruction without having to be written to the register file.

Figure 4-2 Integer Operate Pipeline

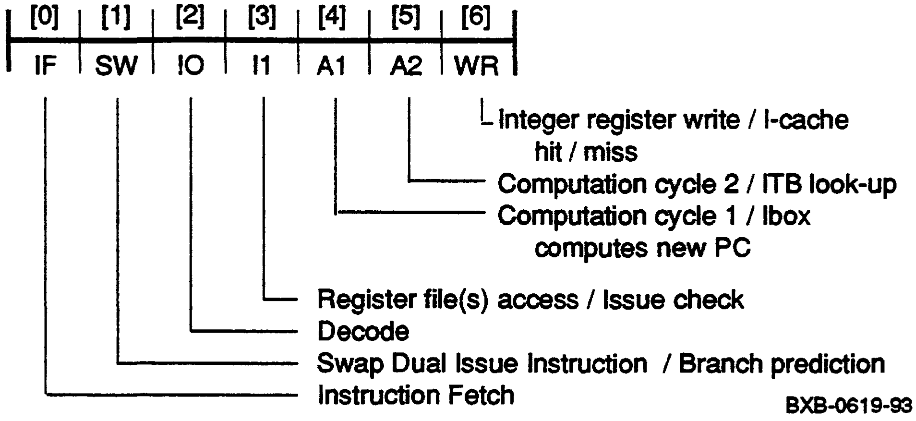

[0] [1] [2] [3] [4] [5] [6]
IF SW !O I1 A1 A2 WR
Integer register write / I-cache hit / miss
Computation cycle 2 / ITB look-up
Computation cycle 1 / Ibox computes new PC
Register file(s) access / Issue check
Decode
Swap Dual Issue Instruction / Branch prediction
Instruction Fetch
BXB-0619-93

Figure 4-3 Memory Reference Pipeline

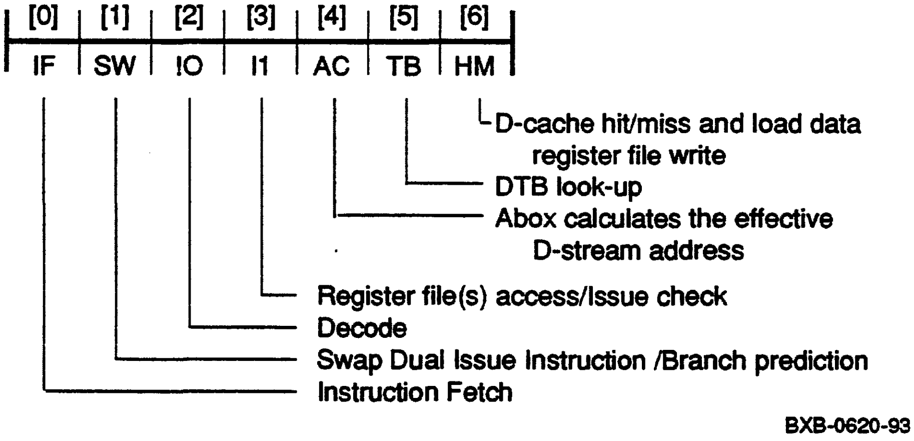
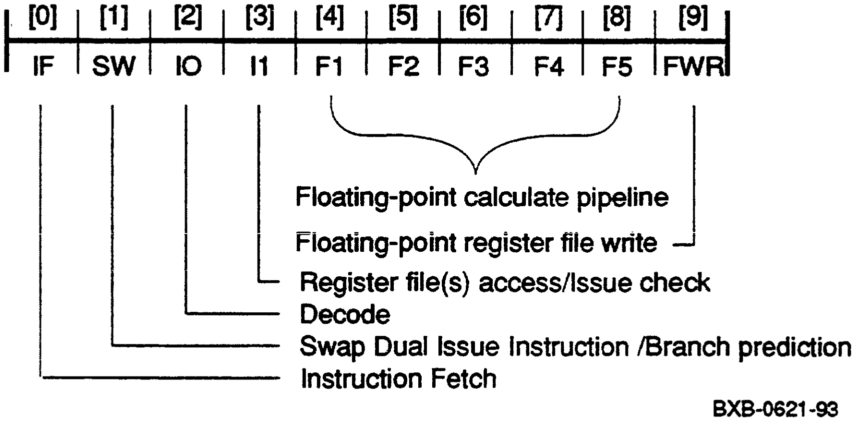

[0] [1] [2] [3] [4] [5] [6]
IF SW !O I1 AC TB HM
D-cache hit/miss and load data register file write
DTB look-up
Abox calculates the effective D-stream address
Register file(s) access/Issue check
Decode
Swap Dual Issue Instruction /Branch prediction
Instruction Fetch
BXB-0620-93

Figure 4-4 Floating-Point Operate Pipeline


[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
IF SW IO I1 F1 F2 F3 F4 F5 FWR
Floating-point calculate pipeline
Floating-point register file write
Register file(s) access/Issue check
Decode
Swap Dual Issue Instruction /Branch prediction
Instruction Fetch
BXB-0621-93

4.3.1 Static and Dynamic Stages

The DECchip 21064 integer pipeline divides instruction processing into four static and three dynamic stages of execution. The DECchip 21064 floating-point pipeline maintains the first four static stages and adds six dynamic stages of execution. The first four stages consist of:

- Instruction fetch
- Swap
- Decode
- Issue logic

These stages are static because instructions can remain valid in the same pipeline stage for multiple cycles while waiting for a resource, or stalling for other reasons.

Dynamic stages always advance state and are unaffected by any stall (also referred to as freeze) in the pipeline. A pipeline freeze may occur while zero instructions issue, or while one instruction of a pair issues and the second is held at the issue stage. A pipeline freeze implies that a valid instruction or instructions are present to be issued but cannot proceed.

Upon satisfying all issue requirements, instructions are allowed to continue through any pipeline toward completion. Instructions cannot be held in a given pipe stage after they are issued. It is up to the issue stage to ensure that all resource conflicts are resolved before an instruction is allowed to continue. The only means of stopping instructions after the issue stage is a chip-internal abort condition.

4.3.2 Aborts

Aborts can result from a number of causes. In general, they are grouped into two classes:

- Exceptions (including interrupts)
- Nonexceptions

There is one basic difference between the two classes: exceptions require that the pipeline be drained of all outstanding instructions before restarting the pipeline at a redirected address. In both exceptions and non-exceptions, the pipeline must be flushed of all instructions that were fetched after the instruction that caused the abort condition. This includes stopping one instruction of a dual-issued pair in the case of an abort condition on the first instruction of the pair.

The non-exception case, however, does not need to drain the pipeline of all outstanding instructions ahead of the aborting instruction. The pipeline can be immediately restarted at a redirected address. Examples of non-exception abort conditions are branch mispredictions, subroutine call/return mispredictions, and instruction cache misses. Data cache misses do not produce abort conditions but can cause pipeline freezes.

If an exception occurs, the processor aborts all instructions issued after the excepting instruction as described. Due to the nature of some error conditions, this can occur as late as the write cycle. Next, the address of the excepting instruction is latched in the EXC_ADDR IPR. When the pipeline is fully drained, the processor begins instruction execution at the address given by the PALcode dispatch. The pipeline is considered drained when:

- All outstanding writes to both the integer and floating-point register file have completed and arithmetic traps have been reported.
- All outstanding instructions have successfully completed memory management and access protection traps.

4.3.3 Nonissue Conditions

There are two basic reasons for nonissue conditions:

- A pipeline freeze when a valid instruction or pair of instructions are prepared to issue but cannot due to a resource conflict. This type of non-issue cycle can be minimized through code scheduling.
- Pipeline bubbles when there is no valid instruction in the pipeline to issue. Pipeline bubbles exist due to abort conditions as described in Section 4.3.2. In addition, a single pipeline bubble is produced whenever a branch-type instruction is predicted to be taken, including subroutine calls and returns. Pipeline bubbles are reduced directly by the hardware through bubble squashing, but can also be effectively minimized through careful coding practices. Bubble squashing involves the ability of the first four pipeline stages to advance whenever a bubble is detected in the pipeline stage immediately ahead of it while the pipeline is otherwise frozen.

4.4 Scheduling and Issuing Rules

The following sections cover scheduling and issuing rules.

4.4.1 Instruction Class Definition

The scheduling and dual issue rules covered in this section are only performance related. There are no functional dependencies related to scheduling or dual issuing. The scheduling and issuing rules are defined in terms of producer-consumer instruction classes. Table 4-1 lists all the instruction classes and indicates the functional box that executes the particular class of instructions.

Table 4-1 Producer-Consumer Classes

Class Name	Box	Instruction List
LD	Abox	All loads (HW_MFPR, RPCC, RS, RC, STC — producers only: FETCH — consumer only).
ST	Abox	All stores (HW_MTPR)
IBR	Ebox	Integer conditional branches
FBR	Fbox	Floating-point conditional branches
JSR	Ebox	Jump to subroutine instructions JMP, JSR, RET, or JSR_COROUTINE (BSR, BR producer only)
IADDLOG	Ebox	ADDL ADDL/V ADDQ ADDQ/V SUBL SUBL/V SUBQ SUBQ/V S4ADDL S4ADDQ S8ADDL S8ADDQ S4SUBL S4SUBQ S8SUBL S8SUBQ LDA LDAH AND BIS XOR BIC ORNOT EQV
SHIFTCM	Ebox	SLL SRL SRA EXTQL EXTLL EXTWL EXTBL EXTQH EXTLH EXTWH MSKQL MSKLL MSKWL MSKBL MSKQH MSKLH MSKWH INSQL INSLL INSWL INSBL INSQH INSLH INSWH ZAP ZAPNOT CMOVEQ CMOVNE CMOVLT CMOVLE CMOVGT CMOVGE CMOVLBS CMOVLBC
ICMP	Ebox	CMPEQ CMPLT CMPLE CMPULT CMPULE CMPBGE
IMULL	Ebox	MULL MULL/V
IMULQ	Ebox	MULQ MULQ/V UMULH
FPOP	Fbox	Floating-point operates except divide
FDIV	Fbox	Floating-point divide

4.4.2 Producer-Consumer Latency

Figure 4-5 shows in a matrix form the issue rules that the DECchip 21064 enforces regarding producer-consumer latencies. Each row and column in the matrix is a class of Alpha AXP instructions. A 1 in the producer-consumer latency matrix indicates one cycle of latency. A one cycle latency means that if instruction B uses the results of instruction A, then instruction B can be issued *one* cycle after instruction A is issued.

When determining latency for a given instruction sequence, first identify the class of each instruction. The following example lists the classes in the comment field:

```
ADDQ       R1, R2, R3 ; IADDLOG class
SRA        R3, R4, R5 ; SHIFT class
SUBQ       R5, R6, R7 ; IADDLOG class
STQ        R7, D(R10) ; ST class
```

The SRA instruction consumes the result (R3) produced by the ADDQ instruction. The latency associated with an iadd-shift producer-consumer pair as specified by the matrix is one. That means that if the ADDQ was issued in cycle *n*, the SRA could be issued in cycle *n*+1.

The SUBQ instruction consumes the result (R5) produced by the SRA instruction. The latency associated with a shift-iadd producer-consumer pair, as specified by the matrix, is two. That means that if the SRA was issued in cycle *n*, the SUBQ could be issued in cycle *n*+2. The Ibox injects one no-op cycle in the pipeline for this case.

The final case has the STQ instruction consuming the result (R7) produced by the SUBQ instruction. The latency associated with an iadd-st producer-consumer pair, when the result of the iadd is the store data, is zero. This means that the SUBQ and STQ instruction pair can be dual-issued if prefetched in the same quadword.

Figure 4-5 Producer-Consumer Latency Matrix

Producer Consumer	LD ❶	JSR	IADDLOG	SHIFTCM	ICMP	IMULL ❸	IMULQ ❸	FPOP	FIDV F/S ❹	FDIV G/T ❹
LD	3	3	2	2	2	21	23	X	X	X
❷ ST	3	3	2/0	2/0	2/0	21/20	23/22	A/4	A/32	A/61
IBR	3	3	1	2	1	21	23	X	X	X
JSR	3	3	2	2	2	21•	23•	X	X	X
IADDLOG	3	3	1	2	2	21•	23•	X	X	X
SHIFTCM	3	3	1	2	2	21•	23•	X	X	X
ICMP	3	3	1	2	2	21•	23•	X	X	X
IMUL	3	3	1	2	2	21/19	23/21	X	X	X
FBR	3	X	X	X	X	X	X	6	34	63
FPOP	3	X	X	X	X	X	X	6	34	63
FDIV	3	X	X	X	X	X	X	6	34/30	63/59

BXB-0448-93

Notes to Figure 4-5:

❶ For loads, a D-cache hit is assumed. The latency for a D-cache miss is dependent on the system configuration.

❷ For some producer classes, two latencies, X/Y, are given with ST consumer class. The X represents the latency for the base address of store; the Y represents the latency for store data. Floating-point results cannot be used as the base address for load or store operations.

❸ For IMUL followed by IMUL, two latencies are given. The first represents the latency with data dependency; in other words, the second IMUL uses the result from the first. The second is the multiply latency without data dependencies.

❹ For FDIV followed by FDIV, two latencies are given. The first represents the latency with data dependency; the second FDIV uses the result from the first. The second is division latency without data dependencies.

X in Figure 4-5 indicates an impossible state, or a state not encountered under normal circumstances. For example, a floating-point branch would not follow an integer compare.

Producer-producer latencies, also known as write-after write-conflicts, are restricted only by the register write order. For most instructions, this is dictated by issue order; however, IMUL, FDIV, and LD instructions may require more time than other instructions to complete and, therefore, must stall following instructions that write the same destination register to preserve write ordering. In general, only cases involving an intervening producer-consumer conflict are of interest. They can occur commonly in a dual-issue situation when a register is reused. In these cases, producer-consumer latencies are equal to or greater than the required producer-producer latency as determined by write ordering and therefore dictate the overall latency. An example of this case is shown in the code:

```
LDQ     R2,D(R0)  ;  R2 destination
ADDQ    R2,R3,R4  ;  wr-rd conflict stalls execution ;
                  ;  waiting for R2
LDQ     R2,D(R1)  ;  wr-wr conflict may dual-issue when
                  ;  addq issues
```

4.4.3 Instruction Issue Rules

The following conditions prevent instruction issue:

- No instruction can be issued until all of its source and destination registers are clean; in other words, all outstanding writes to the destination register are guaranteed to complete in issue order and there are no outstanding writes to the source registers or those writes can be bypassed.
- No LD, ST, FETCH, MB, RPCC, RS, RC, TRAPB, HW_MXPR, or BSR, BR, JSR (with destination other than R31) can be issued after an MB instruction until the MB has been acknowledged on the external EDAL interface.
- No IMUL instructions can be issued if the integer multiplier is busy.
- No SHIFT, IADDLOG, ICMP, or ICMOV instruction can be issued exactly three cycles before an integer multiplication completes.
- No integer or floating-point conditional branch instruction can be issued in the cycle immediately following a JSR, JMP, RET, JSR_COROUTINE, or HW_REI instruction.
- No TRAPB instruction can be issued as the second instruction of a dual-issue pair.
- No LD instructions can be issued in the two cycles immediately following an STC.
- No LD, ST, FETCH, MB, RPCC, RS, RC, TRAPB, HW_MXPR or BSR, BR, JSR (with destination other than R31) instruction can be issued when the Abox is busy due to a load miss or write buffer overflow. For more information, see Section 4.1.3.3.
- No FDIV instruction can be issued if the floating-point divider is busy.
- No floating-point operate instruction can be issued exactly five or exactly six cycles before a floating-point divide completes.

4.4.4 Dual-Issue Table

Table 4-2 can be used to determine instruction pairs that can issue in a single cycle. Instructions are dispatched using two internal data paths or buses. For more information about instructions and their opcodes and definitions, refer to the *Alpha Architecture Reference Manual.* The buses are referred to in Table 4-2 as IB0, IB1, and IBx.

Any instruction identified with IB0 in the table can be issued in the same cycle as any instruction identified with IB1. An instruction that is identified as IBx may be issued with either IB0 or IB1.

Dual-issue is attempted if the input operands are available as defined by the producer-consumer latency matrix (Figure 4-5) and the following requirements are met:

- Two instructions must be contained within an aligned quadword.
- The instructions must not both be in the group labeled as IB0.
- The instructions must not both be in the group labeled as IB1.
- No more than one of JSR, integer conditonal branch, BSR, HW_REI, BR, or floating-point branch can be issued in the same cycle.
- No more than one of load, store, HW_MTPR, HW_MFPR, MISC, TRAPB, HW_REI, BSR, BR, or JSR can be issued in the same cycle.

NOTE: Producer-consumer latencies of zero indicate that dependent operations between these two instruction classes can dual issue. For example, ADDQ R1, R2, R3, and STQ R3, D(R4).

Table 4-2 Opcode Summary (with Instruction Issue Bus)

	00	08	10	18	20	28	30	38
0/8	PAL* IB1	LDA LB0	INTA* IB0	MISC* IB1	LDF IBx	LDL IBx	BR IB1	BLBC IB1
1/9	RSVD IB1	LDAH IB0	INTL* IB0	HW_MFPR IB1	LDG IBx	LDQ IBx	FBEQ IB0	BEQ IB1
2/A	RSVD IB1	RSVD IB1	INTS* IB0	JSR IB1	LDS IBx	LDL_L IBx	FBLT IB0	BLT IB1
3/B	RSVD IB1	LDQ_U IBx	INTM* IB0	HW_LD IB1	LDT IBx	LDQ_L IBx	FBLE IB0	BLE IB1
4/C	RSVD IB1	RSVD IB1	RSVD IB1	RSVD IB1	STF IB0	STL IB1	BSR IB1	BLBS IB1
5/D	RSVD IB1	RSVD IB1	FLTV* IB1	HW_MTPR IB1	STG IB0	STQ IB1	FBNE IB0	BNE IB1
6/E	RSVD IB1	RSVD IB1	FLTI* IB1	HW_REI IB1	STS IB0	STL_C IB1	FBGE IB0	BGE IB1
7/F	RSVD IB1	STQ_U IB1	FLTL* IB1	HW_ST IB1	STT IB0	STQ_C IB1	FBGT IB0	BGT IB1

Key to Opcode Summary

FLTI*—IEEE floating-point instruction opcodes
FLTL*—Floating-point operate instruction opcodes
FLTV*—VAX floating-point instruction opcodes
INTA*—Integer arithmetic instruction opcodes
INTL*—Integer logical instruction opcodes
INTM*—Integer multiply instruction opcodes
INTS*—Integer shift instruction opcodes
JSR*—Jump instruction opcodes
MISC*—Miscellaneous instruction opcodes
PAL*—PALcode instruction (CALL_PAL) opcodes
RSVD*—Reserved for Digital

4.5 PALcode Instructions

Five opcodes are provided by the Alpha AXP architecture as implementation-specific privileged instructions. These instructions are defined independently for each Alpha AXP hardware implementation to provide PALcode software routines with access to specific hardware state and functions. All PALcode instructions are described in the *Alpha Architecture Reference Manual.*

4.5.1 Required PALcode Instructions

The PALcode instructions listed in Table 4-3 must be supported by all Alpha AXP implementations.

Table 4-3 Required PALcode Instructions

Mnemonic			
OpenVMS AXP	OSF/1 AXP	Type	Operation
HALT	Halt	Privileged	Halt processor
IMB	imb	Unprivileged	I-stream memory barrier
DRAINA	draina	Privileged	Drains aborts
SWPPAL	swppal	Privileged	Swap PALcode

4.5.2 PALcode Instructions That Require Recognition

The PALcode instructions listed in Table 4-4 must be recognized by mnemonic and opcode in all operating system implementations, but the effect of each instruction is dependent on the implementation.

Table 4-4 PALcode Instructions That Require Recognition

Mnemonic		
OpenVMS AXP	OSF/1 AXP	Name
BPT	bpt	Breakpoint trap
BUGCHK	bugchk	Bugcheck trap
GENTRAP	gentrap	Generate trap
RDUNIQUE	rdunique	Read unique value
WRUNIQUE	wrunique	Write unique value

4.5.3 Architecturally Reserved PALcode Instructions

The instructions shown in Table 4-5 are implementation dependent and are specific to the DECchip 21064. These instructions are executed in the PALcode environment. They produce OPCDEC exceptions (see Table 10-1) if executed while not in the PALcode environment. These instructions are mapped using the architecturally reserved opcodes (PAL19, PAL1B, PAL1D, PAL1E, PAL1F). They can only be used while executing chip-specific PALcode.

Table 4-5 PALmode Instructions Specific to the DECchip 21064

Mnemonic	Operation
HW_MTPR	Move data to processor register
HW_MFPR	Move data from processor register
HW_LD	Load data from memory
HW_ST	Store data in memory
HW_ REI	Return from PALmode exception

NOTE: PALcode uses the HW_LD and HW_ST instructions to access memory outside the realm of normal Alpha AXP memory management.

4.6 Exceptions and Interrupts

Both exceptions and interrupts divert execution from the normal flow of control. An exception is typically handled by the current process, while an interrupt is caused by some activity outside the current process and typically transfers control outside the process.

The DECchip 21064 processor services 32 interrupt priority levels (IPLs) divided into 16 software levels (0 to 15) and 16 hardware levels (16 to 31). User programs and most operating system software runs at IPL 0 which may be thought of as process IPL. Higher IPLs have higher priority.

The system control block (SCB) specifies the entry points for exception and interrupt service routines. The block is 8 Kbytes long, and must be page aligned. The physical address of its first byte is specified by the value in the System Control Block Base (SCBB) IPR. The operating system or console software must initialize the SCB before any interrupts are enabled.

The SCB consists of 512 entries, each 16 bytes long. The first 8 bytes of an entry, the vector, specify the operating system virtual address of the service routine associated with that entry. The second 8 bytes, the parameter, are an arbitrary quadword value to be passed to the service routine. Refer to the *Alpha Architecture Reference Manual* for details on the system control block and SCB entries.

4.6.1 Exceptions

The Alpha architecture defines three types of exceptions:

- **Faults**
  A fault is an exception condition that occurs during an instruction and leaves the registers and memory in a consistent state such that elimination of the fault condition and subsequent reexecution of the instruction will give correct results. The PC saved in the exception stack frame is the address of the faulting instruction. An REI to the PC will reexecute the faulting instruction.
- **Arithmetic Trap**
  An arithmetic trap is an exception condition that occurs at the completion of the operation that caused the exception. Since several instruc-

tion of the operation that caused the exception. Since several instructions may be in various stages of execution at any time, it is possible for multiple arithmetic traps to occur simultaneously. The PC that is saved in the exception stack frame is that of the next instruction that would have been issued if the trapping condition(s) had not occurred. A CALL_PAL REI to this PC will not reexecute the trapping instruction(s), nor will it reexecute any intervening instructions; it will simply continue execution from the point at which the trap was taken.

- **Synchronous Trap**
  A synchronous trap is an exception condition that occurs at the completion of the operation that caused the exception, and no subsequent instruction is issued before the trap occurs.

4.6.2 Interrupts

The KN7AA module uses the provided hardware interrupts as shown in Table 4-6. The handling of interrupts from the LSB, interval timer, and UARTs is accomplished with both hardware and PALcode.

Table 4-6 KN7AA Interrupts

OpenVMS				irq_h
IPL_{16}	IPL_{10}	OSF/1	Condition	Signal
1F		7	Ctrl/P detection	5
31			Node halt (LCNR<NHALT>)	5
			Machine check (LSB ERR or KN7AA-detected error)	4
		N/A[1]	Unused	
18–1E		6	LSB level 3 interrupt	3
24–30		5	Internal timer	2
17			Interprocessor interrupt	2
23			LSB level 2 interrupts	2
16		4	LSB level 1 interrupts	1
22		3	KN7AA console UARTs	0
			LSB level 0 interrupts	0
			Processor-corrected errors[2]	x
15		N/A	Unused	
21				
14		0–2	Software interrupt asserted	x
20				
10–13				
16–19				
01–0F				
01–15				

[1] Not applicable.

[2] Only DECchip 21064-BA (rev 3) chips generate internal interrupts at IPL 20. Rev 2 DECchip 21064 chips do not correct correctable errors. These chips generate hard errors at IPL 31 instead.

The Alpha AXP processor IPL is defined by the processor state managed by PALcode. The contents of the HIER, SIER, and ASTER DECchip 21064 IPRs defines the processor IPL. It is the responsibility of PALcode to manage the contents of these registers to conform to the processor IPL defined by the Alpha AXP architecture. The PALcode gets entered whenever the processor IPL is low enough (HIER, SIER, and ASTER contain appropriate values) and one of the six interrupt signals is asserted.

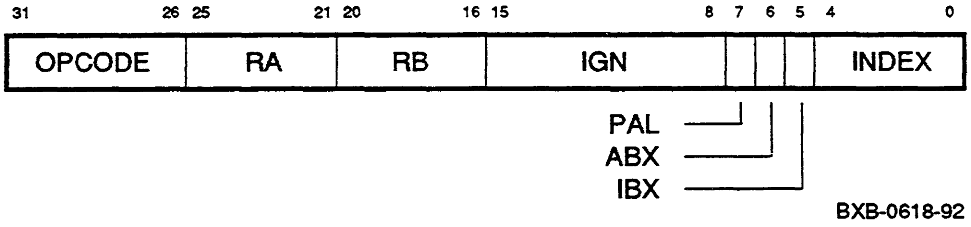
4.7 Internal Processor Registers

The DECchip 21064 contains a number of registers referred to throughout this document as IPRs (internal processor registers). The IPRs are used by the DECchip hardware and the PALcode to implement functions required by the Alpha AXP architecture. Detailed descriptions are provided for only those IPRs whose functions are defined at bit level.

4.7.1 IPR Access

PALcode initializes the ICCSR<HWE> to zero. This means that the IPRs are not visible to the user software. They are accessible only in PALmode by using the HW_MFPR or HW_MTPR instruction (see Figure 4-6). These instructions select the IPR group and reference an IPR within the group. It is possible to access IPRs in different groups with a single instruction by setting their respective bits (PAL, ABX, and IBX) in the HW_MFPR or HW_MTPR instruction, provided the IPRs from the different groups share the same index. Setting the PAL, ABX, and IBX fields to zero generates a no-op.

Figure 4-6 HW_MFPR and HW_MTPR Instruction Format



Refer to Section 10.6 for the field descriptions of the HW_MFPR and HW_MTPR instructions.

NOTE: There are two registers per processor that are associated with the LDQ_L / LDL_L and STQ_C / STL_C instructions: the lock_flag single-bit register and the locked_physical_ address register. The use of these registers is described in the Alpha Architecture Reference Manual. These registers are required by the Alpha AXP architecture but are not implemented by the DECchip 21064. They must be implemented in the application.

Table 4-7 lists the DECchip 21064 IPRs. It also indicates the access type and the index of each IPR.

Table 4-7 DECchip 21064 Internal Processor Registers

Name	Mnemonic	Access	Index
Ibox			
Translation Buffer Tag Register[1]	TB_TAG	W	0
Instruction Translation Buffer PTE Register[1]	ITB_PTE	R/W	1
Instruction Cache Control/Status Register	ICCSR	R/W	2
Instruction Translation Buffer PTE_TEMP Register[1]	ITB_PTE_TEMP	R	3
Exception Address Register	EXC_ADDR	R/W	4
Serial Line Receive Register	SL_RCV	R	5
Instruction Translation Buffer ZAP Register[1]	ITBZAP	W	6
Instruction Translation Buffer ASM Register[1]	ITBASM	W	7
Instruction Translation Buffer IS Register[1]	ITBIS	W	8
Processor Status Register	PS	R/W	9
Exception Summary Register	EXC_SUM	R/W	10
PALcode Base Address Register	PAL_BASE	R/W	11
Hardware Interrupt Request Register	HIRR	R	12
Software Interrupt Request Register	SIRR	R/W	13
Asynchronous Trap Request Register	ASTRR	R/W	14
Hardware Interrupt Enable Register	HIER	R/W	16
Software Interrupt Enable Register	SIER	R/W	17
AST Interrupt Enable Register	ASTER	R/W	18
Serial Line Interrupt Clear Register	SL_CLR	W	19
Serial Line Transmit Register	SL_XMIT	W	22

[1] Used in PALmode only.

Table 4-7 DECchip 21064 Internal Processor Registers (Continued)

Name	Mnemonic	Access	Index
Abox			
Translation Buffer Control Register	TB_CTL	W	0
Data Translation Buffer PTE Register	DTB_PTE	R/W	2
Data Translation Buffer PTE_TEMP Register	DTB_PTE_TEMP	R	3
Memory Management CSR Register	MMCSR	R	4
Virtual Address Register	VA	R	5
Data Translation Buffer ZAP Register	DTBZAP	W	6
Data Translation Buffer ASM Register	DTASM	W	7
Data Translation Buffer IS Register	DTBIS	W	8
BIU Address Register	BIU_ADDR	R	9
BIU Status Register	BIU_STAT	R	10
D-Cache Status Register	DC_STAT	R	12
Fill Address Register	FILL_ADDR	R	13
Abox Control Register	ABOX_CTL	W	14
Alternate Processor Mode Register	ALT_MODE	W	15
Cycle Counter Register	CC	W	16
Cycle Counter Control Register	CC_CTL	W	17
BIU Control Register	BIU_CTL	W	18
Fill Syndrome Register	FILL_SYND	R	19
B-Cache Tag Register	BC_TAG	R	20
Flush IC Register	FLUSH_IC	W	21
Flush IC_ASM Register	FLUSH_IC_ASM	W	23
PAL			
PALcode Temporary Registers	PAL_TEMP	R/W	31–00

[1] Used in PALmode only.

Table 4-8 shows the reset states of the DECchip 21064 IPRs and indicates the registers that need to be initialized by power-up PALcode.

Table 4-8 DECchip 21064 IPR Reset State

IPR	Reset State	Comment[1]
TB_TAG	Undefined	
ITB_PTE	Undefined	
ICCSR	Cleared except ASN, PC0, PC1	Floating-point disabled, single-issue mode, pipe mode enabled, JSR predictions disabled, branch predictions disabled, branch history table disabled, performance counters reset to zero, Perf Cnt0: Total Issues/2, Perf Cnt1: D-cache Misses, superpage disabled
ITB_PTE_TEMP	Undefined	
EXC_ADDR	Undefined	
SL_RCV	Undefined	
ITBZAP	Not applicable	PALcode must do an ITBZAP on reset before writing the ITB (must do HW_MTPR to ITBZAP IR).
ITBASM	Not applicable	
ITBIS	Not applicable	
PS	Undefined	PALcode must set processor status.
EXC_SUM	Undefined	PALcode must clear the Exception Summary IPR and the exception write mask by doing 64 reads.
PAL_BASE	Clear	Cleared on reset.
HIRR	Not applicable	
SIRR	Undefined	PALcode must initialize.
ASTRR	Undefined	PALcode must initialize.
HIER	Undefined	PALcode must initialize.
SIER	Undefined	PALcode must initialize.
ASTER	Undefined	PALcode must initialize.
SL_CLR	Undefined	PALcode must initialize.
SL_XMIT	Undefined	PALcode must initialize. Appears on external pin.
TB_CTL	Undefined	PALcode must select between SP/LP DTB prior to any TB fill.
DTB_PTE	Undefined	
DTB_PTE_TEMP	Undefined	

[1] The B-cache parameters BC RAM read speed, BC RAM write speed, BC write enable control, and BC size are all undetermined on reset. These parameters must be initialized before enabling the B-cache.

Table 4-8 DECchip 21064 IPR Reset State (Continued)

IPR	Reset State	Comment
MMCSR	Undefined	Unlocked on reset.
VA	Undefined	Unlocked on reset.
DTBZAP	Not applicable	PALcode must do an ITBZAP on reset. See ITBZAP.
DTASM	Not applicable	
DTBIS	Not applicable	
BIU_ADDR	Not applicable	Potentially locked.
BIU_STAT	Undefined	Potentially locked.
DC_STAT	Undefined	
FILL_ADDR	Undefined	Potentially locked.
ABOX_CTL	Cleared	<11:0> <- ^x0100 Write buffer enabled, machine checks disabled, correctable read interrupts disabled, I-cache stream buffer disabled, superpages 1 and 2 disabled, endian mode disabled, D-cache disabled, forced hit mode off.
ALT_MODE	Undefined	
CC	Undefined	Cycle counter is disabled on reset.
CC_CTL	Undefined	
BIU_CTL	Cleared	B-cache disabled, parity mode undefined, chip enable asserts during RAM write cycles, B-cache forced-hit mode disabled. BC_PA_DIS field cleared. BAD_TCP cleared. BAD_DP cleared.
FILL_SYND	Undefined	Potentially locked.
BC_TAG	Undefined	Potentially locked.
PAL_TEMP	Undefined	

[1] The B-cache parameters BC RAM read speed, BC RAM write speed, BC write enable control, and BC size are all undetermined on reset. These parameters must be initialized before enabling the B-cache.

4.7.2 IPR Descriptions

This section provides detailed descriptions of the DECchip 21064 IPRs. The list of the DECchip 21064 IPRs includes IPRs that do not carry functional fields and some pseudoregisters. These IPRs are the following:

- **Virtual Address Register (VA)**
  When D-stream faults or DTB misses occur, the effective virtual address associated with the fault or miss is latched in the read-only VA IPR. The VA and MMCSR registers are locked against further updates until the software reads the VA IPR. The VA IPR is unlocked after reset. PALcode must explicitly unlock this register whenever its entry point is higher in priority than a DTB miss.

- **Instruction Translation Buffer ZAP Register (ITBZAP)**
  A write to this IPR invalidates all 12 instruction translation buffer (ITB) entries. It also resets both the NLU pointers to their initial state. The ITBZAP IPR is only written to in PALmode.
- **Instruction Translation Buffer ASM Register (ITBASM)**
  A write to this IPR invalidates all ITB entries in which the <ASM> bit is equal to zero. The ITBASM IPR is only written to in PALmode.
- **Instruction Translation Buffer IS Register (ITBIS)**
  A write to the ITBIS IPR invalidates all 12 ITB entries. It also resets both the NLU pointers to their initial state. The ITBIS IPR is only written to in PALmode.
- **Data Translation Buffer ZAP Register (DTBZAP)**
  The DTBZAP is a pseudoregister. Any write to this register invalidates all 32 DTB entries. It also resets the NLU pointer to its initial state.
- **Data Translation Buffer ASM Register (DTBASM)**
  The DTBASM is a pseudoregister. Any write to this register invalidates all 32 DTB entries in which the ASM bit is zero.
- **Data Translation Buffer Invalidate Single Register (DTBIS)**
  Any write to this pseudoregister will invalidate the DTB entry, which maps the virtual address held in the integer register. The integer register is identified by the Rb field of the HW_MTPR instruction, used to perform the write.
- **Flush Instruction Cache Register (FLUSH_IC)**
  Any write to this pseudoregister flushes the entire instruction cache.
- **Flush Instruction Cache ASM Register (FLUSH_IC_ASM)**
  Any write to this pseudoregister invalidates all I-cache blocks in which the ASM bit is clear.
- **PAL_TEMP IPRs**
  These 32 registers provide temporary storage for PALcode. They are accessed by way of the HW_MTPR and HW_MFPR instructions.

The descriptions of the rest of the DECchip 21064 IPRs follow.

TB_TAG—Translation Buffer Tag Register

Index Ibox 0
Access W

The TB_TAG IPR holds the tag for the next translation buffer update operation in the instruction translation buffer (ITB) or the data translation buffer (DTB). The tag is written to a temporary register and not transferred to the ITB or DTB until the Instruction Translation Buffer Page Table Entry (ITB_PTE) or the Data Translation Buffer Page Table Entry (DTB_PTE) IPR is written. The entry to be written is chosen at the time of the ITB_PTE or DTB_PTE write operation by a not-last-used (NLU) algorithm, implemented in hardware.

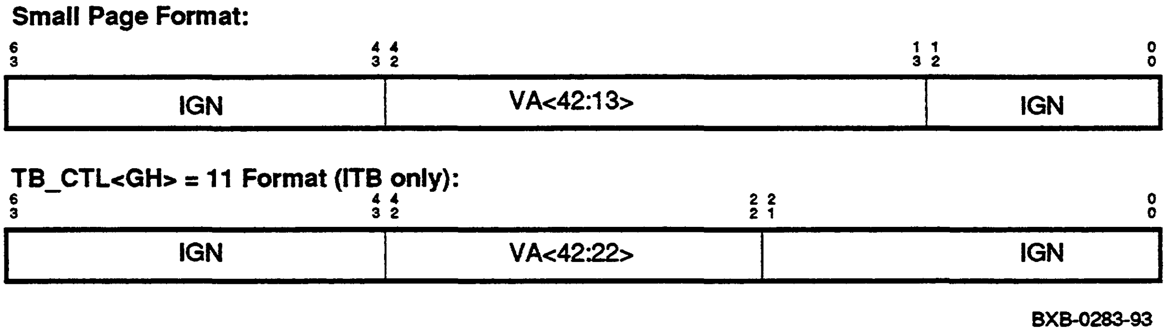


Table 4-9 TB_TAG IPR Bit Definitions

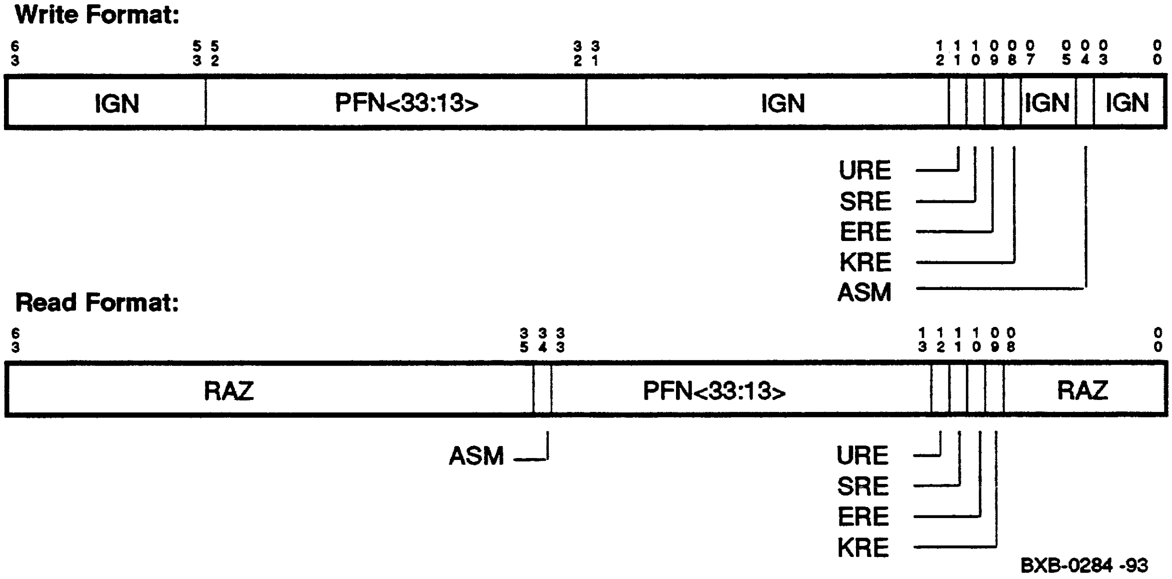
Name	Bit(s)	Type	Function
VA	<42:13>	W	**Virtual Address.** Bits extracted from the virtual address to form the tags for the small pages (8 Kbytes) of the ITB.
VA	<42:22>	W	**Virtual Address.** Bits extracted from the virtual address to form the tags for the large pages (4 Mbytes) of the ITB.

ITB_PTE—Instruction Translation Buffer PTE Register

Index	Ibox 1
Access	R/W

The ITB_PTE IPR represents 12 page table entries split into two distinct arrays. The first eight PTEs provide small page (8 Kbytes) translations while the remaining four provide large page (4 Mbytes) translations. The entry to be written is selected by a not-last-used algorithm implemented in hardware for each array independently, and the status of the TB_CTL IPR. Writes to the ITB_PTE IPR use the memory format bit positions as described in the *Alpha Architecture Reference Manual*, with the exception that some fields are ignored.

Refer to the chapter discussing the appropriate operating system support in this manual for the bit definitions of the PTE.



ICCSR—Instruction Cache Control/Status Register

Address Ibox 2
Access R/W

The ICCSR IPR contains various Ibox hardware enables. The only architecturally defined bit in this register is the floating-point enable (FPE), which enables floating-point instructions. When cleared, all floating-point instructions generate FEN exceptions. Most bits of this IPR are cleared by hardware at reset. Fields that are not cleared at reset include ASN, PC0, and PC1.

NOTE: The hardware enable bit allows the PALcode instructions to execute in kernel mode. This bit is intended for diagnostic or operating system alternative PALcode routines only. It does not allow access to the ITB IPRs if not running in PALmode.

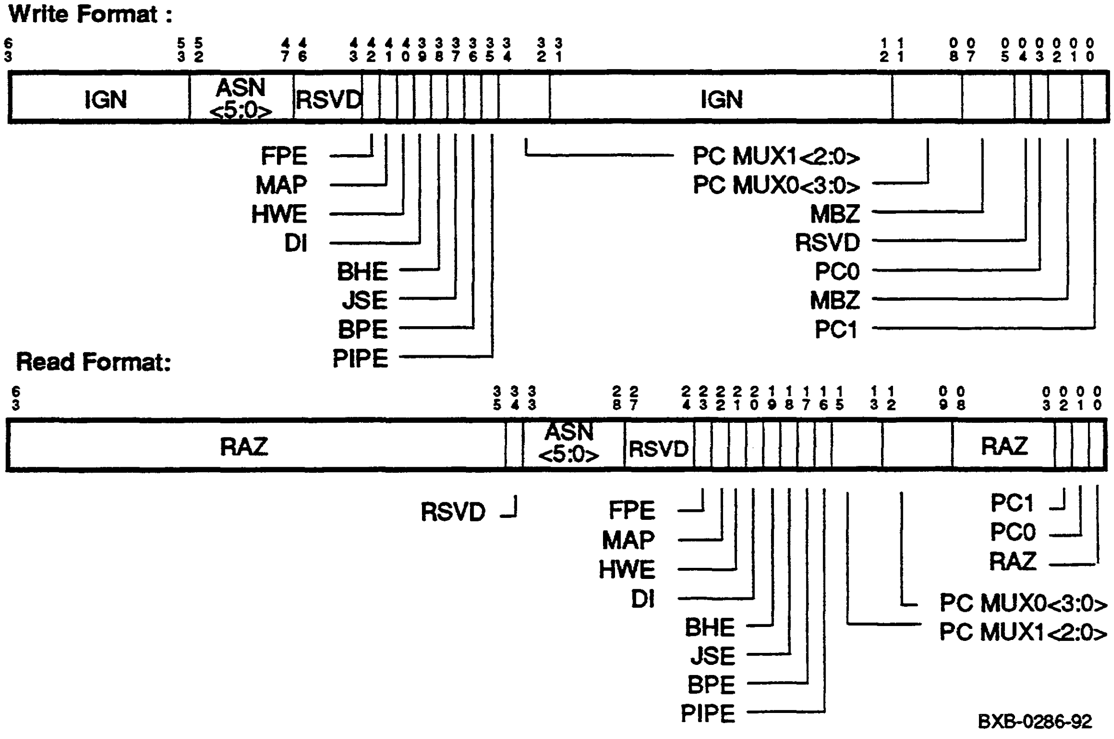


Table 4-10 ICCSR IPR Bit Definitions

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>ASN</td><td>W<52:47>
R<33:28></td><td>R/W, 0</td><td><b>Address Space Number.</b> The ASN field is used with the I-cache to further qualify cache entries and avoid some cache flushes. The ASN is written to the I-cache during fill operations and compared with the I-stream data on fetch operations. Mismatches invalidate the fetch without affecting the I-cache.</td></tr>
<tr><td>FPE</td><td>W<42>
R<23></td><td>R/W, 0</td><td><b>Floating-Point Enable.</b> If set, floating-point instructions can be issued. If clear, floating-point instructions cause FEN exceptions.</td></tr>
<tr><td>MAP</td><td>W<41>
R<22></td><td>R/W, 0</td><td><b>Map</b>. If set, it allows superpage I-stream memory mapping of virtual PC <33:13> directly to physical PC <33:13> essentially bypassing ITB for virtual PC addresses containing virtual PC <42:41> = 2. Superpage mapping is allowed in kernel mode only. The I-cache ASM bit is always set. If clear, superpage mapping is disabled.</td></tr>
<tr><td>HWE</td><td>W<40>
R<21></td><td>R/W, 0</td><td><b>Hardware Enable.</b> If set, it allows the five PALRES instructions (see the <i>Alpha Architecture Reference Manual</i>) to be issued in kernel mode. If cleared, attempts to execute PALRES instructions while not in PALmode result in OPCDEC exceptions.</td></tr>
<tr><td>DI</td><td>W<39>
R<20></td><td>R/W, 0</td><td><b>Dual Issue.</b> If set, dual-instruction issue is enabled. If cleared, instructions can only single issue.</td></tr>
<tr><td>BHE</td><td>W<38>
R<19></td><td>R/W, 0</td><td><b>Branch History Enable.</b> Used with BPE to select branch prediction.
<table>
<tr><th>BPE</th><th>BHE</th><th>Prediction</th></tr>
<tr><td>0</td><td>X</td><td>Not taken</td></tr>
<tr><td>1</td><td>0</td><td>Sign of displacement</td></tr>
<tr><td>1</td><td>1</td><td>Branch history table</td></tr>
</table></td></tr>
<tr><td>JSE</td><td>W<37>
R<18></td><td>R/W, 0</td><td><b>Jump Subroutine Enable.</b> If set, it enables the JSR stack to push a return address. If cleared, JSR stack is disabled.</td></tr>
<tr><td>BPE</td><td>W<36>
R<17></td><td>R/W, 0</td><td><b>Branch Prediction Enable.</b> Used with BHE to select branch prediction. See description of BHE above.</td></tr>
<tr><td>PIPE</td><td>W<38>
R<19></td><td>R/W, 0</td><td><b>Pipeline.</b> If clear, it causes all hardware interlocked instructions to drain the machine and waits for the write buffer to empty before issuing the next instruction. Examples of instructions that do not cause the pipe to drain include HW_MTPR, HW_REI, conditional branches, and instructions that have a destination register of R31. If set, pipeline proceeds normally.</td></tr>
</table>

Table 4-10 ICCSR IPR Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>PCMUX1</td><td>W<34:32>
R<15:13></td><td>R/W, 0</td><td>Performance Counter Mux 1.
<table>
<tr><th>MUX1</th><th>Input</th><th>Comment</th></tr>
<tr><td>000</td><td>D-cache miss</td><td>Counts total D-cache misses.</td></tr>
<tr><td>001</td><td>I-cache miss</td><td>Counts total I-cache misses.</td></tr>
<tr><td>010</td><td>Dual issues</td><td>Counts cycles of dual issue.</td></tr>
<tr><td>011</td><td>Branch mispredicts</td><td>Counts both conditional branch mispredictions and JSR or HW_REI mispredictions. Conditional branch mispredictions cost 4 cycles and others cost 5 cycles of pipeline delay.</td></tr>
<tr><td>100</td><td>FP instructions</td><td>Counts total floating-point operate instructions; that is, no FP branch, load, or store.</td></tr>
<tr><td>101</td><td>Integer operate</td><td>Counts integer operate instructions including LDA and LDAH with destination other than R31.</td></tr>
<tr><td>110</td><td>Store instructions</td><td>Counts total store instructions.</td></tr>
<tr><td>111</td><td>PERF_CNT_H = 1</td><td>Counts external events supplied to a pin at a selected system clock cycle interval.</td></tr>
</table></td></tr>
</table>

Table 4-10 ICCSR IPR Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>PCMUX0</td><td>W<11:8>
R<12:9></td><td>R/W, 0</td><td>Performance Counter Mux 0.
<table>
<tr><th>MUX0</th><th>Input</th><th>Comment</th></tr>
<tr><td>000X</td><td>Total Issues/2</td><td>Counts total issues divided by 2; dual issue increments count by 1.</td></tr>
<tr><td>001X</td><td>Pipeline Dry</td><td>Counts cycles where nothing issued due to lack of valid I-stream data. Causes include I-cache fill, misprediction, branch delay slots, and pipeline drain for exception.</td></tr>
<tr><td>010X</td><td>Load Instructions</td><td>Counts all Load instructions.</td></tr>
<tr><td>011X</td><td>Pipeline Frozen</td><td>Counts cycles where nothing issued due to resource conflict.</td></tr>
<tr><td>100X</td><td>Branch Instructions</td><td>Counts all conditional branches, unconditional branches, JSR, and HW_REI instructions.</td></tr>
<tr><td>1011</td><td>PALmode</td><td>Counts cycles while executing in PALmode.</td></tr>
<tr><td>1010</td><td>Total cycles</td><td>Counts total cycles.</td></tr>
<tr><td>110X</td><td>Total Non-issues/2</td><td>Counts total non_issues divided by 2; that is, no issue increments count by 1.</td></tr>
<tr><td>111X</td><td>PERF_CNT_H = 0</td><td>Counts external events supplied to a pin at a selected system clock cycle interval.</td></tr>
</table></td></tr>
<tr><td>PC1</td><td>W<0>
R<2></td><td>R/W</td><td><b>Performance Counter 1.</b> If clear, it enables performance counter 1 interrupt request after 2^{12} events counted. If set, it enables performance counter 1 interrupt request after 2^{8} events counted.</td></tr>
<tr><td>PC0</td><td>W<3>
R<1></td><td>R/W0</td><td><b>Performance Counter 0.</b> If clear, it enables performance counter 0 interrupt request after 2^{16} events counted. If set, it enables performance counter 0 interrupt request after 2^{12} events counted.</td></tr>
</table>

Performance Counters

The performance counters are reset to zero upon power-up. Otherwise they are never cleared. The counters are intended as a means of counting events over a long period of time, relative to the event frequency. They provide no means of extracting intermediate counter values.

Since the counters continuously accumulate selected events, despite interrupts being enabled, the first interrupt after selecting a new counter input has an error bound as large as the selected overflow range. Some inputs can overcount events occurring simultaneously with D-stream errors that abort the actual event very late in the pipeline.

For example, when counting load instructions, attempts to execute a load resulting in a TB miss exception will increment the performance counter after the first aborted execution attempt and again after the TB fill routine when the load instruction reissues and completes.

Performance counter interrupts are reported six cycles after the event that caused the counter to overflow. Additional delay can occur before an interrupt is serviced if the processor is executing PALcode that always disables interrupts. Events occurring during the interval between counter overflow and interrupt service are counted toward the next interrupt.

Only in the case of a complete counter wrap-around while interrupts are disabled will an interrupt be missed.

The six cycles before an interrupt is triggered implies that a maximum of 12 instructions may have completed before the start of the interrupt service routine.

When counting I-cache misses, no intervening instructions can complete and the exception PC contains the address of the last I-cache miss. Branch mispredictions allow a maximum of only two instructions to complete before start of the interrupt service routine.

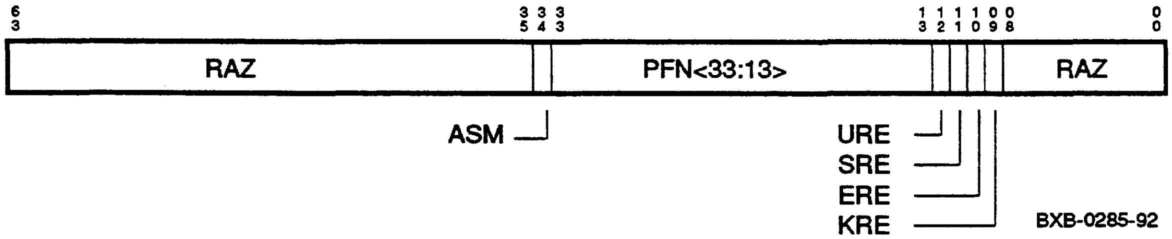
ITB_PTE_TEMP—Instruction Translation Buffer PTE_TEMP Register

Index	Ibox 3
Access	R

The ITB_PTE_TEMP IPR is a holding register for ITB_PTE read data. Reads of ITB_PTE require two instructions to return data to the register file. The two instructions are as follows:

1. **Read the ITB_PTE IPR data to the ITB_PTE_TEMP IPR.**
2. **Read the ITB_PTE_TEMP IPR data to the integer register file.**

The ITB_PTE_TEMP IPR is updated on all ITB accesses, both read and write. A read of the ITB_PTE to the ITB_PTE_TEMP should be followed closely by a read of the ITB_PTE_TEMP to the register file. Refer to the PTE descriptions in Chapters 9 and 10 for the bit definitions of this register.



EXC_ADDR—Exception Address Register

Index Ibox 4
Access R/W

The EXC_ADDR IPR is a read/write register used to restart the system after exceptions or interrupts.

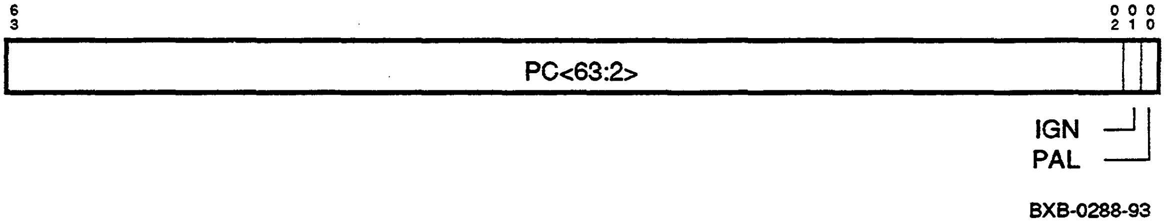


Table 4-11 EXC_ADDR IPR Bit Definitions

Name	Bit(s)	Type	Function
PC	<63:2>	R/W	**Program Counter.** Contains bits <63:2> of the PC. This field is written by hardware following an exception to provide a return address for PALcode.
PAL	<0>	R/W	**PALmode.** If set, the value in EXC_ADDR<63:2> is correct. When clear, the HW_REI instruction executes a jump to native (nonPALmode) mode, enabling address translation.

EXC_ADDR IPR Usage

The instruction pointed to by the EXC_ADDR IPR did not complete its execution. The EXC_ADDR IPR is written by hardware after an exception to provide a return address for PALcode. The HW_REI instruction executes a jump to the address contained in the EXC_ADDR IPR. EXC_ADDR<0> is used to indicate PALmode to the hardware.

CALL_PAL exceptions load the EXC_ADDR with the PC of the instruction following the CALL_PAL. This function allows CALL_PAL service routines to return without needing to increment the value in the EXC_ADDR IPR.

This feature requires careful treatment in PALcode. Arithmetic traps and machine check exceptions can prompt CALL_PAL exceptions resulting in an incorrect value being saved in the EXC_ADDR IPR. In the cases of an arithmetic trap or machine check exception (only in these cases), EXC_ADDR<1> takes on special meaning. PALcode servicing these two exceptions must:

- Interpret a zero in EXC_ADDR<1> as indicating that the PC in EXC_ADDR<63:2> is too large by a value of 4 bytes and subtract 4 before executing an HW_REI from this address.
- Interpret a one in EXC_ADDR<1> as indicating that the PC in EXC_ADDR<63:2> is correct and clear EXC_ADDR<1>.

All other PALcode entry points except reset can expect EXC_ADDR<1> to be zero.

The logic allows the following code sequence to conditionally subtract 4 from the address in the EXC_ADDR register without the use of an additional register. This code sequence must be present in arithmetic trap and machine check flows only.

```
HW_MFPR  Rx,   EXC_ADDR      ; read EXC_ADDR into GPR
SUBQ     Rx,   2, Rx         ; subtract 2 causing borrow
                             ; if bit <1>=0
BIC      Rx, 2, Rx           ; clear bit [1]
HW_MTPR Rx,   EXC_ADDR       ; write back to EXC_ADDR
```

NOTE: Using the HW_MTPR instruction to update the EXC_ADDR register while in the native mode is restricted to bit <0> being equal to zero. The combination of the native mode and EXC_ADDR<0> being equal to one causes UNDEFINED behavior. This combination is only possible through the use of ICCSR<HWE>.

SL_RCV—Serial Line Receive Register

Index Ibox 5
Access R

The SL_RCV IPR contains a single read-only bit (RCV) which is used with the interrupt control registers, the sRomD_h signal, and the sRomClk_h signal to provide an on-chip serial line function.

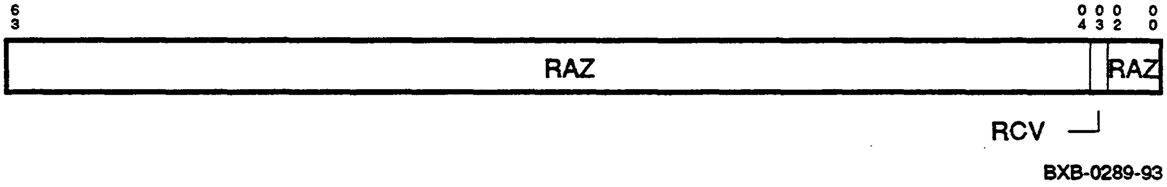


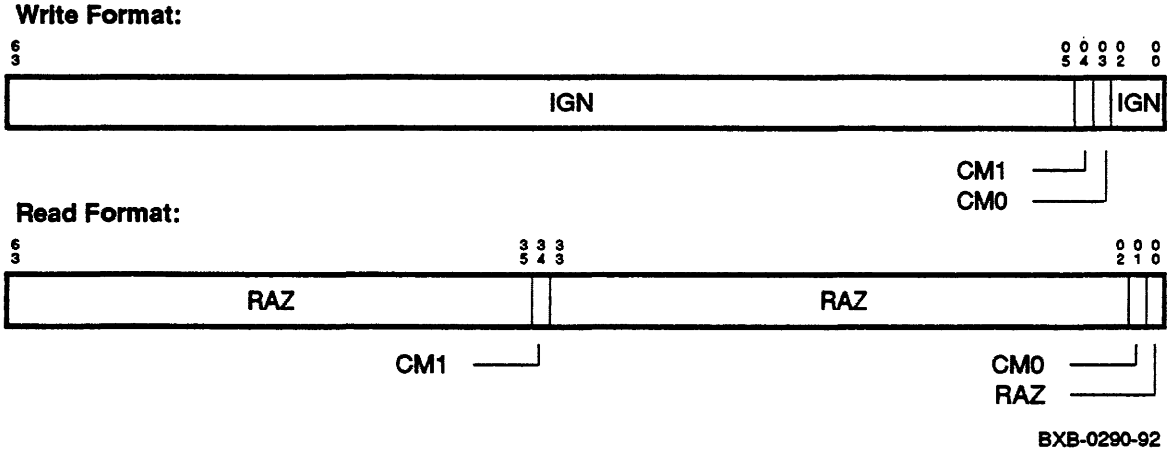
Table 4-12 SL_RCV IPR Bit Definitions

Name	Bit(s)	Type	Function
RCV	<3>	R	**Serial Line Receive.** This bit is functionally connected to the sRomD_h pin after the I-cache is loaded from the external SROM. Using a software timing loop, RCV can be read to receive external data one bit at a time. A serial line interrupt is requested on detection of any transition on the receive line that sets the SL_REQ bit in the HIRR IPR. The serial line interrupt can be disabled by clearing HIER<SL_EN>.

PS—Processor Status Register

Index Ibox 9
Access R/W

The PS IPR contains only the current mode bits of the architecturally defined PS. Refer to the *Alpha Architecture Reference Manual* for the functional descriptions of the current mode bits.



EXC_SUM—Exception Summary Register

Index	Ibox 10
Access	R/W

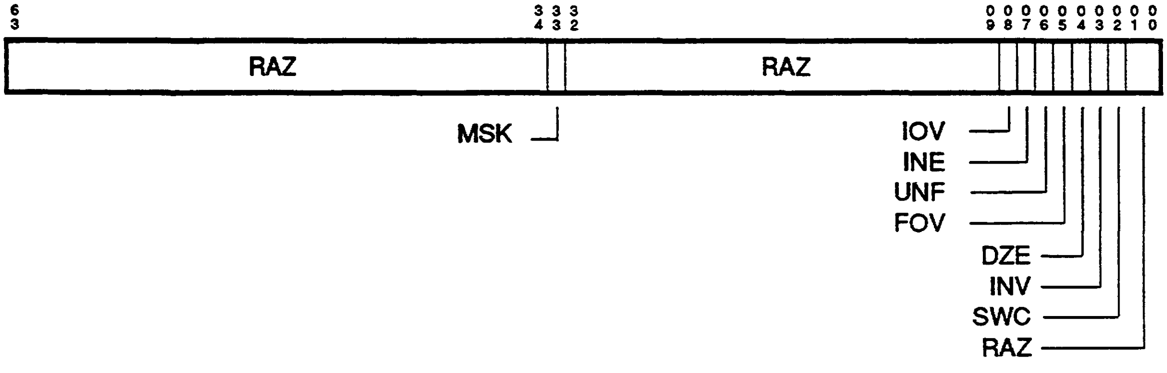
The EXC_SUM IPR records the various types of arithmetic traps that occurred since the last time the EXC_SUM was written (cleared). When the result of an arithmetic operation produces an arithmetic trap, the corresponding EXC_SUM bit is set.

The register containing the result of the operation is recorded in the Exception Register Write Mask parameter (see the *Alpha Architecture Reference Manual*), as a single bit in a 64-bit field specifying registers F31–F0 and I31–I0. The EXC_SUM IPR provides a one-bit window to the Exception Register Write Mask parameter. This is visible only through the EXC_SUM IPR.

Each read to the EXC_SUM shifts one bit in order F31–F0 then I31–I0. The read also clears the corresponding bit. The EXC_SUM must be read 64 times to extract the complete mask and clear the entire register. If no integer traps are present (IOV=0), only the first 32 bits of the corresponding register in the floating-point register file need to be read and cleared.

Any write to EXC_SUM clears bits <8:2> and does not affect the write mask bit.

The write mask parameter bit clears three cycles after a read. Code intended to read the parameter must allow at least three cycles between reads. This allows the clear and shift operations to complete in order to ensure reading successive bits.



BXB-0291-93

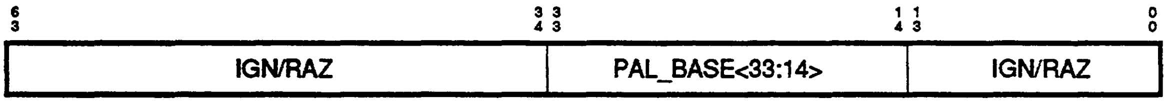
Table 4-13 EXC_SUM IPR Bit Definitions

Name	Bit(s)	Type	Function
MSK	<33>	RC	**Mask**. Exception Register Write Mask parameter window.
IOV	<8>	WA	**Integer Overflow.** When set, indicates Fbox convert to integer overflow or integer arithmetic overflow.
INE	<7>	WA	**Inexact Error.** When set, indicates floating inexact error.
UNF	<6>	WA	**Underflow.** When set, indicates floating-point underflow.
FOV	<5>	WA	**Floating-Point Overflow.** When set, indicates floating-point overflow.
DZE	<4>	WA	**Divide by Zero.** When set, indicates divide by zero.
INV	<3>	WA	**Invalid.** When set, indicates invalid operation.
SWC	<2>	WA	**Software Completion.** When set, indicates software completion possible. The bit is set after a floating-point instruction containing the /S modifier completes with an arithmetic trap and all previous floating-point instructions that trapped since the last HW_MTPR EXC_SUM also contained the /S modifier. The SWC bit is cleared whenever a floating-point instruction without the /S modifier completes with an arithmetic trap. The bit remains cleared regardless of additional arithmetic traps until the register is written by way of an HW_MTPR instruction. The SWC bit is always cleared upon any HW_MTPR write to the EXC_SUM IPR.

PAL_BASE—PALcode Base Address Register

Index Ibox 11
Access R/W

The PAL_BASE IPR contains the base address for PALcode. This register is cleared by the hardware at reset. It establishes the reference (base) address to which an offset is added to determine the entry point to the PALcode.



BXB-00292-93

Table 4-14 PAL_BASE IPR Bit Definitions

Name	Bit(s)	Type	Function
PAL_BASE	<33:14>	R/W	**PALcode Base Address.** Contains the PALcode base address.

HIRR—Hardware Interrupt Request Register

Index Ibox 12
Access R

The HIRR IPR provides a record of all currently outstanding interrupt requests and summary bits at the time of the read. For each bit of the HIRR<5:0>, there is a corresponding bit of the Hardware Interrupt Enable IPR (HIER) that must be set to enable that interrupt.

In addition to returning the status of the hardware interrupt requests, a read of the HIRR returns the state of the software interrupt and AST requests.

NOTE: A read of the HIRR can return a value of zero if the hardware interrupt was released before the read (passive release).

The register guarantees that the HWR bit reflects the status as shown by the HIRR bits. All interrupt requests are blocked while executing in PALmode.

Read Format:

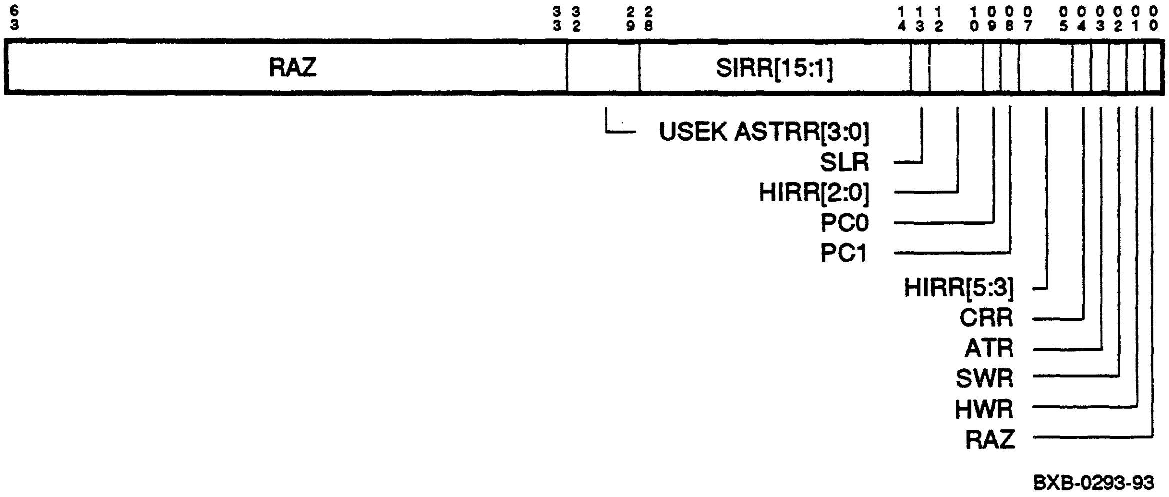


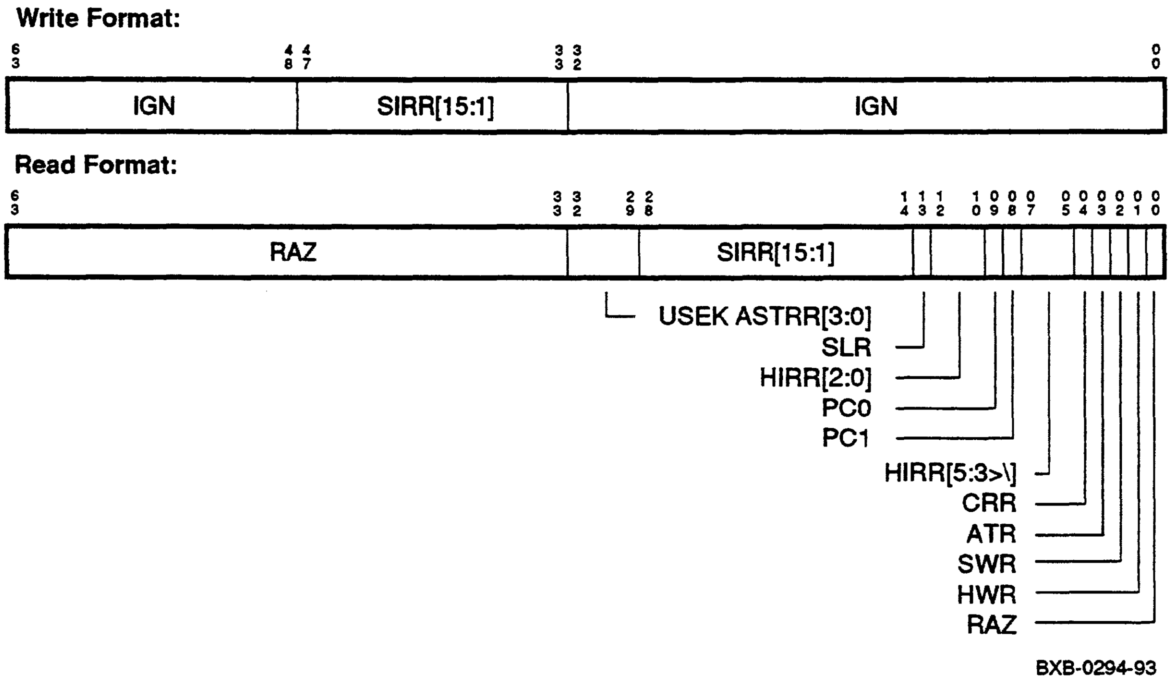
Table 4-15 HIRR IPR Bit Definitions

Name	Bit(s)	Type	Function
ASTRR[3:0]	<32:29>	R	**AST Request.** Corresponds to AST requests 3 through 0 (USEK). When a bit is set, the corresponding interrupt request is posted. The four bits are expanded as follows: ASTRR — Operating Mode 3 — UAR: User AST request 2 — SAR: Supervisor AST request 1 — EAR: Executive AST request 0 — KAR: Kernel AST request
SIRR[15:1]	<28:14>	R	**Software Interrupt Request.** Corresponds to software interrupt requests 15 through 1. When a bit is set, the corresponding interrupt request is posted.
SLR	<13>	R	**Serial Line Interrupt Request.** When set, a serial line interrupt request is posted. See also SL_RCV, SL_XMIT, and SL_CLR.
HIRR[5:0]	<12:10> <7:5>	R	**Hardware Interrupt Request.** Reflects the state of signals Irq_h [5:0]. Any bit set in the HIRR field indicates an interrupt request on the corresponding Irq_h line.
PC0	<9>	R	**Performance Counter 0 Interrupt Request.** When set, indicates that an interrupt request is posted by PC0.
PC1	<8>	R	**Performance Counter 1 Interrupt Request.** When set, indicates that an interrupt request is posted by PC1.
CRR	<4>	R	**Correctable Read.** When set, indicates that a correctable read error interrupt request is posted. This interrupt is cleared by way of the SL_CLR IPR.
ATR	<3>	R	**Asynchronous Trap Request.** Is set if any AST request and corresponding enable is set. This bit also requires that the processor mode be equal to or higher than the request mode. SIER[2] must be asserted to allow AST interrupt requests.
SWR	<2>	R	**Software.** Is set if any software interrupt request and corresponding enable is set.
HWR	<1>	R	**Hardware.** Is set if any hardware interrupt request and corresponding enable is set.

SIRR—Software Interrupt Request Register

Index Ibox 13
Access R/W

The SIRR IPR is used to control software interrupt requests. For each bit of the SIRR there is a corresponding bit of the Software Interrupt Enable IPR (SIER) that must be set to request an interrupt. Reads of the SIRR return the complete set of interrupt request registers and summary bits. All interrupt requests are blocked while executing in PALmode. See Table 4-15 for the SIRR IPR bit definitions.

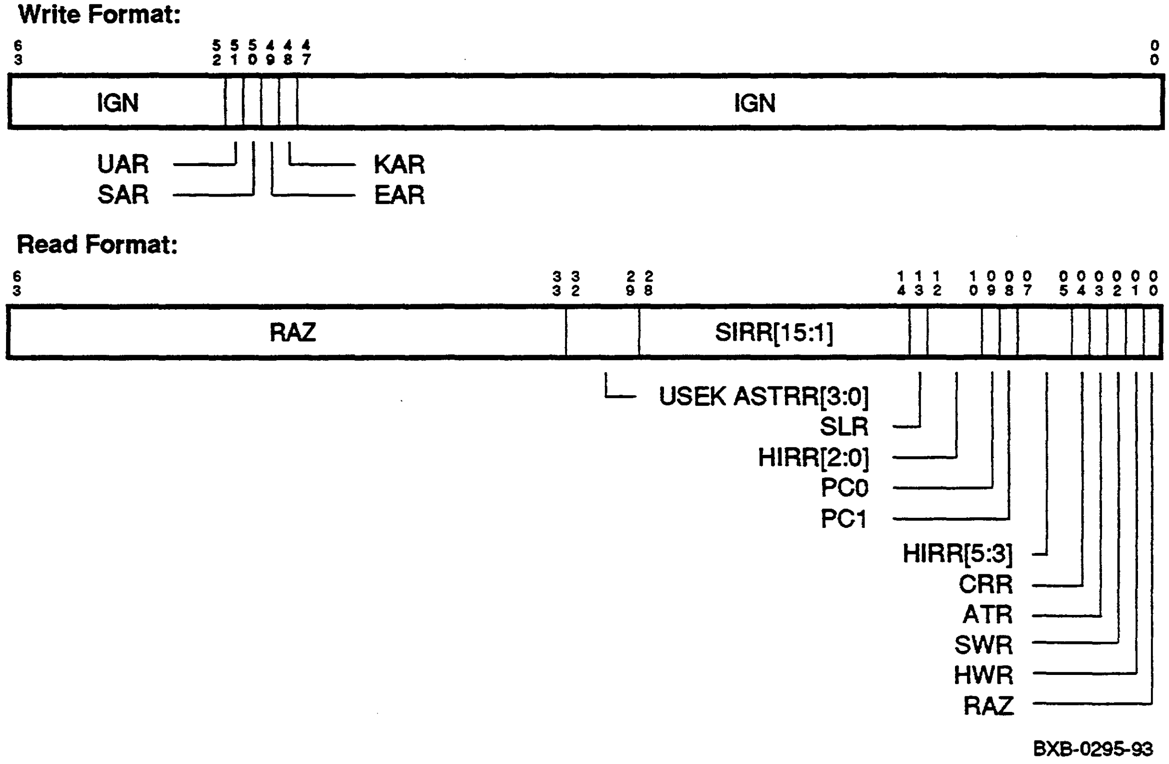


ASTRR—Asynchronous Trap Request Register

Index Ibox 14
Access R/W

The ASTRR IPR contains bits to request AST interrupts in each of the processor modes. To generate an AST interrupt, the corresponding enable bit in the ASTER IPR must be set. Also, the processor must be in the selected processor mode or higher privilege as described by the current value of the PS CM bits. AST interrupts are enabled if SIER[2] is asserted. This provides a mechanism to lock out AST requests over certain IPL levels.

All interrupt requests are blocked while executing in PALmode. Reads of the ASTRR IPR return the complete set of interrupt request registers and summary bits. See Table 4-15 for the ASTRR IPR bit definitions.



HIER—Hardware Interrupt Enable Register

Index Ibox 16
Access R/W

The HIER IPR is used to enable corresponding bits of the HIRR requesting interrupt. The PC0, PC1, SLE, and CRE bits of this register enable:

1. **Performance counter interrupts**
2. **Serial line interrupts**
3. **Correctable read error interrupts**

There is a one-to-one correspondence between the interrupt requests and enable bits. As with the reads of the interrupt request registers, reads of the HIER IPR return the complete set of interrupt enable registers. See Table 4-15 for details.

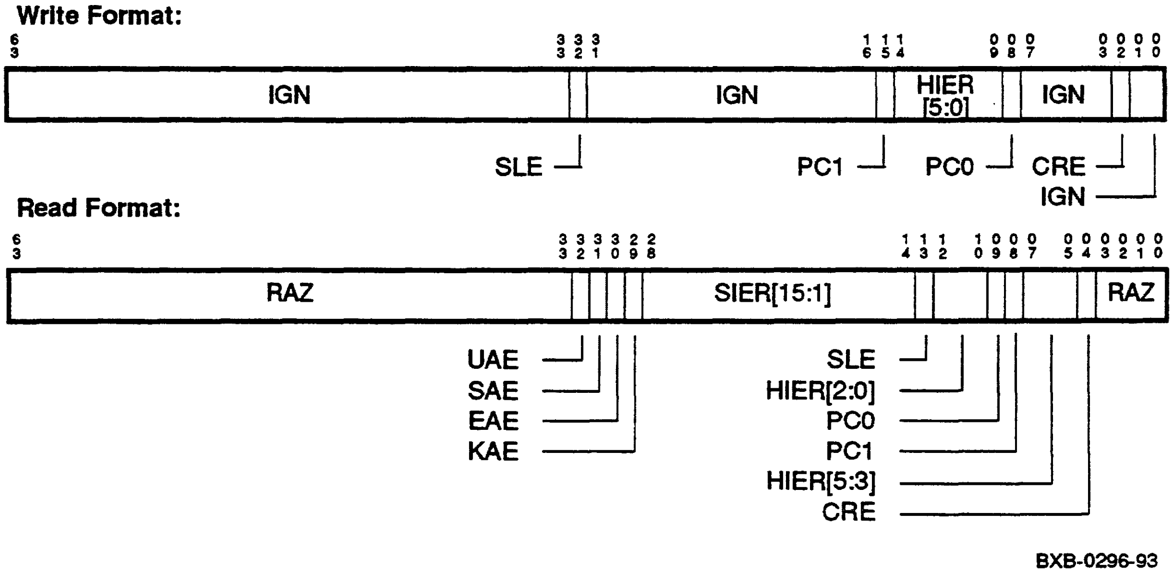


Table 4-16 HIER IPR Bit Definitions

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>SLE</td><td>W<32>
R<13></td><td>R/W</td><td><b>Serial Line Interrupt Enable.</b> If set, enables the serial line interrupts. See also SL_RCV, SL_XMIT, and SL_CLR.</td></tr>
<tr><td>ASTRR[3:0]</td><td><32:29></td><td>R</td><td><b>AST Interrupt Enable.</b> Corresponds to AST interrupt enable bits 3 through 0 (USEK). If a bit is set in this field, the corresponding interrupt is enabled. The four bits are expanded as follows:
<table>
<tr><th>ASTRR(3:0)</th><th>Operating Mode</th></tr>
<tr><td>3</td><td>UAE: User AST interrupt enable</td></tr>
<tr><td>2</td><td>SAE: Supervisor AST interrupt enable</td></tr>
<tr><td>1</td><td>EAE: Executive AST interrupt enable</td></tr>
<tr><td>0</td><td>KAE: Kernel AST interrupt enable</td></tr>
</table></td></tr>
<tr><td>SIER[15:1]</td><td><28:14></td><td>R</td><td><b>Software Interrupt Enable.</b> Corresponds to software interrupt requests 15 through 1. Any bit set in this field indicates that the corresponding software interrupt is enabled.</td></tr>
<tr><td>PC1</td><td>W<15>
R<8></td><td>R/W</td><td><b>Performance Counter 1 Interrupt Enable.</b> When set, enables PC1 interrupts.</td></tr>
<tr><td>HIER[5:0]</td><td>W<14:9>
R<12:10>
R<7:5></td><td>R/W</td><td><b>Hardware Interrupt Enable.</b> Interrupt enable bits for signals Irq_h[5:0]. Any bit set in this field enables the corresponding hardware interrupt.</td></tr>
<tr><td>PC0</td><td>W<8>
R<9></td><td>R/W</td><td><b>Performance Counter 0 Interrupt Enable.</b> When set, enables PC0 interrupts.</td></tr>
<tr><td>CRE</td><td>W<2>
R<4></td><td>R/W</td><td><b>Correctable Read Error Interrupt Enable.</b> If set, enables the interrupt. The interrupt request is cleared by way of the SL_CLR.</td></tr>
</table>

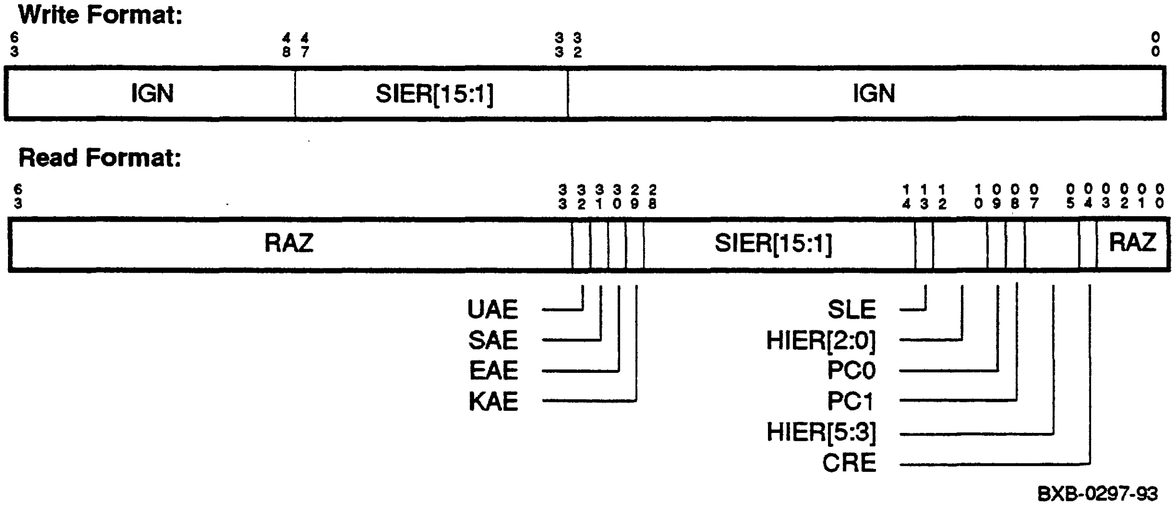
SIER—Software Interrupt Enable Register

Index Ibox 17

Access R/W

The SIER IPR is used to enable corresponding bits of the SIRR requesting interrupts. There is a one-to-one correspondence between the interrupt requests and enable bits. As with the reads of the interrupt request registers, reads of the SIER return the complete set of interrupt enable registers.

Refer to Table 4-16 for bit definitions of the SIER.

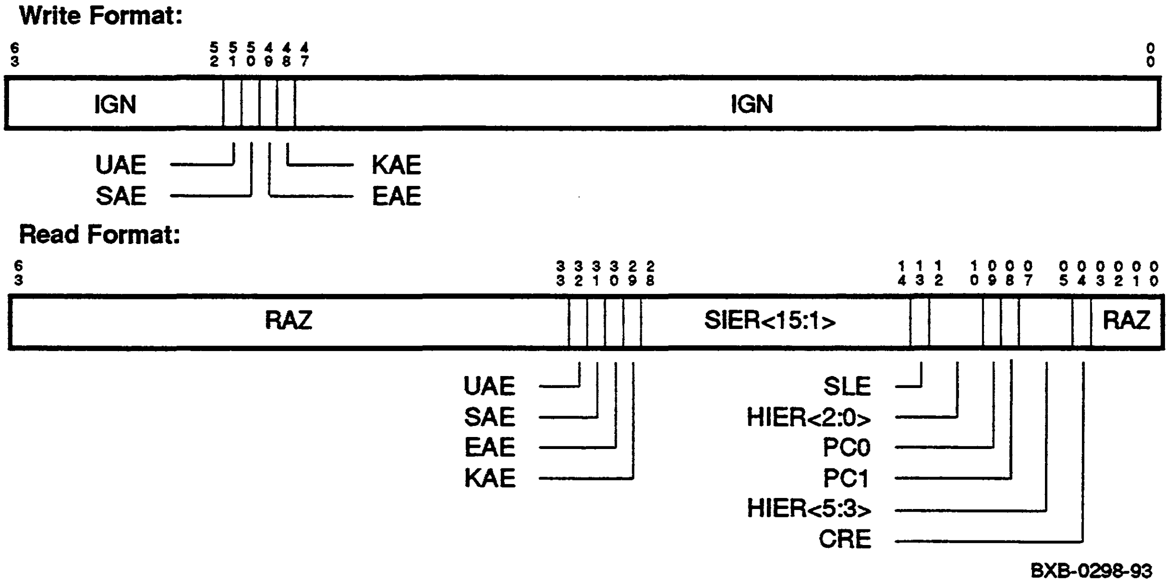


ASTER—AST Interrupt Enable Register

Index Ibox 18
Access R/W

The ASTER IPR is used to enable corresponding bits of the ASTRR requesting interrupts. There is a one-to-one correspondence between the interrupt requests and enable bits. As with the reads of the interrupt request registers, reads of the ASTER return the complete set of interrupt enable registers.

Refer to Table 4-16 for bit definitions of the ASTER.



SL_CLR—Interrupt Clear Serial Line Register

Index Ibox 19
Access W

The SL_CLR IPR is a write-only register that clears:

1. **Serial line interrupt requests**
2. **Performance counter interrupt requests**
3. **CRD interrupt requests**

The indicated bit must be written with a zero to clear the selected interrupt source.

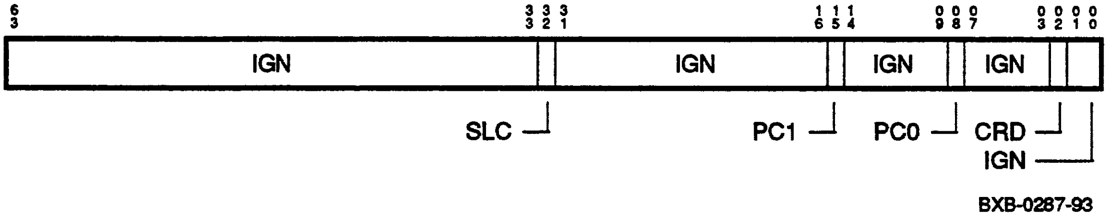


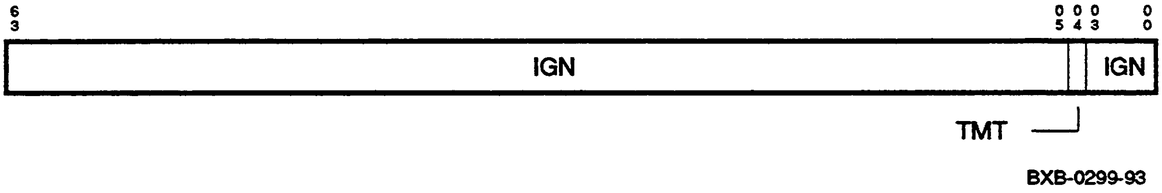
Table 4-17 SL_CLR IPR Bit Definitions

Name	Bit(s)	Type	Function
SLC	<32>	W0C	**Serial Line Clear.** Clears the serial line interrupt request.
PC1	<15>	W0C	**Performance Counter 1.** Clears the performance counter 1 interrupt request.
PC0	<8>	W0C	**Performance Counter 0.** Clears the performance counter 0 interrupt request.
CRD	<2>	W0C	**Correctable Read.** Clears the correctable read error interrupt request.

SL_XMIT—Serial Line Transmit Register

Index Ibox 22
Access W

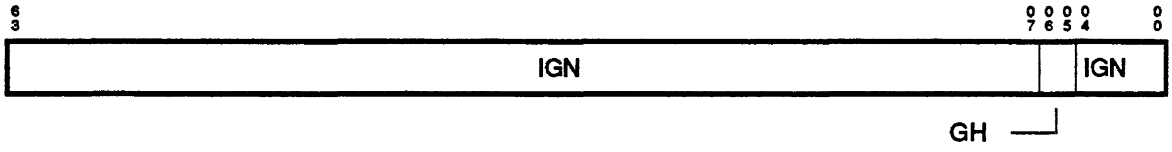
The SL_XMIT IPR contains a single write-only bit. This bit is used with the interrupt control registers, the sRomD_h signal, and the sRomClk_h signal to provide an on-chip serial line function. The TMT bit is functionally connected to the sRomClk_h signal after the I-cache is loaded from the external SROM. Writing the TMT bit can be used to transmit data off chip, one bit at a time, under a software timing loop.



TB_CTL—Translation Buffer Control Register

Index Abox 0
Access W

The TB_CTL IPR controls the granularity of the translation buffer.



BXB-0600-93

Table 4-18 TB_CTL IPR Bit Definitions

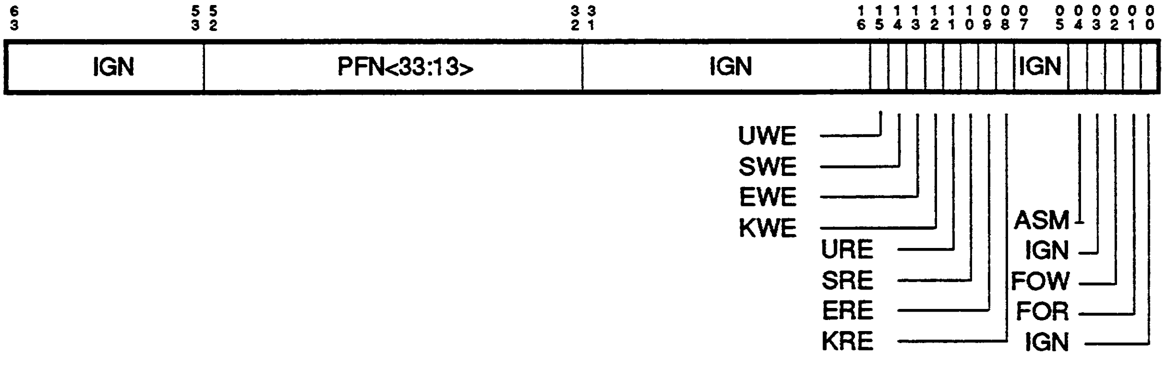
<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>GH</td><td>&lt;6:5&gt;</td><td>R/W</td><td><b>Granularity Hint.</b> Selects between the DECchip 21064 TB page mapping sizes when writing or reading the ITB and the DTB. There are two sizes in the ITB and four sizes in the DTB.
<table>
<tr><th>TB_CTL&lt;6:5&gt;</th><th>ITB Page Size</th><th>DTB Page Size</th></tr>
<tr><td>00</td><td>8 Kbytes</td><td>8 Kbytes</td></tr>
<tr><td>01</td><td>8 Kbytes</td><td>8*8 Kbytes</td></tr>
<tr><td>10</td><td>8 Kbytes</td><td>64*8 Kbytes</td></tr>
<tr><td>11</td><td>512*8 Kbytes</td><td>512*8 Kbytes</td></tr>
</table>
</td></tr>
</table>

DTB_PTE—Data Translation Buffer PTE Register

Index Abox 2
Access R/W

The DTB_PTE IPR represents the 32-entry DTB. The entry to be written is chosen by a not-last-used (NLU) algorithm implemented in the hardware. Writes to the DTB_PTE IPR use the memory format bit positions as described in the *Alpha Architecture Reference Manual,* with the exception that some fields are ignored. The valid bit is not represented in hardware.

Refer to the chapter discussing the appropriate operating system support in this manual for the bit definitions of the PTE.



BXB-0601-93

DTB_PTE_TEMP—Data Translation Buffer PTE_TEMP Register

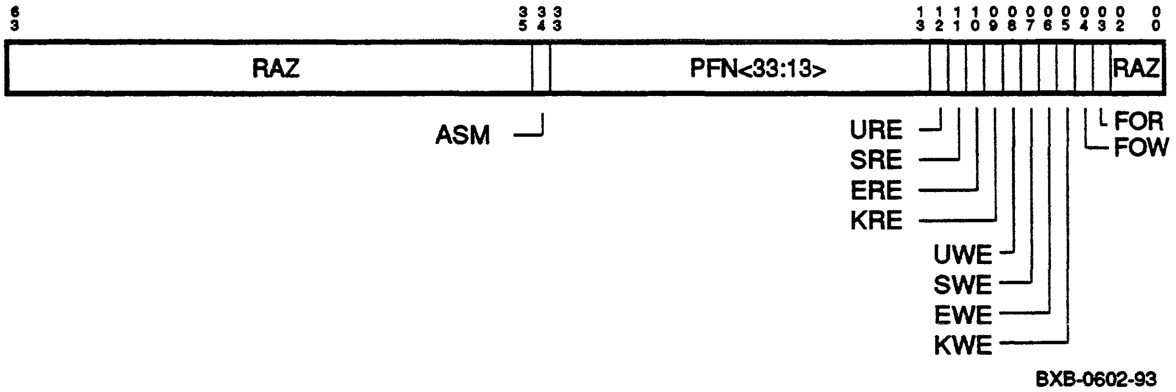
Index Abox 3
Access R

The DTB_PTE_TEMP IPR is a holding register for the DTB_PTE read data. Reads of the DTB_PTE require two instructions to return the data to the register file. The two instructions are as follows:

1. Read the DTB_PTE register data to the DTB_PTE_TEMP.
2. Read the DTB_PTE_TEMP register data to the integer register file.

The ITB_PTE_TEMP IPR is updated on all ITB accesses, both read and write. A read of the ITB_PTE to the ITB_PTE_TEMP should be followed closely by a read of the ITB_PTE_TEMP to the register file.

Refer to Chapter 9 (OpenVMS AXP System Support) or Chapter 10 (DEC OSF/1 AXP System Support) for the bit definitions of the PTE.



MMCSR—Memory Management CSR Register

Index Abox 4
Access R

The MMCSR IPR saves information about D-stream faults. The VA and MMCSR IPRs are locked against further updates until the software reads the VA. PALcode must explicitly unlock this register whenever its entry point is higher in priority than a DTB miss. The MMCSR bits are only modified by the hardware when the register is not locked, and a memory management error or a DTB miss occurs. The MMCSR IPR is unlocked after reset.

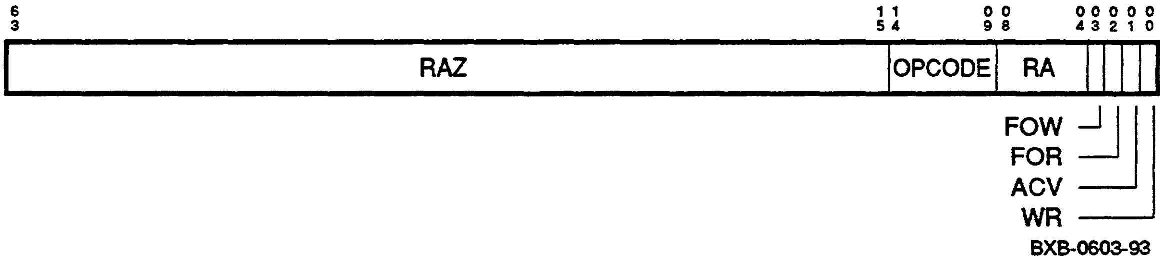


Table 4-19 MMCSR IPR Bit Definitions

Name	Bit(s)	Type	Function
OPCODE	<14:9>	R	**Opcode.** Contains the Opcode field of the faulting instruction.
RA	<8:4>	R	**Register A.** The RA field of the faulting instruction.
FOW	<3>	R	**Fault on Write.** Set if the reference was a write and the PTE's FOW bit was set.
FOR	<2>	R	**Fault on Read.** Set if the reference was a read and the PTE's FOR bit was set.
ACV	<1>	R	**Access Violation.** Set if reference caused an access violation.
WR	<0>	R	**Write.** Set if reference that caused error was a write.

BIU_ADDR—BIU Address Register

Index Abox 9
Access R

The BIU_ADDR IPR contains the physical address associated with errors reported by BIU_STAT<7:0>. Its contents are meaningful only when one of BIU_HERR, BIU_SERR, BC_TPERR, or BC_TCPERR are set. Reads of the BIU_ADDR register unlock both BIU_ADDR and BIU_STAT<7:0>.

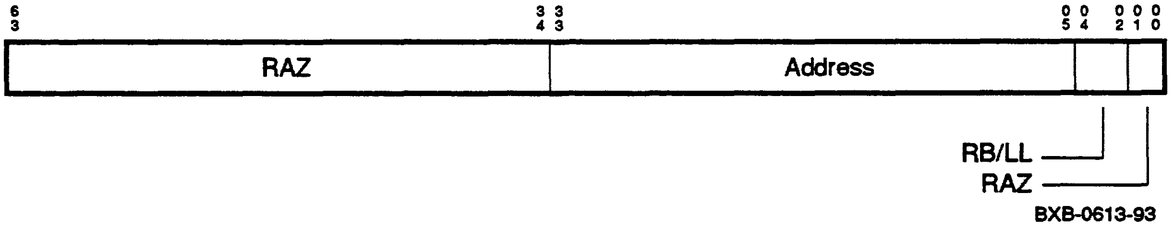


Table 4-20 BIU_ADDR IPR Bit Definitions

Name	Bit(s)	Type	Function
ADDRESS	<33:5>	R	**Address.** Reflects the states of adr_h signals [33:5] associated with the EDAL interface transaction that resulted in the error indicated in BIU_STAT<7:0>.
RB/LL	<4:2>	R	**Read_Block or Load_Locked.** If the BIU_CMD field of the BIU_STAT IPR indicates that the transaction that received the error was Read_Block or Load_Locked, then the state of RB/LL is UNPREDICTABLE. If the BIU_CMD field of the BIU_STAT IPR encodes any EDAL interface command other than Read_Block or Load_Locked, then RB/LL reads as zeros.

BIU_STAT—BIU Status Register

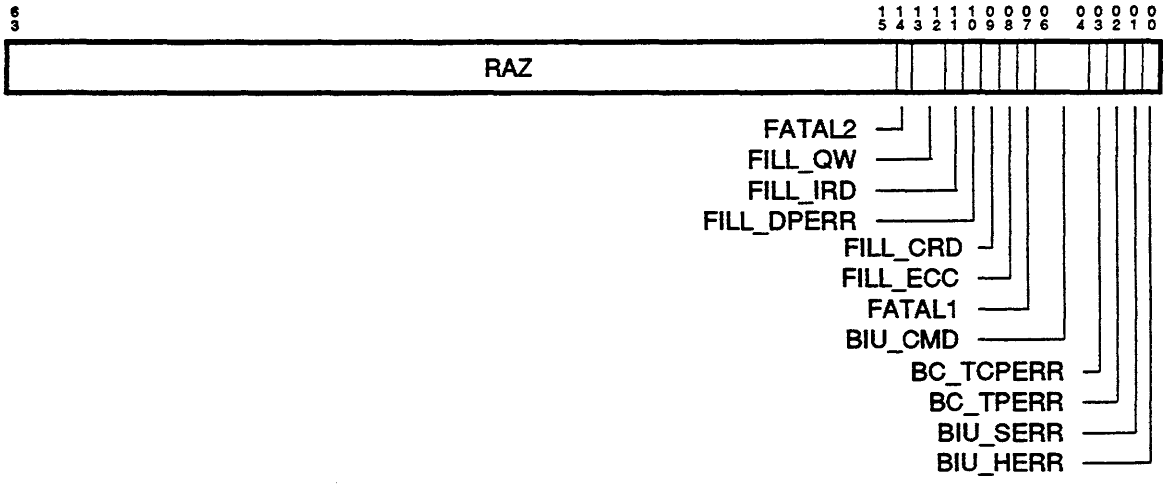
Index Abox 10
Access R

Bits <6:0> of the BIU_STAT IPR are locked against further updates when one of the following bits is set:

BIU_HERR
BIU_SERR
BC_TPERR
BC_TCPERR

The address associated with the error is latched and locked in the BIU_ADDR IPR. BIU_STAT<7:0> and BIU_ADDR are unlocked when the BIU_ADDR register is read. When FILL_ECC or FILL_DPERR is set, BIU_STAT<13:8> are locked against further updates. The address associated with the error is latched and locked in the FILL_ADDR IPR. BIU_STAT <14:8> and FILL_ADDR are unlocked when the FILL_ADDR IPR is read.

This register is not unlocked or cleared by reset and needs to be explicitly cleared by PALcode.



BXB-0608-93

Table 4-21 BIU_STAT IPR Bit Definitions

Name	Bit(s)	Type	Function
FATAL2	<14>	R	**Fatal 2.** When set, indicates that a primary cache fill operation resulted in either a multi-bit ECC error or in a parity error while FILL_ECC or FILL_DPERR was already set.
FILL_QW	<13:12>	R	**Fill Quadword.** Identifies the quadword within the hexword primary cache fill block that caused the error. This field is only meaningful when either FILL_ECC or FILL_DPERR is set. FILL_QW can be used together with FILL_ADDR<33:5> to get the complete physical address of the bad quadword.
FILL_IRD	<11>	R	**Fill I-Cache Read.** When set, indicates that the error that caused FILL_ECC or FILL_DPERR to be set occurred during an I-cache fill. When clear, indicates that the error occurred during a D-cache fill. This bit is only meaningful when either FILL_ECC or FILL_DPERR is set.
FILL_DPERR	<10>	R	**Fill Data Parity Error.** When set, indicates that the BIU received data with a parity error from outside the CPU chip while performing either a D-cache or I-cache fill. FILL_DPERR is only meaningful when the CPU chip is in parity mode, as opposed to ECC mode.
FILL_CRD	<9>	R	**Fill Correctable Read.** When set, indicates that the information latched in BIU_STAT <13:8>, FILL_ADDR IPR, and FILL_SYND IPR relates to an errored quadword that does not contain multi-bit errors in either of its component longwords. This bit is only meaningful when FILL_ECC is set.
FILL_ECC	<8>	R	**Fill ECC Error.** When set, indicates that P-cache fill data received from outside the CPU chip contained an ECC error.
FATAL1	<7>	R	**Fatal 1.** When set, indicates that an external cycle was terminated with the cAck_h pins indicating HARD_ERROR, or that a B-cache tag probe encountered bad parity in the tag address RAM or the tag control RAM while one of BIU_HERR, BIU_SERR, BC_TPERR, or BC_TCPERR was already set.
BIU_CMD	<6:4>	R	**BIU Command.** Latches the cycle type on the cReq_h pins when a BIU_HERR, BIU_SERR, BC_TPERR, or BC_TCPERR error occurs.

Table 4-21 BIU_STAT IPR Bit Definitions (Continued)

Name	Bit(s)	Type	Function
BC_TCPERR	<3>	R	**B-Cache Tag Control Parity Error.** When set, indicates that an external cache tag probe encountered bad parity in the tag control RAM.
BC_TPERR	<2>	R	**B-Cache Tag Parity Error.** When set, indicates that an external cache tag probe encountered bad parity in the tag address RAM.
BIU_SERR	<1>	R	**BIU Soft Error.** When set, indicates that an external cycle was terminated with the cAck_h pins indicating SOFT_ERROR.
BIU_HERR	<0>	R	**BIU Hard Error.** When set, indicates that an external cycle was terminated with the cAck_h pins indicating HARD_ERROR.

DC_STAT—D-Cache Status Register

Index Abox 12
Access R

The DC_STAT IPR is intended for use by diagnostics. For chip revisions less than 3, PALcode must first issue the following instruction before issuing the load or store whose D-cache lookup result is to be recorded into DC_HIT:

```
HW_MTPR R31, 4B (hex)
```

For pass 3 chips, software need not execute the HW_MTPR instruction before using DC__STAT. Also, the field marked Unpredictable reads zero in the pass 3 chips.

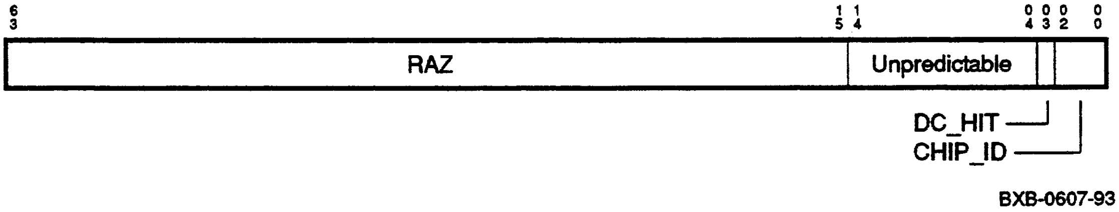


Table 4-22 DC_STAT IPR Bit Definitions

Name	Bit(s)	Type	Function
DC_HIT	<3>	R	**D-Cache Hit.** Indicates whether the last load or store instruction processed by the Abox hit (DC_HIT set) or missed (DC_HIT clear) the D-cache. Loads that miss the D-cache can be completed without requiring external reads.
CHIP_ID	<2:0>	R	**Chip Identification.** This field has a value of 111 (bin) for Revision 3 DECchip 21064 processors. Any other value in this field indicates a lower revision level.

FILL_ADDR—Fill Address Register

Index Abox 13
Access R

The FILL_ADDR IPR stores the physical address associated with errors reported by BIU_STAT<14:8>. The contents of this IPR are meaningful only when FILL_ECC or FILL_DPERR is set. Reads of the FILL_ADDR unlock FILL_ADDR, BIU_STAT<14:8>, and FILL_SYNDROME.

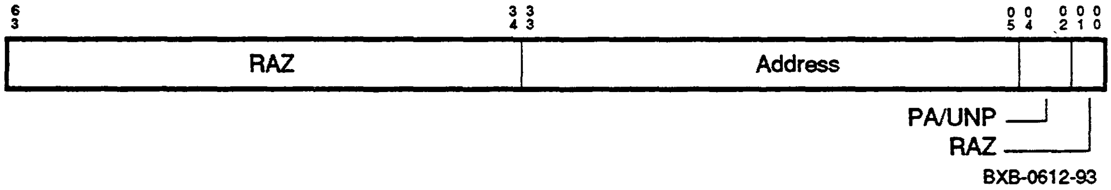


Table 4-23 FILL_ADDR IPR Bit Definitions

Name	Bit(s)	Type	Function
ADDRESS	<33:5>	R	**Address.** Identifies the 32-byte cache block that the CPU was attempting to read when the error occurred.
PA/UNP	<4:2>	R	**Physical Address or Unpredictable.** If the FILL_IRD bit of the BIU_STAT IPR is clear, it indicates that the error occurred during a D-stream cache fill. At such times, PA/UNP contains bits <4:2> of the physical address generated by the load instruction that triggered the cache fill. If FILL_IRD is set, then the state of PA/UNP is UNPREDICTABLE.

ABOX_CTL—Abox Control Register

Index Abox 14
Access W

The ABX_CTL IPR controls the Abox functions. PALcode writes to this register at initialization and keeps an image of the register which appears in error log entries and is readable by the user.

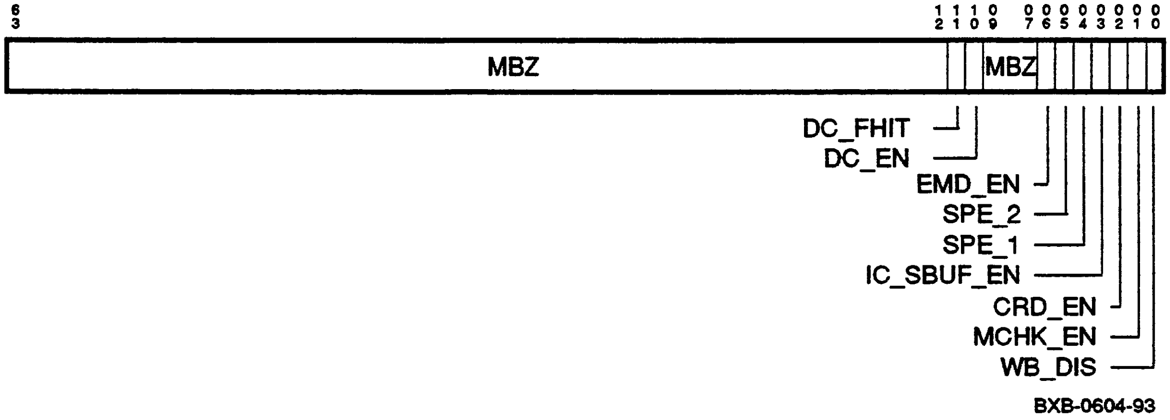


Table 4-24 ABX_CTL IPR Bit Definitions

Name	Bit(s)	Type	Function
DC_FHIT	<11>	W, 0	**D-Cache Force Hit.** When set, this bit forces all D-stream references to hit in the D-cache. This bit takes precedence over DC_EN. That is, when DC_FHIT is set and DC_EN is clear, all D-stream references hit in the D-cache.
DC_EN	<10>	W, 0	**D-Cache Enable.** When clear, this bit disables and flushes the D-cache. When set, this bit enables the D-cache.
EMD_EN	<6>	W, 0	**Endian Mode Enable.** Used to provide limited hardware support for big endian data formats. When set, this bit inverts the physical address bit <2> for all D-stream references. The chip endian mode is only selected during PALcode initialization.

Table 4-24 ABX_CTL IPR Bit Definitions (Continued)

Name	Bit(s)	Type	Function
SPE_2	<5>	W, 0	**Superpage Enable 2.** When set, enables one-to-one superpage mapping of the D-stream virtual addresses with VA <33:13> directly to physical addresses PA <33:13>, if virtual address bits VA <42:41> = 2. Virtual address bits VA <40:34> are ignored in this translation. Access is only allowed in kernel mode.
SPE_1	<4>	W, 0	**Superpage Enable 1.** When set, enables one-to-one superpage mapping of the D-stream virtual addresses with VA <42:30> = 1FFE to the physical addresses with PA <33:30> = 0. Access is only allowed in kernel mode.
IC_SBUF_EN	<3>	W, 0	**I-Cache Stream Buffer Enable.** When set, enables operation of a single-entry I-cache stream buffer.
CRD_EN	<2>	W, 0	**Correctable Read Interrupt Enable.** When set, the Abox generates an interrupt request whenever an EDAL interface transaction is terminated with a cAck_h code of SOFT_ERROR.
MCHK_EN	<1>	W, 0	**Machine Check Enable.** When set, the Abox generates a machine check when errors (that are not correctable by the hardware) are encountered. When cleared, uncorrectable errors do not cause a machine check. However, the BIU_STAT, DC_STAT, BIU_ADDR, and FILL_ADDR IPRs are updated and locked when the errors occur.
WB_DIS	<0>	W, 0	**Write Buffer Unload Disable.** When set, prevents the write buffer from sending write data to the BIU. This bit should only be set by diagnostics.

ALT_MODE—Alternate Processor Mode Register

Index Abox 15
Access W

The ALT_MODE IPR stores information that specifies the alternate processor mode.

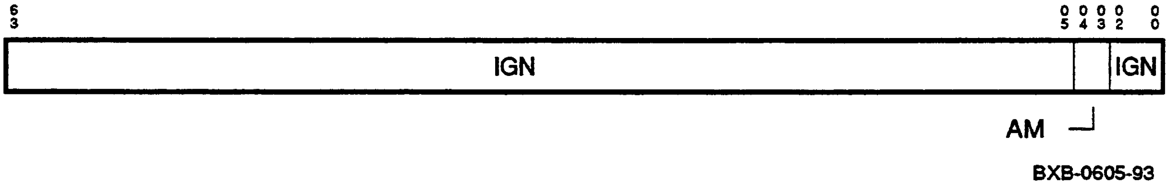


Table 4-25 ALT_MODE IPR Bit Definitions

Name	Bit(s)	Type	Function
AM	<4:3>	W	**Alternate Mode.** Specifies the alternate processor mode used by HW_LD and HW_ST instructions that have their ALT bit (<14>) set. The alternate modes are selected as follows: AM — Processor Mode 00 — Kernel 01 — Executive 10 — Supervisor 11 — User

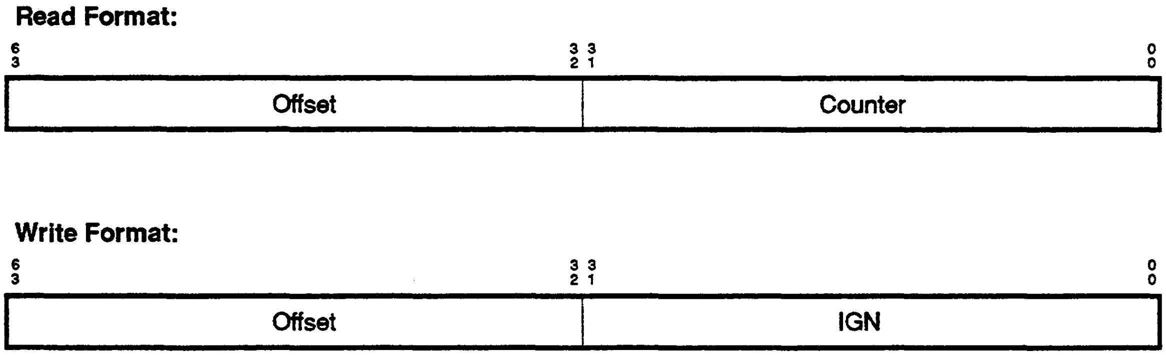
CC—Cycle Counter Register

Index Abox 16

Access W

The DECchip 21064 supports a cycle counter, as described in the *Alpha Architecture Reference Manual.* When enabled, the CC IPR increments once each CPU cycle. The HW_MTPR Rn, CC writes the CC<63:32> with the value held in the Rn<63:32>; bits <31:0> are not changed.

This IPR is read by the RPCC instruction and is written to by the HW_MTPR Rn, CC instruction as defined in the *Alpha Architecture Reference Manual.*


Read Format:

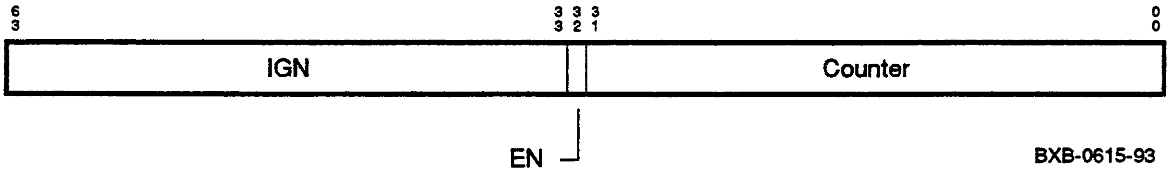
Write Format:

BXB-0614-93

CC_CTL—Cycle Counter Control Register

Index	Abox 17
Access	W

The CC_CTL IPR is used to write to the CC IPR. The HW_MTPR Rn, CC_CTL instruction writes the CC<31:0> with the value held in Rn <31:0>; bits <63:32> bits are not changed. The CC<3:0> must be written with zero. If Rn<32> is set, then the counter is enabled; otherwise, the counter is disabled.



BIU_CTL—BIU Control Register

Index	Abox 18
Access	W

The BIU_CTL IPR is a write-only register that controls the operating parameters of the BIU interface and the B-cache. PALcode writes to this register at initialization and keeps an image of the register which appears in error log entries and is readable by the user.

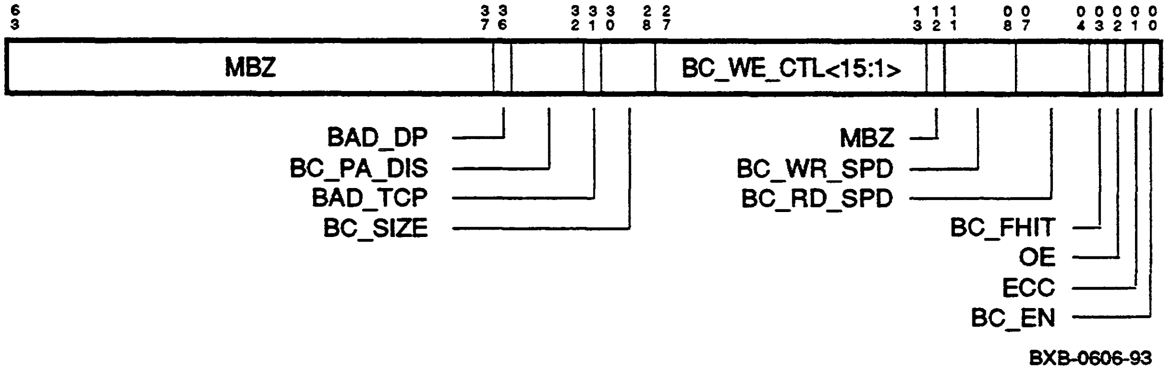


Table 4-26 BIU_CTL IPR Bit Definitions

Name	Bit(s)	Type	Function
BAD_DP	<36>	W, 0	**Bad Data Parity.** When set, causes the DECchip 21064 to invert the value placed on bits <0, 7, 14, 21> of the check_h [27:0] field during off-chip writes. This produces bad parity when the DECchip 21064 is in parity mode and bad check bit codes when in ECC mode.
BC_PA_DIS	<35:32>	W, 0	**B-Cache Physical Address Disable.** This 4-bit field is used to prevent the CPU chip from using the B-cache to service reads and writes based upon the quadrant of physical address space that they reference. The correspondence between this bit field and the physical

Table 4-26 BIU_CTL IPR Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td></td><td></td><td></td><td>address space is as follows:
<table>
<tr><th>BIU_CTL Bits</th><th>Physical Address</th></tr>
<tr><td>&lt;35&gt;</td><td>PA &lt;33:32&gt; = 3</td></tr>
<tr><td>&lt;34&gt;</td><td>PA &lt;33:32&gt; = 2</td></tr>
<tr><td>&lt;33&gt;</td><td>PA &lt;33:32&gt; = 1</td></tr>
<tr><td>&lt;32&gt;</td><td>PA &lt;33:32&gt; = 0</td></tr>
</table>
When a read or write reference is presented to the BIU, the values of BC_PA_DIS, BC_EN, and the physical address bits <33:32> determine whether an attempt is to be made to use the B-cache to satisfy the reference. If the B-cache is not to be used for a given reference, the BIU does not probe the tag store and makes the appropriate system request immediately. The value of BC_PA_DIS has no impact on which portions of the physical address space may be cached in the P-cache. System components control this by way of the dRAck_h field of the EDAL interface.</td></tr>
<tr><td>BAD_TCP</td><td>&lt;31&gt;</td><td>W, 0</td><td><b>Bad Tag Control Parity.</b> When set, causes the DECchip 21064 to write bad parity into the tag control RAM whenever DECchip 21064 does a fast B-cache write.</td></tr>
<tr><td>BC_SIZE</td><td>&lt;30:28&gt;</td><td>W, 0</td><td><b>B-Cache Size.</b> This field is used to indicate the size of the B-cache as follows:
<table>
<tr><th>BC_SIZE</th><th>Size of B-Cache</th></tr>
<tr><td>000</td><td>128 Kbytes</td></tr>
<tr><td>001</td><td>256 Kbytes</td></tr>
<tr><td>010</td><td>512 Kbytes</td></tr>
<tr><td>011</td><td>1 Mbyte</td></tr>
<tr><td>100</td><td>2 Mbytes</td></tr>
<tr><td>101</td><td>4 Mbytes</td></tr>
<tr><td>110</td><td>8 Mbytes</td></tr>
</table></td></tr>
<tr><td>BC_WE_CTL</td><td>&lt;27:13&gt;</td><td>W, 0</td><td><b>B-Cache Write Enable Control.</b> This field is used to control the timing of the write enable and chip enable signals during writes into the data and tag control RAMs. It consists of 15 bits, where each bit determines the value placed on the write enable and chip enable signals during a given CPU cycle of the RAM write access. When a given bit of the BC_WE_CTL is set, the write enable and chip enable signals are asserted during the corresponding CPU cycle of the RAM access. BIU_CTL<13> corresponds to the second cycle of the write access,</td></tr>
</table>

Table 4-26 BIU_CTL IPR Bit Definitions (Continued)

Name	Bit(s)	Type	Function
			BIU_CTL<14> to the third CPU cycle, and so on. The write enable signals will never be asserted in the first CPU cycle of a RAM write access. Unused bits in this field must be written with zeros.
BC_WR_SPD	<11:8>	W, 0	**B-Cache Write Speed.** Indicates to the BIU the write cycle time of the RAMs used to implement the off-chip B-cache, measured in CPU cycles. It should be written with a value equal to one less than the write cycle time of the B-cache RAMs. The access times for writes must be in the range of 16 to 2 CPU cycles, which means that the values for the BC_RD_SPD field are in the range of 15 to 1.
BC_RD_SPD	<7:4>	W, 0	**B-Cache Read Speed.** Indicates to the BIU the read access time of the RAMs used to implement the off-chip B-cache, measured in CPU cycles. It should be written with a value equal to one less than the read access time of the B-cache RAMs. The access times for reads must be in the range of 16 to 4 CPU cycles, which means that the values for the BC_RD_SPD field are in the range of 15 to 3.
BC_FHIT	<3>	W, 0	**B-Cache Force Hit.** When this bit is set and the BC_EN bit is also set, all EDAL interface Read_Block and Write_Block transactions are forced to hit in the B-cache. Tag and tag control parity are ignored. The BC_EN takes precedence over BC_FHIT. When BC_EN is cleared and BC_FHIT is set, no tag probes occur and external requests are directed to the cReq_h pins. *NOTE: The BC_PA_DIS field takes precedence over BC_FHIT.*
OE	<2>	W, 0	**Output Enable.** When set, the DECchip 21064 does not assert its chip enable signals during RAM write cycles, thus allowing the corresponding pins to be connected to the output enable pins of the cache RAMs.
ECC	<1>	W, 0	**Error Checking and Correction.** When set, the DECchip 21064 generates/expects ECC on the check_h pins. When cleared, the DECchip 21064 generates/expects parity on four of the check_h signals.
BC_EN	<0>	W, 0	**B-Cache Enable.** When cleared, the B-cache is disabled. When the B-cache is disabled, the BIU does not probe the B-cache tag store for read/write references; it launches a request on cReq_h immediately.

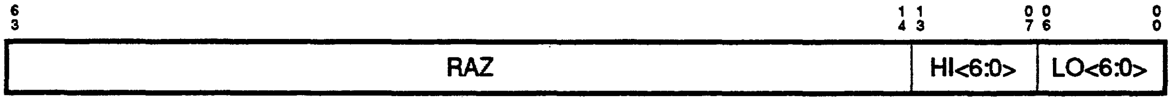
FILL_SYND—Fill Syndrome Register

Index Abox 19

Access R

The FILL_SYND IPR stores the syndrome bits. If the DECchip 21064 is in ECC mode and an ECC error is recognized during a P-cache fill operation, the syndrome bits associated with the bad quadword are locked in the FILL_SYND IPR. A syndrome value of zero means that no errors were found in the associated longword. The FILL_SYND IPR is unlocked when the FILL_ADDR IPR is read. Table 4-28 lists the syndromes associated with correctable single-bit errors.

If the processor is in parity mode and a parity error is recognized during a P-cache fill operation, the FILL_SYND IPR indicates which of the longwords in the quadword got bad parity.



BXB-0609-93

Table 4-27 FILL_SYND IPR Bit Definitions

Name	Bit(s)	Type	Function
HI<6:0>	<13:7>	R	**High <6:0>.** Contains the syndrome associated with the upper longword of the quadword. If the processor is operating in parity mode, bit <0> (FILL_SYND<7>) of this field is set to indicate that the upper longword was corrupted. Bits <6:1> (FILL_SYND<13:8>) read as zeros in parity mode.
LO<6:0>	<6:0>	R	**Low <6:0>.** Contains the syndrome associated with the lower longword of the quadword. If the processor is operating in parity mode, bit <0> (FILL_SYND<0>) of this field is set to indicate that the lower longword was corrupted. Bits <6:1> (FILL_SYND<6:1>) read as zeros in parity mode.

Table 4-28 Syndromes for Single-Bit Errors

Data Bit	Syndrome (Hex)	Check Bit	Syndrome (Hex)
<0>	4F	0	01
<1>	4A	1	02
<2>	52	2	04
<3>	54	3	08
<4>	57	4	10
<5>	58	5	20
<6>	5B	6	40
<7>	5D		
<8>	23		
<9>	25		
<10>	26		
<11>	29		
<12>	2A		
<13>	2C		
<14>	31		
<15>	34		
<16>	0E		
<17>	0B		
<18>	13		
<19>	15		
<20>	16		
<21>	19		
<22>	1A		
<23>	1C		
<24>	62		
<25>	64		
<26>	67		
<27>	68		
<28>	6B		
<29>	6D		
<30>	70		
<31>	75		

BC_TAG—B-Cache Tag Register

Index Abox 20
Access R

The BC_TAG IPR is loaded with the results of every B-cache tag probe, unless locked. When a tag, tag control parity, or primary fill data error (parity or ECC) occurs, BC_TAG is locked against further updates. PALcode may read the LSB of this register by using the HW_MFPR instruction. Each time an HW_MFPR from BC_TAG completes, the contents of BC_TAG are shifted one bit position to the right, so that the entire register can be read using a sequence of HW_MFPRs. PALcode can unlock the BC_TAG with an HW_MTPR to BC_TAG. Successive HW_MFPRs from the BC_TAG must be separated by at least one null cycle.

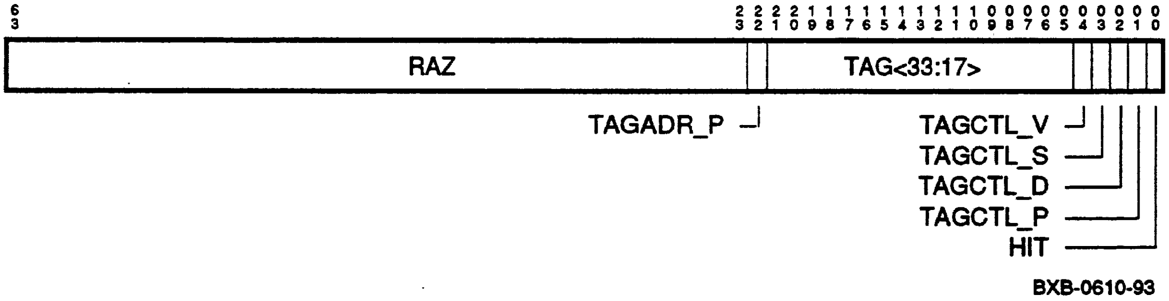


Table 4-29 BC_TAG IPR Bit Definitions

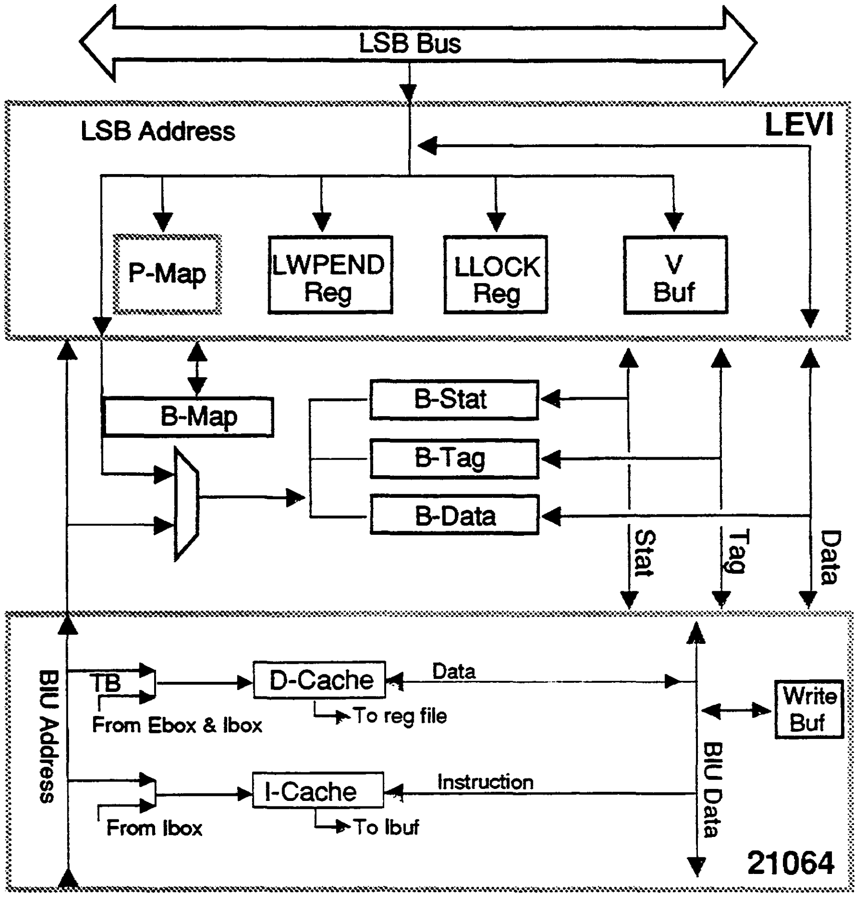
Name	Bit(s)	Type	Function
TAGADR_P	<22>	R	**Tag Address Parity.** Reflects the state of the tagAdrP_h signal of the DECchip 21064 when a tag, tag control, or data parity error occurs.
TAG<33:17>	<21:5>	R	**Tag.** Contains the tag that is being probed currently. *NOTE: Unused bits in the TAG field are always clear, based on the size of the B-cache, as determined by BIU_CTL<BC_SIZE>.*
TAGCTL_V	<4>	R	**Tag Control Valid.** Reflects the state of the tagCtlV_h signal of the DECchip 21064 when a tag, tag control, or data parity error occurs.
TAGCTL_S	<3>	R	**Tag Control Shared.** Reflects the state of the tagCtlS_h signal of the DECchip 21064 when a tag, tag control, or data parity error occurs.
TAGCTL_D	<2>	R	**Tag Control Dirty.** Reflects the state of the tagCtlD_h signal of the DECchip 21064 when a tag, tag control, or data parity error occurs.
TAGCTL_P	<1>	R	**Tag Control Parity.** Reflects the state of the tagCtlP_h signal of the DECchip 21064 when a tag, tag control, or data parity error occurs.
HIT	<0>	R	**Hit.** When set, indicates that there was a tag match when a tag, tag control, or data parity error occurred.

Chapter 5

Cache Memory

The KN7AA CPU module features a two-level cache memory. The first level is implemented on the DECchip 21064 and is referred to as the primary cache (P-cache). The second level resides on the module, external to the DECchip 21064, and is referred to as the backup cache (B-cache). Both caches are accessed with physical addresses. Memory access is performed hierarchically. Instruction and data are first sought from the P-cache, then the B-cache, and finally from memory/another CPU. Figure 5-1 shows the KN7AA CPU module cache organization.

Figure 5-1 KN7AA CPU Module Cache Organization



BXB-0210-93

5.1 P-Cache

The P-cache consists of an 8-Kbyte instruction cache (I-cache) and an 8-Kbyte write-through data cache (D-cache). The I-cache and the D-cache are physically addressed, direct-mapped caches with 32-byte blocks. The I-cache is used to service requests from the Ibox. The D-cache is used to service requests from the DECchip 21064 load/store unit.

The P-cache is a subset of the B-cache at all times.

5.2 B-Cache

The B-cache is implemented in three RAM structures: B-tag, B-data, and B-stat, located between the DECchip 21064 and the LSB interface.

The B-cache stores 4 Mbytes of data. It is organized as direct-mapped, with a block (line) size of 64 bytes to match the LSB bus. For each block, the following information is stored:

- Tag: Consists of bits <33:22> of the physical address
- Tag parity bit: Reflects even parity over the field
- Valid bit (V): Indicates whether the line can be considered
- Shared bit (S): Indicates whether this line might be resident in another cache in the system
- Dirty bit (D): Indicates whether the line has been written to by this CPU and has more recent data than memory.
- Status parity bit: Reflects even parity over the V, S, and D bits.

The B-cache organization groups the status bits in a single 64K X 4 RAM (B-stat) and allows these bits to be updated without changing the value of the tag. This in turn allows the CPU to set the Dirty bit on write hits to nonshared blocks. In general, the tag field is only loaded by the LSB interface, and the status and data stores are loaded by both the processor and the LSB interface.

5.3 B-Cache States

The B-cache state is defined by the three status bits: Valid, Shared, and Dirty. Table 5-1 shows the legal combinations of the status bits.

From the perspective of the DECchip 21064, a tag probe for a read is successful if the tag matches the address and the V bit is set. A tag probe for a write is successful if the tag matches the address, the V bit is set, and the S bit is clear.

Table 5-1 B-Cache States

B-Stat V	 S	 D	State of Cache Line Assuming Tag Match
0	X	X	Valid miss.
1	0	0	Valid for read or write. This cache line contains the only cached copy of the block. The copy in memory is identical to this block.
1	0	1	Valid for read or write. This cache line contains the only cached copy of the block. The contents of the block have been modified more recently than the copy in memory.
1	1	0	Valid for read or write but writes must be broadcast on the bus. This cache line may also be present in the cache of another CPU. The copy in memory is identical to this block.
1	1	1	Valid for read or write but writes must be broadcast on the bus. This cache line may also be present in the cache of another CPU. The contents of the block have been modified more recently than the copy in memory.

5.4 B-Cache State Changes

The state of any given cache line in the B-cache is affected by both processor actions and actions of other nodes on the LSB bus.

Table 5-2 shows what effect processor actions have on the state of a given B-cache line and the resulting/required bus traffic. Table 5-3 shows what effect bus actions have on the state of a given B-cache line, and the resulting/required module action. In these tables, *Match* means that the tag stored at the index matches the supplied address and the <V> bit is set for the index. *Dirty* means that the <D> and <V> bits are set for the index. *Invalid* means that the <V> bit is not set.

LSB writes always clean (make non-dirty) the cache line in both the initiating node and all nodes that choose to take the update. They also update the appropriate location in main memory.

The KN7AA CPU module decides whether to take an update or not as a function of the state of the P-cache backmap (P-map, Section 5.6.1). If the LSB interface determines that the block referenced by an LSB write command is resident in the P-cache, the relevant block is updated in the B-cache with the LSB write data and also invalidated in the P-cache. If the LSB interface determines that the block referenced by an LSB write command is not resident in the P-cache (therefore not "interesting"), but is resident in the B-cache, it invalidates the relevant block in the B-cache.

Table 5-2 Effect of Processor Action on B-Cache Line

Processor Request	Tag Probe Result[1]	Action on LSB	LSB Response	Next Cache State
Read	*Invalid*	Read	$\overline{\textit{Shared}}$	$\overline{\textit{Shared}}$, $\overline{\textit{Dirty}}$
Read	*Invalid*	Read	*Shared*	*Shared*, $\overline{\textit{Dirty}}$
Write	*Invalid*	Read	$\overline{\textit{Shared}}$	$\overline{\textit{Shared}}$, *Dirty*
Write	*Invalid*	Read, Write	*Shared*	*Shared*, $\overline{\textit{Dirty}}$
Read	$\overline{\textit{Match}}$ AND $\overline{\textit{Dirty}}$	Read	$\overline{\textit{Shared}}$	$\overline{\textit{Shared}}$, $\overline{\textit{Dirty}}$
Read	$\overline{\textit{Match}}$ AND $\overline{\textit{Dirty}}$	Read	*Shared*	*Shared*, $\overline{\textit{Dirty}}$
Write	$\overline{\textit{Match}}$ AND $\overline{\textit{Dirty}}$	Read	$\overline{\textit{Shared}}$	$\overline{\textit{Shared}}$, *Dirty*
Write	$\overline{\textit{Match}}$ AND $\overline{\textit{Dirty}}$	Read, Write	*Shared*	*Shared*, $\overline{\textit{Dirty}}$
Read	$\overline{\textit{Match}}$ AND *Dirty*	Read, Wr-Victim	$\overline{\textit{Shared}}$	$\overline{\textit{Shared}}$, $\overline{\textit{Dirty}}$
Read	$\overline{\textit{Match}}$ AND *Dirty*	Read, Wr-Victim	*Shared*	*Shared*, $\overline{\textit{Dirty}}$
Write	$\overline{\textit{Match}}$ AND *Dirty*	Read, Wr-Victim	$\overline{\textit{Shared}}$	$\overline{\textit{Shared}}$, *Dirty*
Write	$\overline{\textit{Match}}$ AND *Dirty*	Read, Write, Wr-Victim	*Shared*	*Shared*, $\overline{\textit{Dirty}}$
Read	*Match*	None	None	No change
Write	*Match* AND $\overline{\textit{Shared}}$	None	None	$\overline{\textit{Shared}}$, *Dirty*
Write	*Match* AND *Shared*	Write	$\overline{\textit{Shared}}$	$\overline{\textit{Shared}}$, $\overline{\textit{Dirty}}$
Write	*Match* AND *Shared*	Write	*Shared*	*Shared*, $\overline{\textit{Dirty}}$

[1] An overscore on a cache block status bit indicates the complement of the state. For example, $\overline{\text{Shared}}$ = Not Shared.

For diagnostic and system performance measurement purposes, the KN7AA module implements two alternate behavior modes in response to LSB writes. LMODE<WMODE> allows selection of either the normal mode as described, using the P-map, or forces all LSB writes to cause a B-cache invalidate/update.

Table 5-3 Effect of LSB Bus Action on B-Cache Line

LSB Operation	Tag Probe Result[1]	Module Response	Next Cache State	Comment
Read	$\overline{\textit{Match}}$ OR *Invalid*	$\overline{\textit{Shared}}$, $\overline{\textit{Dirty}}$	No change	
Write	$\overline{\textit{Match}}$ OR *Invalid*	$\overline{\textit{Shared}}$, $\overline{\textit{Dirty}}$	No change	
Read	*Match* AND $\overline{\textit{Dirty}}$	*Shared,* $\overline{\textit{Dirty}}$	*Shared,* $\overline{\textit{Dirty}}$	
Read	*Match* AND *Dirty*	*Shared, Dirty*	*Shared, Dirty*	This module must supply the data.
Write	*Match* AND line is interesting	*Shared,* $\overline{\textit{Dirty}}$	*Shared,* $\overline{\textit{Dirty}}$	This module takes the update.
Write	*Match* AND line is uninteresting	$\overline{\textit{Shared}}$, $\overline{\textit{Dirty}}$	*Invalid*	This module takes the invalidate.

[1] An overscore on a cache block status bit indicates the complement of the state. For example, $\overline{\text{Shared}}$ = Not Shared.

The KN7AA CPU module also compares incoming LSB addresses to those found in the LLOCK register, LVICT register, and LWPEND register (see Chapter 6). The behavior of the KN7AA CPU module in these cases is shown in Table 5-4.

Table 5-4 KN7AA CPU Module Response to Incoming Addresses

LSB Operation	Address Register Matched	Module Response	Action
Read	LLOCK register	*Shared*	No action
Write	LLOCK register	$\overline{\textit{Dirty}}$	Clear LLOCK<31>
Read	LVICT register	*Shared, Dirty*	Supply data from victim buffer
Write	LVICT register	$\overline{\textit{Shared}}$, $\overline{\textit{Dirty}}$	Invalidate victim buffer; remove bus request
Read	LWPEND register	*Shared*	No action
Write	LWPEND register	*Shared,* $\overline{\textit{Dirty}}$	Accept update to B-cache

[1] An overscore on a cache block status bit indicates the complement of the state. For example, $\overline{\text{Shared}}$ = Not Shared.

5.5 Write Policy

The KN7AA module performs LSB write operations as follows:

- **Victims**
  If a given cache line is valid and dirty and the tag does not match the address for the given processor request, the line must be written back to memory. To enhance performance, this victim is written back to memory after the refill. The victim data must be removed from the B-cache data store and held in a victim buffer (see Section 5.7) for later transmission on the LSB bus. While a block is in a victim buffer, the KN7AA must respond to all reads and writes that reference the block (see Table 5-4).
- **Shared Blocks**
  If the response to a tag probe for a processor write is shared, the write must be broadcast on the LSB bus.

5.6 Cache Backmaps

The KN7AA CPU module implements two backmaps (or duplicate tag stores) that keep track of the contents of the P-cache and the B-cache. They are referred to as P-map and B-map. The backmaps are cycled with every bus transaction to allow the KN7AA CPU module to properly respond to a given bus command/address.

5.6.1 P-Map

The P-map is located in the LEVI gate arrays and consists of four identical structures, each 64 entries deep. Each P-map entry contains a value that is equal to the difference between the B-cache tag (address bits <33:22>) and the P-cache tag (address bits <31:12>), valid bit, and an even parity bit. Thus, the P-map is 12 bits wide: Address bits <21:12>, V, and P. The P-map is loaded by the DECchip 21064 during B-cache D-stream read hits and by the LSB interface during B-cache D-stream read misses. The LSB interface control can read the P-map whenever an LSB write hits in the B-map.

The KN7AA CPU module enforces inclusion, which ensures that the valid contents of the P-cache are always a subset of the valid contents of the B-cache. Therefore, the KN7AA CPU module must invalidate P-cache lines whenever the given block becomes invalid in the B-cache. This occurs on refills (either a dirty victim or a nonshared victim) and on updates.

When an update occurs on the LSB bus, and the given address yields a tag match and the entry is valid in the P-map, the B-cache takes the update and the CPU module invalidates the corresponding entry in the P-cache.

5.6.2 B-Map

The B-map is located on the module and is a structure 64K entries deep. Each entry consists of the B-cache tag (address bits <33:22>), valid bit, and even parity bit. The B-map is written by the LSB interface at the same time that the B-cache tag is written (within the context of B-cache manipulation, due to either processor action or bus action). The B-map is read on every LSB bus command/address cycle. The contents of the B-map inform the KN7AA CPU control logic when to request the B-cache to form an appropriate bus response. The processor does not read or write the B-map. The LSB interface only reads and writes the B-map in the LSB time domain.

5.7 Victim Buffer

The KN7AA CPU module implements a victim buffer to hold the contents of a victimized block in the B-cache. A victim block is a B-cache line that is valid and dirty but has a tag mismatch for a processor request. The processor tag probe yields a miss and the appropriate block is fetched from memory. However, the block in the B-cache at this index must be written back to memory since it is dirty. The KN7AA CPU module posts the miss refill to the bus before actually performing the victim write.

A single victim block and victim address pair is stored in the LEVI chips for later transmission on the bus. While the victim buffer contains a valid victim, the KN7AA CPU module treats this block like a second set in the B-cache, compares all bus addresses to the victim address, and responds to bus reads and writes as required by the bus protocol (see Table 5-4).

The KN7AA CPU module has a single victim buffer. It therefore does not process a second B-cache miss before writing the victim block to memory.

5.8 B-Cache Operating Modes

The backup cache has two modes of operation:

- B-cache on
- B-cache force hit

The operating modes are controlled by two bits in the BIU_CTL register: BC_ENB (bit <0>) and BC_FHIT (bit <3>).

Table 5-5 shows how the operating mode of the B-cache is selected.

Table 5-5 Selection of the B-Cache Operating Mode

BIU_CTL<3>	BIU_CTL<0>	Operating Mode
0	1	B-cache on
1	1	Force hit

The On state is the normal operating mode of the B-cache. It is selected by setting BIU_CTL<0> and clearing BIU_CTL<3>.

NOTE: In reality, the B-cache is never off. If BIU_CTL<0> is cleared, the processor bypasses the B-cache and goes directly to the LSB. This function should be used only by diagnostics.

The B-cache force hit mode is selected by setting BC_FHIT (BIU_CTL<3>) when the B-cache is enabled. When this bit is set, all memory space reads and writes to the B-cache, both I-stream and D-stream, are forced to hit. The tag store state is not changed. The data RAMs are accessed as if the tag store access produced a dirty-valid hit. In a multiprocessor environment, the B-cache must be flushed of all dirty blocks before force hit mode is selected.

Force hit mode is intended to be used only for testing and initialization. Tag store parity and data RAM ECC errors are detected in this mode.

5.9 Cache Initialization

On power-up or following a reset, the processor microcode and the console firmware initialize the P-cache and the B-cache. In the initialized state, the P-cache is enabled for I-stream and D-stream operations, and the B-cache is on.

CAUTION: The cache subsystem is initialized to a determined state. Software must never turn the B-cache off once the system is up and running. Turning the B-cache off during normal operation places the system in an UNDETERMINED state.

Chapter 6

LSB Bus Interface

The CPU module connects to the LSB bus through LEVI, the LSB interface, which is implemented in two gate array chips, LEVI-A and LEVI-B. LEVI controls all the tags, maps, and data RAMs on the CPU module. It contains the P-map, which maps the processor P-cache.

LEVI performs the following major tasks:

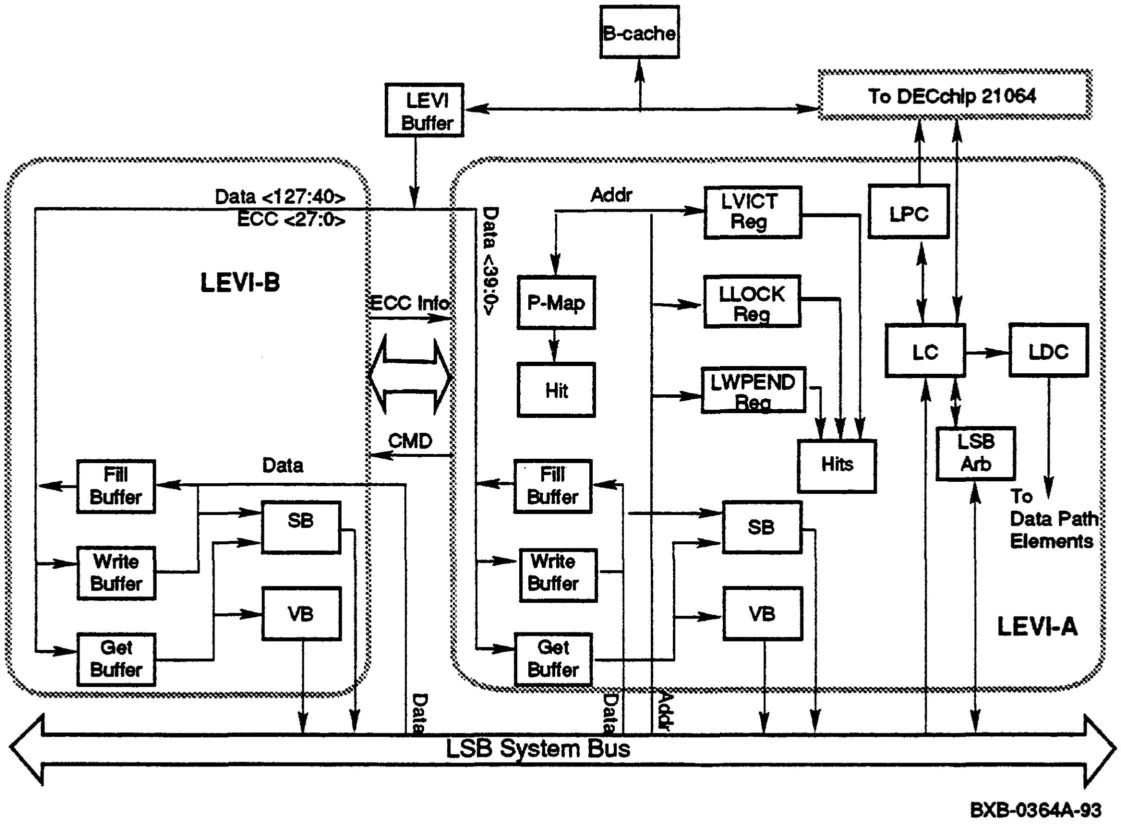
- Translates CPU, memory, and I/O space references to the appropriate LSB transactions.
- Supports control of writebacks to memory and cache fills from memory in reponse to processor actions.
- Supports control of cache invalidates, cache updates, and cache block transfers to the LSB bus in response to LSB actions.
- Initiates reads and writes to the CPU node private address space (the Gbus on the CPU module).
- Supports LSB required interrupt logic.
- Implements all LSB required registers.

This chapter discusses the role of LEVI in transactions between the CPU module and other modules on the LSB bus. Sections include:

- LEVI Address Path
- LEVI Data Path
- LEVI Controllers
- Interfacing Rules
- Address Space Mapping
- LEVI Transactions

Figure 6-1 shows a block diagram of the LEVI chips.

Figure 6-1 LEVI Block Diagram



6.1 LEVI Address Path

The LEVI address path (see Figure 6-1) is implemented in the LEVI-A chip. It consists of the following major elements:

- **P-Map**
  The P-map consists of four 64 X 16 dual-ported RAMs maintained exclusively by LEVI. Each entry in the P-map represents a P-cache block in the processor. LEVI writes to the P-map during processor read hits to the B-cache. One use of the P-map is deciding whether to update or invalidate a B-cache block during LSB writes from another node. If the LSB write hits in the P-map, the update is taken; otherwise LEVI invalidates the B-cache. Another use is to invalidate P-cache blocks that are being displaced by B-cache fills.

- **LVICT Register**
  LEVI keeps the address of the last victimized B-cache block and a valid bit in the LVICT register. Once the LSB Victim Write takes place, the Valid bit is cleared. Should another LSB (non-Victim) write match the address in the LVICT register, LEVI invalidates its own LVICT register (see Table 5-4).

- **LLOCK Register**
  The address is latched and LLOCK<31> is set when the processor issues an LDxL instruction. LLOCK<31> is cleared after a successful STxC instruction. LSB reads that hit in the LLOCK register cause LEVI to respond "Shared" so that all subsequent writes to the address

are visible on the LSB bus. LSB writes that hit in the LLOCK register cause LEVI to clear the LLOCK<31> (see Table 5-4).

- **LWPEND Register**
This register contains the address of the pending write and a valid bit. If an LSB Write hits the LWPEND register, LEVI-A takes the update even if the write missed in the P-map to assure that the write about to be issued has the latest data (see Table 5-4).

6.2 LEVI Data Path

The LEVI data path, like the LSB bus and CPU module data paths, is 156 bits wide: 128 bits of data and 28 bits of ECC (7 bits for each longword). Note that LEVI treats the data and ECC bits identically since there is no ECC correction between the B-cache and the LSB bus.

An array of buffers in the LEVI data path serve to store data and synchronize data movement within LEVI. The buffers are implemented in both LEVI chips, as shown in Figure 6-1. The main buffer elements on the LEVI data path are the following:

- **Fill Buffer**
The fill buffer works with the LEVI buffer on the module to receive, and possibly hold, four octawords of LSB data headed for the B-cache. The data pipeline shifts from the gate array time domain to the clock-forwarded module time domain in the fill buffer. The fill buffer also merges write buffer data with LSB data following B-cache write misses.

- **Get Buffer**
The get buffer also works with the LEVI buffer; it captures B-cache blocks headed for the LSB bus. The data pipeline shifts back from the module to the gate array time domain in the get buffer.

- **Write Buffer**
The write buffer captures two octawords of write data from the processor in three situations:

  B-cache write misses
  B-cache write hits to shared blocks
  CSR writes

  The write buffer also receives a write data mask (LEVI-A gets four of eight bits; LEVI-B gets all eight bits) and ADDR<5> from the processor. The mask and address bits indicate which longwords are to be merged with B-cache data.

- **Stall Buffer**
The stall buffer holds four octawords of B-cache data for broadcast onto the LSB bus. It also merges write buffer data with B-cache data during writes to shared blocks and processor CSR writes.

- **Victim Buffer**
The victim buffer holds B-cache blocks victimized by cache fills. The buffer holds only a single cache block so transactions that cause other victims are held off until the current victim reaches the LSB bus.

6.3 LEVI Controllers

The control functions on the LEVI transactions are implemented in three controllers in the LEVI chips:

- LEVI processor controller (LPC)
- LEVI data controller (LDC)
- LSB controller (LC)

6.3.1 LEVI Processor Controller

The LPC provides the control interface between the processor and LEVI. It fields requests from the processor and initiates LEVI responses. Major functions include:

- **LoadLock/StoreCond**
  The LPC first probes the cache; misses generate requests to the LSB controller for LSB transactions.
- **Gbus Read**
  The LPC asks the LSB controller for a CSR read and does extra handshaking on the Gbus. It appears on the LSB as a private command (not a CSR read).
- **Gbus Write**
  The LPC controls the write to the LEVI write buffer, then handshakes with the Gbus and acknowledges the processor. No LSB transaction is requested.
- **Processor Read Fill**
  After a processor read miss and after LEVI has received the missed data from the LSB bus, the LPC loads the processor with the two octawords it has waited for. The LEVI data controller (LDC) briefly interrupts the B-cache fill after the first octaword write to allow the processor to load two octawords from the LEVI buffer. The LDC then completes the final three octaword writes.
- **Processor Write Data**
  The LPC controls the processor writes to the write buffer on LEVI when the processor cannot write directly to the B-cache.
- **P-Cache Invalidates**
  Whenever LEVI invalidates P-map entries, the LPC invalidates the corresponding P-cache entries in the processor.

The LPC runs in the processor time domain.

6.3.2 LEVI Data Controller

The LDC directs data traffic moving between the LSB and the B-cache based on requests from the LC. Each transaction described below moves one B-cache block. The LDC is involved in the following transactions:

- **GetRAM**
  GetRAM moves one B-cache block to the LSB by way of the stall buffer. The LSB controller (LC) requests this transfer when another node has issued a read to a block and the local (and only valid) copy is dirty.

- **GetWBRAM**
  GetWBRAM is used on processor writes to shared B-cache blocks and processor CSR writes. The fetched B-cache block is conditionally merged with the contents of the write buffer (based on the values of the write data mask and ADDR<5>) before being driven onto the LSB by way of the stall buffer.

- **GetVic**
  GetVic is used to route B-cache blocks that are victimized by B-cache fills to the victim buffer. Note that blocks in the victim buffer must await an LSB slot; blocks in the stall buffer have already had their LSB slots allocated (by the LC).

- **FillRAM**
  FillRAM moves one block directly from the LSB to the B-cache. The LC requests this to take an update to a shared block or to complete the first read of a processor write miss to a shared B-cache block.

- **FillProcRAM**
  FillProcRAM is requested following processor read misses. The LDC moves data from the LSB to the fill buffer and the LEVI buffer on the module. The data pipeline is frozen briefly after the first octaword write to the B-cache to allow the LPC to load the first two octawords into the processor (in its own time domain) by way of the LEVI buffer. The LPC then releases the processor but retains control of the B-cache so that the LDC can write the remaining three octawords to the B-cache.

- **FillProc**
  FillProc services processor CSR reads. This transaction is identical to FillProcRAM discussed above except that writes to the B-cache are suppressed. (CSR data is not cached.)

- **FillWBRAM**
  FillWBRAM merges processor write data in the Write buffer with the incoming LSB data and writes the result into the B-cache. Merging is based on the values of the write data mask and ADDR<5>. The LC requests this transaction following processor write misses to blocks that are not shared.

The LDC uses clock forwarding on the CPU module for data transfers between LEVI and the B-cache.

6.3.3 LSB Controller

The LC is the central controller of the LEVI chipset. It receives requests from the LPC and issues requests to the LPC, LDC, and the LSB arbiter. The LC responds to both processor-initiated and LSB-initiated transactions. Specifically, the LC performs the following functions:

- Controls the address path and LEVI access to the B-map, B-stat, and B-tag RAMs on the CPU module.

- Schedules all LEVI and CPU module operations except B-cache hits and Gbus transactions. Requests from the LC to the LPC and LDC move data around the module, the gate arrays, and the LSB bus.

- Asserts the LSB address and control signals (CNF, ERR) according to LSB protocol. LSB SHARED and DIRTY are asserted based on the results of B-tag and B-stat lookup. LSB STALL is asserted when required by internal conflicts.

- The LC also controls B-cache access from the processor or LEVI with the LSynch signal (Section 6.4.1).

The LC schedules transfers of data between the LEVI and the CPU module during dedicated LSB cycles.

6.4 Interfacing Rules

Logic on the CPU module synchronizes dual-ported accesses to the B-cache and the P-map, since these components are accessed by both the processor and LEVI. LSB arbitration rules govern node accesses to the LSB bus.

All cache data is longword ECC protected (seven bits per longword). LEVI does look-aside ECC error detection but no ECC error correction.

The LEVI chips calculate ECC for each longword and compare it against the received ECC. Any difference between calculated and received ECC indicates an error, which is signaled to the system. The ECC for longword 0 and a partial ECC syndrome for longword 1 are passed each cycle from LEVI-B to LEVI-A.

6.4.1 Dual-Ported Access Synchronization

Dual-ported B-cache and P-map accesses are synchronized with the LSynch semaphore. LSynch is also used to synchronize access to the CPU module data path during Gbus references.

Whenever LSynch is deasserted (default state), the processor can read or write the B-stat, B-tag, and B-data RAMs directly.

NOTE: The B-map RAMs are never accessed by the processor.

During LoadLock and StoreCond requests, whenever LSynch is deasserted, the LPC can read or write the B-stat, B-tag, and B-data RAMs directly.

During RBlock and WBlock requests to Gbus addresses, whenever LSynch is deasserted, the LPC can transfer data between the processor and the Gbus buffer on the CPU module. LEVI has priority to assert LSynch and access the B-cache to service LSB transactions, since the LSB is nonpended. LEVI also accesses the B-cache to complete processor transactions that miss the B-cache. When the LC asserts LSynch, the processor and the LPC suspend B-stat/B-tag references (tag probes), P-map updates (PMapWE), and B-data references within a fixed number of LSB cycles. The LEVI is then free to access any resource within the CPU module until it deasserts LSynch at the completion of the LSB-related access.

6.4.2 LSB Arbitration

LEVI watches all LSB traffic to adhere to the arbitration rules. Specifically, read, write, or victim transactions from any node that reference a common memory bank cannot occur more frequently than once every three transactions (or once every 15 LSB cycles). CSR transactions are also limited in the same manner.

6.5 Address Space Mapping

The LEVI chips define which portion of the address space is cacheable or noncacheable. Cacheable address space is memory space and noncacheable address space is I/O space. The LEVI chips further separate I/O space into LSB bus CSR space and local Gbus space.

The LEVI interface ensures that processor references to memory result in an LSB bus read or write command, while references to I/O space result in an LSB read CSR, write CSR command, or private command.

Table 6-1 gives the encodings of commands that LEVI can send to the LSB bus.

Table 6-1 LSB Command Field Encodings

LSB D<37:35>	Command
000	Read
001	Write
010	Reserved
011	Write Victim
100	Read CSR
101	Write CSR
110	Reserved
111	Private

6.6 LEVI Transactions

As the CPU module's interface to the LSB bus, LEVI responds to transactions initiated from two sources:

- Processor (CPU chip)
- LSB bus (other nodes)

These transactions require that both the processor and LEVI have access to the B-cache on the CPU module and the P-map in LEVI. The dual-ported accesses to these components are synchronized with the LSynch semaphore (Section 6.4.1). The two LEVI chips operate in both the processor and the LSB bus time domains.

6.6.1 Processor-Initiated Transactions

LEVI responds to the following processor requests:

- **Read/Write Hit**
  During a D-stream read hit, LEVI updates its P-map. It takes no other action.
- **Block Read/Write**
  LEVI captures B-stat and B-tag data, arbitrates for the LSB bus, issues the read/write command code on the bus, receives/drives data on the bus, and updates all tags, maps, and B-stat bits.

- **LoadLock/StoreCond**
  LEVI waits for LSynch to deassert, if necessary, then probes the B-tag. On an LDxL (LoadLock) command that hits in the B-cache, LEVI completes the read request and sets LLOCK<31>. If the LDxL is a B-cache miss, LEVI issues an LSB bus read command and sets LLOCK<31>. On an STxC (StoreCond) request from the processor, LEVI checks LLOCK<31>. If this bit is set, (success) and the B-cache tag lookup results in a hit, LEVI immediately completes the write and clears LLOCK<31>. If the tag probe results in a miss, and LLOCK<31> is set, LEVI issues an LSB bus write command. On an STxC, if LLOCK<31> is clear, LEVI returns failed status to the processor.
- **Gbus Read/Write**
  LEVI waits for LSynch to deassert, if necessary. For Gbus reads, LEVI-A arbitrates for the LSB bus, issues a private command, forwards data from the Gbus to the processor by way of the LSB bus. Gbus writes slip through LEVI to the Gbus without an LSB transaction.

The processor can be engaged in only one external operation at a time. This means that once the processor makes a transaction request to LEVI, it remains idle until released by LEVI.

6.6.2 LSB-Initiated Transactions

LEVI responds to transactions initiated by other nodes on the LSB. These transactions include:

- **Read**
  LEVI checks each read address against the B-map. If there is a match, LEVI then checks the B-stat RAMs. It returns B-cache data if the Dirty bit is set. LEVI returns a victimized block, which is sitting in the victim buffer, if the block's address matches the read address.
- **Write**
  When the write address matches that of a valid block in the B-map, LEVI reacts as follows. If the address also hits in the P-map, LEVI takes the update and invalidates the P-cache block in the processor. Otherwise, the B-cache block is simply invalidated. Note that this behavior can be altered with the LMODE register.
- **Victim Write**
  LEVI ignores victim writes from other nodes.
- **CSR Read/Write**
  Only registers in the LSB node space can be read or written from the LSB. Gbus registers cannot be accessed from the LSB. Note that LEVI can also respond to its own processor-generated CSR transactions on the bus.
- **Private**
  Private transactions are used to return Gbus data to the processor, to allow access to the B-tag, B-stat, B-map, and P-map structures directly by the processor, and to resolve STxC boundary conditions. LEVI does not respond to private commands from other modules.

LEVI is pipelined to track up to three interleaved LSB transactions.

6.6.3 Transaction Ordering

The processor controller (LEVI PC, Section 6.3.1) and the LSB controller (LEVI LC, Section 6.3.3) work together to guarantee strict ordering of transactions issued on the LSB. Processor and LEVI actions proceed in stages as shown in Table 6-2.

Table 6-2 Processor-LEVI Actions During Transactions

Processor Action	LEVI Action
P1. The processor issues a request with address A1.	In response to **P1**, LEVI performs the following actions:
	L1. LEVI initiates an LSB transaction with address A1.
P2. The processor can issue a new request with address A2 any time after **L1** completes.	**L2**. If **P1** was a WBlock and **L1** was an LSB Read that received a shared response, LEVI issues an LSB Write with address A1.
	L3. If **L1** was an LSB Read and the B-cache block being displaced had the Dirty bit set, LEVI issues an LSB Write Victim command.
	In response to **P2**, LEVI performs the following action:
	L4. LEVI initiates an LSB transaction with address A2.

Chapter 7

Console Overview

The KN7AA CPU module supports the LSB system console with combined hardware/software elements that control the system at power-up, on reset, or on CPU halts. This chapter describes the console hardware that resides on the CPU module. Sections include:

- CPU Console Hardware
- Console Program Invocation
- Console Registers

The console user interface and commands are discussed in the *Console Reference Manual.*

7.1 CPU Console Hardware

The KN7AA CPU module provides hardware to support the console functions. This hardware includes:

- A serial ROM (read-only memory) for first-level console program storage
- A set of FEPROMs (flash programmable ROMs) for second-level console program storage
- An EEPROM (electrically erasable/programmable ROM) for miscellaneous parameter/log storage
- A set of UARTs (universal asynchronous receivers/transmitters) that allow the console program to communicate serially with one console terminal and the system power supplies
- A watch chip that provides a programmable internal timer and a battery-backed-up time-of-year (TOY) clock for use by operating system software
- A set of parallel I/O ports for functions such as LED status indicators and node identification
- A serial I/O port for manufacturing diagnostic use

The CPU module provides access to ROM, EEPROM, console UARTs, the watch chip, and other functions through the 8-bit Gbus.

All Gbus component registers and memory stores are located in node private space, which means that their addresses are constant and are independent of slot identification. Table 7-1 gives the address ranges allocated to the Gbus components.

Every Gbus memory store byte or register byte is located on a 64-byte, naturally aligned boundary. For example, the first byte of FEPROM storage is located at byte address 3 F000 0000; the second byte is at 3 F000 0040. Also note that a single 128-Kbyte FEPROM consumes 8 Mbytes of address space. This addressing restriction implies that processor code cannot be executed from this address space.

Table 7-1 Gbus Components

Component	Address
Console ROM	3 F000 0000 to 3 F37F FFC0
Console EEPROM	3 F380 0000 to 3 F3FF FFC0
UART registers	3 F400 0000 to 3 F500 00C0
Watch registers	3 F600 0000 to 3 F600 0FC0
Gbus$WHAMI	3 F700 0000
Gbus$LEDs	3 F700 0040
Gbus$PMask	3 F700 0080
Gbus$Intr	3 F700 00C0
Gbus$Halt	3 F700 0100
Gbus$LSBRST	3 F700 0140
Gbus$Misc	3 F700 0180
Gbus$RMode	3 F780 0000
Gbus$LTagRW	3 F780 0100

7.1.1 Serial ROM

After power-up, node reset, or system reset, but before any instructions are executed, the DECchip 21064 automatically loads its internal I-cache through the serial I/O port from an external, 8-Kbyte serial ROM (SROM).

The SROM contains the first level of console/diagnostic/bootstrap code (serial ROM code). This code initializes all programmable features of the DECchip 21064, diagnosing any faults detected along the bootstrap path and bootstrapping code execution out to the second level of console /diagnostic/ bootstrap code (the main console program). The first level bootstrap copies the main console program code from FEPROM storage to the B-cache and transfers control flow to the B-cache. Once the serial ROM is loaded into the B-cache, the same serial I/O port becomes available for use by software as a diagnostic interface.

7.1.2 Serial Port

The DECchip 21064 provides an initialization and diagnostic interface in the form of a serial I/O port. The serial I/O port is a full duplex connection between the CPU chip and a module connector. The port is accessed and controlled through internal processor registers.

The serial I/O port drives a LED indicator, which may flash as data is transmitted over the serial port, but is otherwise available to diagnostic code as a status indicator.

7.1.3 FEPROMs

The console program is stored in a set of 128K X 8 FEPROM chips. This code does not appear in a structure of contiguous locations in the processor's address space. Specifically, each byte of FEPROM storage appears on a 64-byte naturally aligned boundary. This implies that the console program cannot execute directly out of FEPROM, but instead must be copied into a more compact contiguous space in cacheable memory and executed from there. This process of copying the code store and transferring control flow is known as the first-level bootstrap and is performed by the serial ROM code, as explained in Section 7.1.1.

The FEPROMs can be programmed online without assistance from an external programming device. The FEPROMs cannot be patched; they can only be erased and programmed as a whole.

7.1.4 EEPROM

A single 8K X 8 EEPROM is used for miscellaneous parameter and log storage. This store does not appear in a contiguous address space. Each byte of EEPROM storage appears on a 64-byte boundary.

The EEPROM can be written to byte-by-byte online, without assistance from an external programming device.

7.1.5 UARTs

The CPU module has six serial communication lines but uses only three. The communication lines are named and assigned as follows:

- UART0A is connected to the LSB local console terminal line LOC_RX/ LOC_TX (computer room terminal for field service).
- UART1B is connected to the LSB power supply status lines PS_RX and PS_TX.
- UART2A is dedicated to Ctrl/P character detection. Its receive line can tap receive characters off LOC_RX, OP_RX, or RD_RX as selected by the Gbus$PMask register. Its transmit line is unused. UART2A enables IPL 15 interrupts. If no serial lines are selected for console operation (the processor is halt-protected), then all receive characters result in an IPL 15 interrupt. For UART2A to detect Ctrl/P characters, all control settings must be programmed to match the console terminal UART.
- UART0B, UART1A, and UART2B are unused.

The LSB console serial lines are connected to all CPU slots. After power-up or system initialization, the CPU modules arbitrate for use of the common console lines; the winner is allowed to drive them. The default configuration of the serial lines at power-up is as follows:

Baud rate set to 9600
No parity
One stop bit
8-bit characters

One physical component (DUART) implements two UARTs, hence the naming of the UARTs as UART0A, UART0B, and so on, where the number indicates the physical component and the letter indicates the individual UART within the component. Control of these UARTs is accomplished through a set of registers in each UART. These registers are listed in Table 7-2.

7.1.5.1 Ctrl/P Character Detection and Halt Protection

UART2A is dedicated to detecting Ctrl/P characters received from the console terminal.

UART2A intercepts a copy of all UART receive characters from the console terminal line and compares for Ctrl/P. Ctrl/P characters result in an IPL 1F interrupt (halt) posted to the processor (reflected in the Gbus$Halt register). Note that the IPL 1F interrupt is in addition to the IPL 15 interrupt.

7.1.5.2 UART Register Addressing

Each UART in a DUART component is controlled independently through its own set of registers (some registers are shared between two UARTs within a DUART). All UART registers are either read only (for status and data receive) or write only (for control and data transmit). Read registers and write registers share common addresses, that is, reading and writing a single address accesses two separate registers.

For each UART there are two read registers and two write registers that are directly accessible in the processor's address space: RR0, WR0, RR8, and WR8. RR0 and WR0 are the main status and control registers for the UART. RR8 and WR8 are the data receive and transmit registers.

For each UART there are a number of other control and status registers that are indirectly accessible through RR0 and WR0. These registers are accessed by writing the correct index value into WR0 and then reading RR0 or writing WR0. After the second read/write operation occurs, the index value is automatically reset back to zero.

7.1.6 Watch Chip

A watch chip resides on the Gbus and provides a battery-backed-up time-of-year clock and 50 bytes of battery-backed-up RAM. The chip contains a built-in crystal oscillator, an internal timer, and a 10-year lithium battery.

7.2 Console Program Invocation

The DECchip 21064 operates in console mode when the CPU module encounters one of the following conditions:

- System reset through power-up, control panel reset, or reset through the Gbus$LSBRST register

- Module reset performed by setting NRST (LCNR<30>)
- Module halted by setting NHALT (LCNR<29>)
- Ctrl/P character received from the console terminal

7.3 Console Registers

Table 7-2 lists the console registers with their addresses and indicates the components in which they are implemented.

A number of console/diagnostic/interrupt related registers listed in Table 7-2 are referred to with a prefix of Gbus$. These registers provide the following control and status functions:

- Node identification
- LED status indicators
- Interrupt status summaries
- Console terminal selection
- Halt protection
- System reset

This section provides descriptions of individual Gbus registers. The remaining console registers are listed in Table 7-2 for reference only. All Gbus registers are eight bits wide.

Table 7-2 Console Registers

Register	Address	Implementation
UARTxx$WR0[1]	UARTxx_BASE[1]	DUART chip
UARTxx$WR1	Index 0001	DUART chip
UARTxx$WR2	Index 0010	DUART chip
UARTxx$WR3	Index 0011	DUART chip
UARTxx$WR4	Index 0100	DUART chip
UARTxx$WR5	Index 0101	DUART chip
UARTxx$WR6	Index 0110	DUART chip
UARTxx$WR7	Index 0111	DUART chip

[1] **UART Base Addresses:**

xx = 0B; BASE = 3 F400 0000
xx = 0A; BASE = 3 F400 0080
xx = 1B; BASE = 3 F480 0000
xx = 1A; BASE = 3 F480 0080
xx = 2B; BASE = 3 F500 0000
xx = 2A; BASE = 3 F500 0080

Table 7-2 Console Registers (Continued)

Register	Address	Implementation
UARTxx$WR8	UARTxx_BASE+40H	DUART chip
UARTxx$WR9	Index 1001	DUART chip
UARTxx$WR10	Index 1010	DUART chip
UARTxx$WR11	Index 1011	DUART chip
UARTxx$WR12	Index 1100	DUART chip
UARTxx$WR13	Index 1101	DUART chip
UARTxx$WR14	Index 1110	DUART chip
UARTxx$WR15	Index 1111	DUART chip
UARTxx$RR0	UARTxx_BASE	DUART chip
UARTxx$RR1	Index 0001	DUART chip
UARTxx$RR2	Index 0010	DUART chip
UARTxx$RR3	Index 0011	DUART chip
UARTxx$RR8	UARTxx_BASE+40H	DUART chip
UARTxx$RR10	Index 1010	DUART chip
UARTxx$RR13	Index 1101	DUART chip
UARTxx$RR15	Index 1111	DUART chip
Watch$Seconds	3 F600 0000	Watch chip
Watch$Minutes	3 F600 0080	Watch chip
Watch$Hours	3 F600 0100	Watch chip
Watch$Day_of_Month	3 F600 01C0	Watch chip
Watch$Month	3 F600 0200	Watch chip
Watch$Year	3 F600 0240	Watch chip
Watch$CSRA	3 F600 0280	Watch chip
Watch$CSRB	3 F600 02C0	Watch chip
Watch$CSRC	3 F600 0300	Watch chip
Watch$CSRD	3 F600 0340	Watch chip
Backup RAM (50 bytes)	3 F600 0380 to 3 F600 0FC0	Watch chip
Gbus$WHAMI	3 F700 0000	CPU module
Gbus$LEDs	3 F700 0040	CPU module
Gbus$PMask	3 F700 0080	CPU module
Gbus$Intr	3 F700 00C0	CPU module
Gbus$Halt	3 F700 0100	CPU module
Gbus$LSBRST	3 F700 0140	CPU module
Gbus$Misc	3 F700 0180	CPU module
Gbus$RMode	3 F780 0000	CPU module
Gbus$LTagRW	3 F780 0100	LEVI

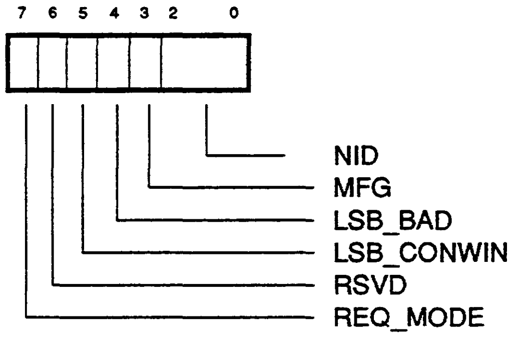
[1] **UART Base Addresses:**

xx = 0B; BASE = 3 F400 0000
xx = 0A; BASE = 3 F400 0080
xx = 1B; BASE = 3 F480 0000
xx = 1A; BASE = 3 F480 0080
xx = 2B; BASE = 3 F500 0000
xx = 2A; BASE = 3 F500 0080

Gbus$WHAMI

Address	3 F700 0000
Access	RO

The Gbus$WHAMI register provides information on system configuration and reflects the status of certain backplane signals.



BXB-0243A-93

Table 7-3 Gbus$WHAMI Register Bit Definitions

Name	Bit(s)	Type	Function
REQ_MODE	<7>	RO	**Request Mode.** Indicates the maximum number of CPU modules that this CPU module supports in a system.

Gbus$WHAMI <7>	CPUs Allowed in LSB Slots
0	0–3
1	0–7

Table 7-3 Gbus$WHAMI Register Bit Definitions (Continued)

Name	Bit(s)	Type	Function
LSB_CONWIN	<5>	RO	**LSB CONWIN.** Reflects the inverted state of the LSB_CONWIN L backplane signal. When set, indicates that Gbus$LEDs<1> is clear (asserted) in one or more CPU modules.
LSB_BAD	<4>	RO	**LSB Bad.** Reflects the inverted state of the LSB_BAD L backplane signal. When set, indicates that LSB_BAD L is driven by one or more CPU modules.
MFG	<3>	RO	**Manufacturing Status.** Used by manufacturing.
NID	<2:0>	RO	**Node ID.** Identifies the CPU module by the slot (0–7) where it resides.

Gbus$LEDs

Address 3 F700 0040
Access R/W

The Gbus$LEDs register is used for lighting a series of LEDs on the module to aid in debug and to indicate self-test status. Writing a zero to a bit in this register lights the corresponding LED.

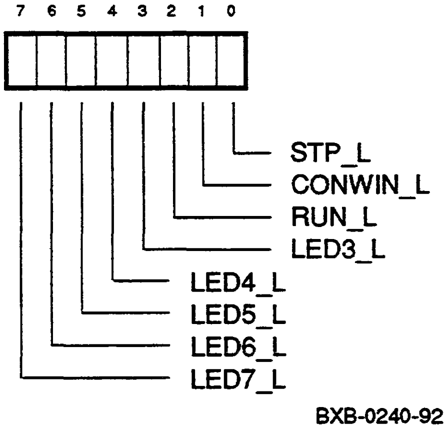


Table 7-4 Gbus$LEDs Register Bit Definitions

Name	Bit(s)	Type	Function
LEDs_L	<7:3>	R/W	**LEDs Low.** When a bit in this field is set, the associated LED signal is asserted low.
RUN_L	<2>	R/W	**RUN Low.** When set, the associated LED signal is asserted low. The state of this bit also indicates whether the currently running software is the operating system (and not the diagnostic/console program).
CONWIN_L	<1>	R/W	**CONWIN Low.** When set, the associated LED signal is asserted low. Also drives the backplane signal LSB_CONWIN L. The state of this signal can be read through the Gbus$WHAMI register.
STP_L	<0>	R/W	**Self-Test Passed Low.** When set, the associated LED signal is asserted low.

Gbus$PMask

Address 3 F700 0080
Access R/W

The Gbus$PMask register controls halts to the processor.

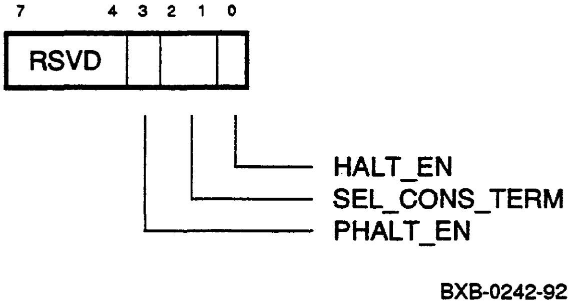


Table 7-5 Gbus$PMask Register Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<7:4>	R/W, 1	**Reserved.** Initialized to ones.
PHALT_EN	<3>	R/W, 1	**Ctrl/P Halt Enable.** When set, enables Ctrl/P characters received by the UART selected in the Select Console Terminal field of this register to halt the processor. The Halt Enable bit of this register must also be set for a Ctrl/P character to generate a halt.

Table 7-5 Gbus$PMask Register Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>SEL_CONS_TERM</td><td>&lt;2:1&gt;</td><td>R/W, 1</td><td><b>Select Console Terminal.</b> Selects one of three console terminals for Ctrl/P character detection.
<table>
<tr><th>Gbus$PMask &lt;2:1&gt;</th><th>Console Terminal Selected</th></tr>
<tr><td>00</td><td>UART0A (local terminal)</td></tr>
<tr><td>01</td><td>UART0B (Reserved)</td></tr>
<tr><td>10</td><td>UART1A (remote diagnostic control)</td></tr>
<tr><td>11</td><td>UART2A placed into module-level loopback mode. In this mode, the UART2A receive line is driven by the UART2A transmit line. PHALT_EN (bit <3> of this register) must be zero (Ctrl/P halts disabled) while modifying SEL_CONS_TERM to avoid erroneous halts.</td></tr>
</table></td></tr>
<tr><td>HALT_EN</td><td>&lt;0&gt;</td><td>R/W, 1</td><td><b>Halt Enable.</b> When set, enables halts to the processor, including halts generated by LCNR<NHALT> or by detection of a Ctrl/P character received by a UART selected in the Select Console Terminal field of this register. When clear, all halts to the processor are disabled. PHALT_EN must also be set for Ctrl/P characters to generate a halt.</td></tr>
</table>

Gbus$Intr

Address 3 F700 00C0

Access R/W

The Gbus$Intr register stores interrupt summary information. Specifically, it provides a means to determine the source of IPL 14, IPL 15, and IPL 16 interrupts to the processor.

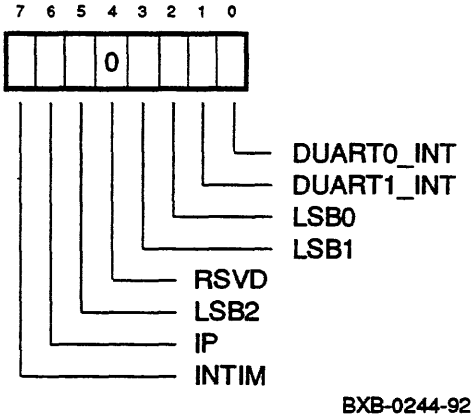


Table 7-6 Gbus$Intr Register Bit Definitions

Name	Bit(s)	Type	Function
INTIM	<7>	RO, 0	**Interval Timer.** When set, indicates that the watch chip is asserting its interval timer output.
IP	<6>	W1C, 0	**Interprocessor.** When set, indicates that the LEVI-A chip has detected a write to the LIPINTR register with data selecting this node.
LSB2	<5>	RO, 0	**LSB 2.** When set. indicates that the LEVI-A chip has an LSB level 2 interrupt pending.
RSVD	<4>	R0	**Reserved.** Reads as zero.

Table 7-6 Gbus$Intr Register Bit Definitions (Continued)

Name	Bit(s)	Type	Function
LSB1	<3>	RO, 0	**LSB 1.** When set, indicates that the LEVI-A chip has an LSB level 1 interrupt pending.
LSB0	<2>	RO, 0	**LSB 0.** When set, indicates that the LEVI-A chip has an LSB level 0 interrupt pending.
DUART1_INT	<1>	RO, 0	**DUART1 Interrupt.** When set, indicates that either UART1A or UART1B is requesting an interrupt for the processor. This bit is cleared when all possible DUART1 interrupt sources are cleared.
DUART0_INT	<0>	RO, 0	**DUART0 Interrupt.** When set, indicates that either UART0A or UART0B is requesting an interrupt for the processor. This bit is cleared when all possible DUART0 interrupt sources are cleared.

Gbus$Halt

Address 3 F700 0100
Access R/W

The Gbus$Halt register summarizes halt and power conditions.

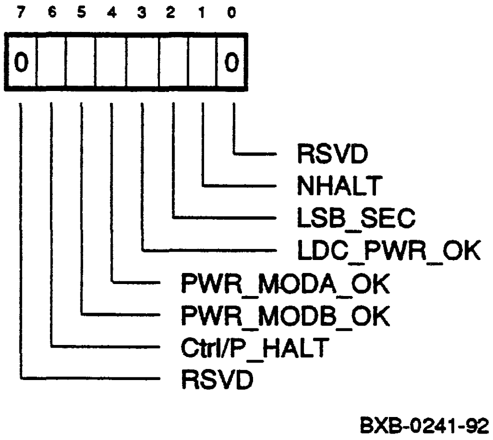


Table 7-7 Gbus$Halt Register Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<7>	R0	**Reserved.** Reads as zero.
Ctrl/P_HALT	<6>	W1C, 0	**Ctrl/P Halt.** Set when a Ctrl/P character is received by the UART selected in the Gbus$PMask register.
PWR_MODB_OK	<5>	RO	**Power Module B Okay.** Set when Power Module B of the I/O PIUs (plug-in unit) is working properly. Cleared when Module B fails.
PWR_MODA_OK	<4>	RO	**Power Module A Okay.** Set when Power Module A of the I/O PIUs (plug-in unit) is working properly. Cleared when Module A fails.

Table 7-7 Gbus$Halt Register Bit Definitions (Continued)

Name	Bit(s)	Type	Function
LDC_PWR_OK	<3>	RO	**LDC Power Okay.** Is set when all local disk converters (LDC) in the platform are working properly. Cleared when no LDCs are installed or when one or more of the LDCs fails.
LSB_SEC	<2>	RO	**LSB Secure.** Reflects the inverted state of the backplane signal LSB_SECURE L. When set, indicates that the control panel keyswitch is in the Secure position and that Ctrl/P halts to the processor are disabled by hardware.
NHALT	<1>	RO	**Node Halt.** Reflects the state of LCNR<NHALT>.
RSVD	<0>	R0	**Reserved.** Reads as zero.

Gbus$LSBRST

Address 3 F700 0140
Access R/W

The Gbus$LSBRST register is used for initiating a system reset sequence. When the CPU chip writes any value to this register, the LSB RESET signal is asserted for 512 LSB cycles.

BXB-0264-92

Gbus$Misc

Address 3 F700 0180
Access R/W

The Gbus$Misc register controls various system functions.

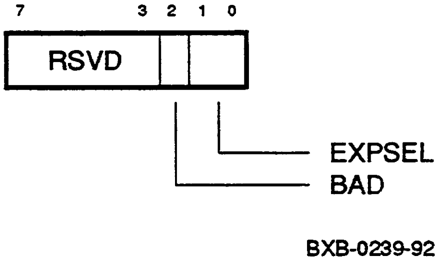


Table 7-8 Gbus$Misc Register Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<7:3>	RO, 1	**Reserved.** Initialized to ones.
BAD	<2>	R/W, 1	**Bad.** When set, causes the module to drive LSB BAD which, in turn, lights the control panel fault LED. The state of this bit does not affect the Self-Test-Passed LED on the module or the STP bits in the Gbus$LEDs and LCNR registers. This bit allows software to assert LSB BAD on behalf of another system component. To determine if any module is driving LSB BAD, software should read Gbus$WHAMI<LSB_BAD>, not Gbus$Misc<BAD>.

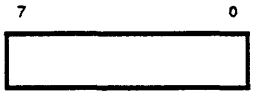
Table 7-8 Gbus$Misc Register Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>EXPSEL</td><td>&lt;1:0&gt;</td><td>R/W, 1</td><td><b>Expander Select.</b> Selects which cabinet the power supply UART lines are logically connected to, and therefore, which of three 48V regulators are connected to the power supply lines.
<table>
<tr><th>Gbus$Misc
&lt;1:0&gt;</th><th>Power Supply Connection</th></tr>
<tr><td>00</td><td>PS lines logically connected to main CPU cabinet.</td></tr>
<tr><td>01</td><td>PS lines logically connected to right expander cabinet.</td></tr>
<tr><td>10</td><td>PS lines logically connected to left expander cabinet.</td></tr>
<tr><td>11</td><td>PS transmit line is looped back to PS receive line.</td></tr>
</table>
</td></tr>
</table>

Gbus$RMode

Address 3 F780 0000
Access R/W

The Gbus$RMode register is a write-only register. A write to it sets LDIAG<FRIGN> and logically disconnects the CPU module from the LSB bus. This register is intended for use as a backup system should there be a problem with the LSB interface and writes to the LDIAG register be unsuccessful (writes to the LDIAG register require a successful LSB transaction while writes to Gbus space are completed without any LSB access). Note that software should write to the LDIAG register as a first choice and use the Gbus$RMode register only if the write to the LDIAG register fails.

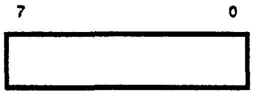


BXB-0264-92

Gbus$LTagRW

Address 3 F780 0100
Access R/W

The Gbus$LTagRW register, when used with LTAGA, LTAGW, and LDIAG registers, allows software to read and write the B-cache, B-map, and P-map tags. See descriptions of the LTAGA, LTAGW, and LDIAG registers in Chapter 9.



BXB-0264-92

Chapter 8

I/O Operations

I/O operations handled by the KN7AA CPU module include I/O reads, I/O writes, and device interrupts. The DECchip 21064 uses four hardcoded SCB vectors for all device interrupts. Interrupt service routines at the four SCB vectors are required to determine the source of the interrupt and invoke the appropriate service routine.

From the perspective of I/O operations, registers are divided into two groups: local registers and remote registers. Registers that reside on the KN7AA CPU module and the LSB bus are local registers. Those that reside on I/O buses are remote registers. Local registers are directly accessible to software; remote registers are not. Access to remote registers is achieved by means of the mailbox protocol. The LMBOX register is provided to assist software in the mailbox protocol.

8.1 Mailbox Data Structure

Remote control and status registers (CSRs) are accessed through 64-byte naturally aligned mailbox data structures located in main memory. Read requests are posted in mailboxes. Data is returned in memory with status in the following quadword. Mailboxes are allocated and managed by the operating system software. Figure 8-1 shows a mailbox data structure.

Figure 8-1 Mailbox Data Structure

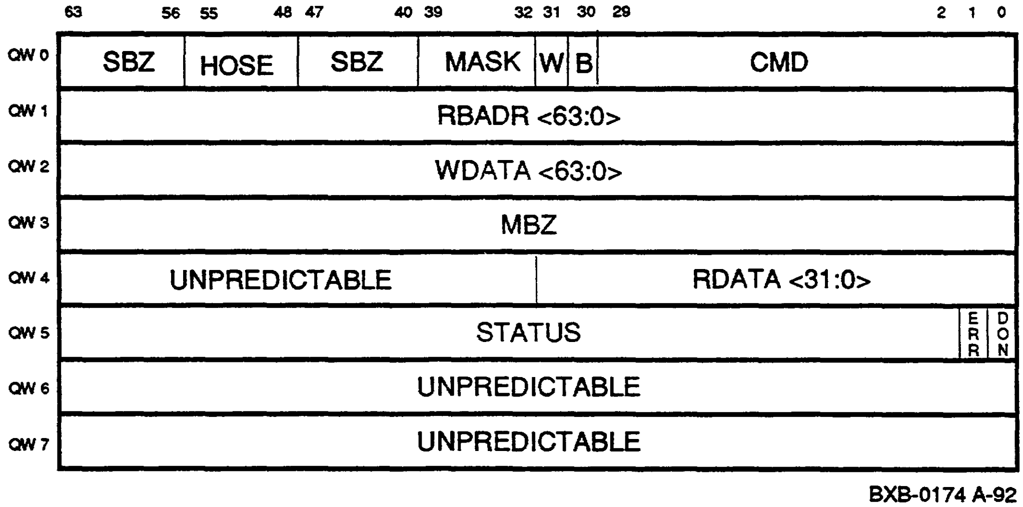


Table 8-1 describes the mailbox data structure. Refer to the *DEC 7000 AXP System / VAX 7000 I / O System Technical Manual* for a detailed description of the mailbox protocol.

Table 8-1 Mailbox Data Structure

Field	Bit(s)	Type	Quad-word	Function
HOSE	<55:48>	R/W	0	**Hose.** Used to determine which remote bus the command is meant for.
MASK	<39:32>	R/W	0	**Mask.** Contains the byte mask. The I/O module does not use this field.
CMD	<29:0>	R/W	0	**Command.** Contains the command. Value is I/O bus adapter specific.
RBADR	<63:0>	R/W	1	**Remote Broadcast Address.** Contains the address to be broadcast on the remote bus.
WDATA	<63:0>	R/W	2	**Write Data.** Contains the write data to be broadcast on the remote bus.
RDATA	<31:0>	R/W	4	**Read Data.** Contains read data returned from the remote bus.
STATUS	<63:2>	R/W	5	**Status.** Contains status information provided by the remote bus.
ERR	<1>	R/W	5	**Error.** When set, indicates that a mailbox operation failed.
DON	<0>	R/W	5	**Done.** Status bit set by the I/O module when a mailbox operation is complete.

8.2 Mailbox Operation

The I/O module services mailbox requests by means of four mailbox pointer CSRs (LMBPR registers; see Section 8.4) located in the I/O module's node space. There is one LMBPR address for each CPU node. Software sees only one LMBPR register address, but the CPU module replaces the least significant two bits of the address (that is, D<2:1>) with the least significant two bits of the node ID (that is, NID<1:0>). If a given LMBPR register is in use when it is written to, the I/O module does not acknowledge it and CNF is not asserted. Processors use the lack of CNF assertion on writes to the LMBPR register to indicate a busy status. The write is retried later under software control.

To perform a write to the LMBPR register, microcode must know the address of the LMBPR register and the address of the mailbox data structure to be loaded into the LMBPR register. Another memory structure needs to be created to pass this information to microcode. This structure is called the Mailbox Pointer and consists of two longwords. Figure 8-2 shows the mailbox pointer structure. Table 8-2 gives the bit definitions of the mailbox pointer structure.

Figure 8-2 Mailbox Pointer Structure

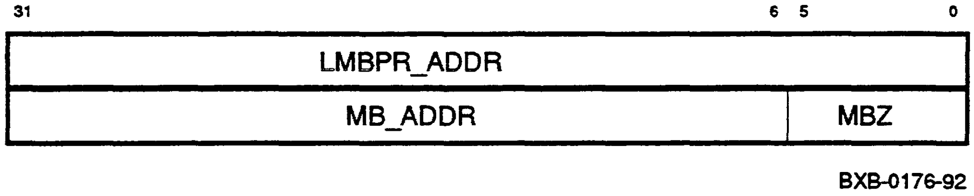


Table 8-2 Mailbox Pointer Structure

Name	Bit(s)	Type	Function
MB_ADDR	<31:6>	WO	**Mailbox Address.** Contains the physical address of the mailbox data structure. Since this structure is aligned on a 64-byte boundary, bits <5:0> of the address must be zero.
LMBPR_ADDR	<31:0>	WO	**LMBPR Address.** Contains the virtual address of the LMBPR register.

When software has created the mailbox data structure and the mailbox pointer structure, it can start the I/O operation. An MTPR to the LMBOX register (Section 8.4) initiates the I/O operation. Microcode reads the MB_ADDR field out of the mailbox pointer structure and then writes the value to the LMBPR register using the address provided in the mailbox pointer structure. An EDAL store conditional command is used to perform the write. Microcode then checks the Zero Condition Code bit (PSL<2>) in the BIU_STAT register to determine if the write passed or failed. If the write passed, PSL<2> is set; otherwise, PSL<2> is cleared. Software can loop on the MTPR to the LMBOX register until the write passes.

After the I/O module has accepted the write to LMBPR, it performs the I/O operation. Software can now poll the status bit in the mailbox data structure until the I/O operation is complete. When the I/O operation has completed, DON in the mailbox data structure (see Table 8-1) is set. If an error occurred during the transaction, LBER<E> (see Chapter 9) is also set. If the operation was an I/O write, no further action is required. If the operation was an I/O read, software can now fetch the returned data from the RDATA field in the mailbox data structure.

8.3 Device Interrupt Handling

The KN7AA module uses the device interrupts as shown in Table 8-3. Interrupts from the LSB and the UARTs (device interrupts) are handled by both hardware and software. After an interrupt has been posted to the CPU chip through one of the four IRQ lines, the CPU chip passes control to the operating system through four dedicated SCB entry points. Table 8-3 shows the device interrupt sources and their matching SCB entry points.

Table 8-3 KN7AA CPU Interrupts

Interrupt Level (Hex)	Interrupt Condition	DECchip 21064 IRQ Pin	SCB Vector
17	LSB level 3 interrupts	3	DC
16	Interprocessor interrupt	2	D8
16	LSB level 2 interrupts	2	D8
15	Console UARTs	1	D4
15	LSB level 1 interrupts	1	D4
14	LSB level 0 interrupts	0	D0

For IPL 16 and IPL 17 interrupts, software reads the Gbus$Intr register to determine if the interrupt is posted by an LSB I/O device, another processor in the system, or a UART. If an interprocessor or a UART interrupt has been received, software can directly pass control to the appropriate service routine. For LSB I/O interrupts, software must get the device interrupt vector from the I/O module.

8.4 I/O Operation Registers

Two registers are used for I/O operations:

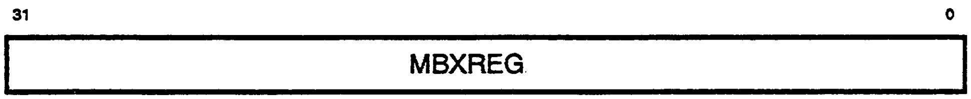
- Mailbox Pointer CSR (LMBPR)
- Mailbox Register (LMBOX)

The LMBPR register resides on the IOP module and is described in the *DEC 7000 AXP System / VAX 7000 I / O System Technical Manual.* The description of the LMBOX register follows.

LMBOX—LSB Mailbox Register

Address BB + 00
Access R/W

The LMBOX register contains the physical address of the mailbox pointer structure.



BXB-0175-92

Table 8-4 LMBOX Register Bit Definitions

Name	Bit(s)	Type	Function
MBXREG	<31:0>	WO	**Mailbox Register.** Contains the physical address of the mailbox pointer structure.

Chapter 9

CPU Module Registers

The KN7AA CPU module, like the memory and I/O modules on the LSB bus, contains two groups of registers:

- LSB required registers
- CPU-specific registers

LSB required registers are used for internode communication over the LSB bus. CPU-specific registers are used to perform functions specific to the CPU module.

9.1 Register Mapping

All CPU module registers reside in node space. The only exceptions to this rule are the two interrupt registers, LIOINTR and LIPINTR, which reside in LSB broadcast space.

CPU module registers are mapped to the node space as offsets to a base address (BB). The base address is implemented in hardware and depends on the node ID, which is determined by the LSB backplane slot occupied by the module. Table 9-1 gives the physical base addresses of nodes on the LSB bus.

Table 9-1 LSB Node Space Base Addresses

Node ID	Module	Physical Base Address (BB) (Byte)
0	CPU/Memory	3 F800 0000
1	CPU/Memory	3 F840 0000
2	CPU/Memory	3 F880 0000
3	CPU/Memory	3 F8C0 0000
4	CPU/Memory	3 F900 0000
5	CPU/Memory	3 F940 0000
6	CPU/Memory	3 F980 0000
7	CPU/Memory	3 F9C0 0000
8	I/O	3 FA00 0000

Table 9-2 lists the CPU module registers and gives the address of each register as an offset from a selected node space base address.

NOTE: *Two CPU registers listed in Table 9-2, LIOINTR and LIPINTR, are located in LSB broadcast space, the base address of which is 3 FE00 0000.*

Table 9-2 CPU Module Registers

Register Name	Mnemonic	Address (Byte Offset)
LSB Required		
Device Register	LDEV	BB[1] + 0000
Bus Error Register	LBER	BB + 0040
Configuration Register	LCNR	BB + 0080
Memory Mapping Register 0	LMMR0	BB + 0200
Memory Mapping Register 1	LMMR1	BB + 0240
Memory Mapping Register 2	LMMR2	BB + 0280
Memory Mapping Register 3	LMMR3	BB + 02C0
Memory Mapping Register 4	LMMR4	BB + 0300
Memory Mapping Register 5	LMMR5	BB + 0340
Memory Mapping Register 6	LMMR6	BB + 0380
Memory Mapping Register 7	LMMR7	BB + 03C0
Bus Error Syndrome Register 0	LBESR0	BB + 0600
Bus Error Syndrome Register 1	LBESR1	BB + 0640
Bus Error Syndrome Register 2	LBESR2	BB + 0680
Bus Error Syndrome Register 3	LBESR3	BB + 06C0
Bus Error Command Register 0	LBECR0	BB + 0700
Bus Error Command Register 1	LBECR1	BB + 0740
I/O Interrupt Register	LIOINTR	BSB[2] + 0000
Interprocessor Interrupt Register	LIPINTR	BSB + 0040
CPU-Specific		
Mode Register	LMODE	BB + 0C00
Module Error Register	LMERR	BB + 0C40
Lock Address Register	LLOCK	BB + 0C80
Diagnostic Control Register	LDIAG	BB + 0D00
Tag Address Register	LTAGA	BB + 0D40
Tag Write Data Register	LTAGW	BB + 0D80
Console Communication Register 0	LCON0	BB + 0E00
Console Communication Register 1	LCON1	BB + 0E40
Performance Counter Control Register	LPERF	BB + 0F00
Performance Counter 0 Register	LCNTR0	BB +0F40
Performance Counter 1 Register	LCNTR1	BB + 0F80
Last Miss Address Register	LMISSADDR	BB + 0FC0

[1] BB is the node space base address of the CPU module in hex.

[2] BSB is the broadcast space base address, which is 3 FE00 0000.

9.2 Register Descriptions

LSB required registers have the following characteristics:

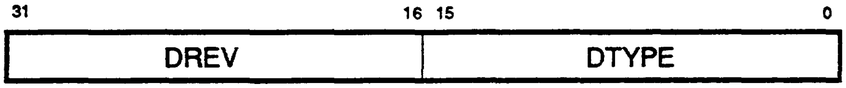
- All writes are 32 bits wide. Byte or word operations are not supported.
- Writes directed to a read-only register may be accepted and acknowledged, but no action is taken, and the content of the register is not affected.

CPU-specific registers appear in the LSB CSR space.

LDEV—Device Register

Address BB + 0000
Access R/W

The LDEV register is loaded during initialization with information that identifies a node.



BXB-0100-92

Table 9-3 LDEV Register Bit Definitions

Name	Bit(s)	Type	Function
DREV	<31:16>	R/W, 0	**Device Revision.** Identifies the revision level of an LSB node. For the KN7AA CPU module, the value of this field is zero.
DTYPE	<15:0>	R/W, 0	**Device Type.** Identifies the type of node. For the KN7AA CPU module, the value of this field is set to 8001 (hex).

LBER—Bus Error Register

Address BB + 0040
Access R/W

The LBER register stores the error bits that are flagged when an LSB node detects errors in the LSB operating environment and logs the failing commander ID. The status of this register remains locked until software resets the error bit(s).

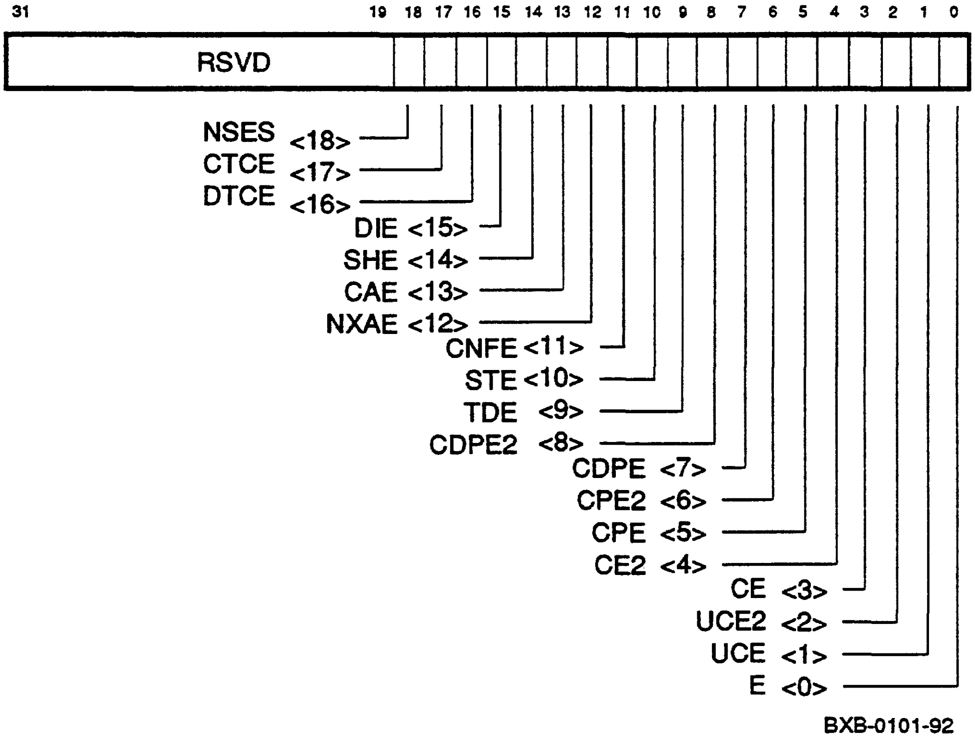


Table 9-4 LBER Register Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<31:19>	R0	**Reserved.** Read as zero.
NSES	<18>	R, 0	**Node-Specific Error Summary**. Set when an error condition is reported in the LMERR register.
CTCE	<17>	W1C, 0	**Control Transmit Check Error**. Set when an LSB control line is driven incorrectly by the CPU module. When CTCE is set, ERR is asserted by the CPU module for one cycle.
DTCE	<16>	W1C, 0	**Data Transmit Check Error.** Set when the CPU module detects an error while driving the D<127:0> and ECC<27:0> lines during a data or command cycle. When DTCE is set, ERR is asserted by the CPU module for one cycle.
DIE	<15>	W1C, 0	**Dirty Error.** Set if the CPU module receives an asserted DIRTY signal during a cycle when DIRTY signals are not allowed. When DIE is set, ERR is asserted by the CPU module for one cycle.
SHE	<14>	W1C, 0	**Shared Error.** Set if the CPU module receives an asserted SHARED signal during a cycle when SHARED signals are not allowed. When SHE is set, ERR is asserted by the CPU module for one cycle.
CAE	<13>	W1C, 0	**Command/Address Error.** Set if the CPU module receives an asserted CA signal during a cycle when CA signals are not allowed. When CAE is set, ERR is asserted by the CPU module and error registers are locked.
NXAE	<12>	W1C, 0	**Nonexistent Address Error.** Set when the CPU module does not receive confirmation for a command it sent on the LSB. When NXAE is set, ERR is asserted by the CPU module for one cycle.
CNFE	<11>	W1C, 0	**CNF Error.** Set if the CPU module receives a confirmation signal during a cycle that does not permit confirmation. When CNFE is set, ERR is asserted by the CPU module for one cycle.
STE	<10>	W1C, 0	**STALL Error.** Set when the CPU module receives a STALL signal during a cycle that does not permit stalls. When STE is set, ERR is asserted by the CPU module for one cycle.
TDE	<9>	W1C, 0	**Transmitter During Error.** Set if a CE, UCE, CPE, or CDPE error occurs during a cycle when the CPU module was driving D<127:0>. When TDE is set, ERR is asserted by the CPU module for one cycle.

Table 9-4 LBER Register Bit Definitions (Continued)

Name	Bit(s)	Type	Function
CDPE2	<8>	W1C, 0	**Second CSR Data Parity Error.** Set when a second parity error occurs while CDPE is set on a CSR data cycle.
CDPE	<7	W1C, 0	**CSR Data Parity Error.** If a parity error occurs during a CSR data cycle, the CPU module sets CDPE, asserts ERR for one cycle, and locks the error registers.
CPE2	<6>	W1C, 0	**Second Command Parity Error.** Set when a second parity error occurs on a command cycle while CPE is set.
CPE	<5>	W1C, 0	**Command Parity Error.** If a parity error occurs on a command cycle, the CPU module sets CPE, asserts ERR for one cycle, and locks the error registers.
CE2	<4>	W1C, 0	**Second Correctable Data Error.** Set when a second correctable ECC error occurs on a data cycle while CE is set.
CE	<3>	W1C, 0	**Correctable Data Error.** If the CPU module detects an ECC error on the LSB, it sets CE, asserts ERR for one cycle, and locks the error registers.
UCE2	<2>	W1C, 0	**Second Uncorrectable Data Error.** Set when the CPU module detects a second uncorrectable data error while UCE is set.
UCE	<1>	W1C, 0	**Uncorrectable Data Error.** If the CPU module detects an uncorrectable ECC error on the LSB during a data cycle, it sets UCE, asserts ERR for one cycle, and locks the error registers.
E	<0>	W1C, 0	**Error.** Set whenever the CPU module detects assertion of ERR on the LSB.

LCNR—Configuration Register

Address BB + 0080
Access R/W

The LCNR register contains LSB configuration setup and status information.

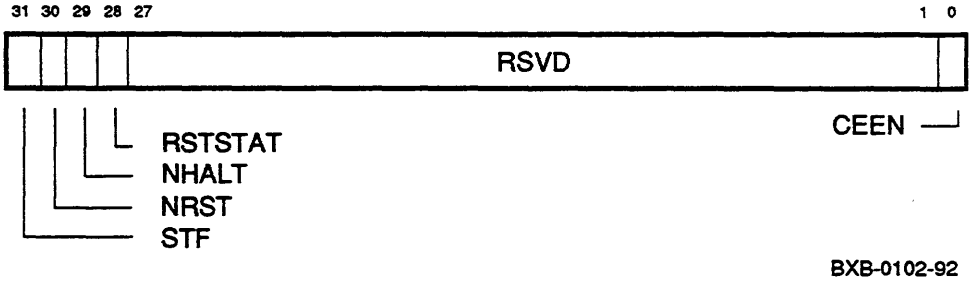


Table 9-5 LCNR Register Bit Definitions

Name	Bit(s)	Type	Function
STF	<31>	W1C, 1	**Self-Test Fail.** When set, indicates that this node has not yet completed self-test.
NRST	<30>	W, 0	**Node Reset.** When set, the node enters console mode and undergoes a reset sequence.
NHALT	<29>	R/W, 0	**Node Halt.** When set, a CPU node enters console mode.
RSTSTAT	<28>	W1C, 0	**Reset Status.** When set, provides an indication to console software that a given CPU node should not attempt to become the boot processor, but should rather join an already running system. This bit is set when NRST (LCNR<30>) is set. It is cleared with a write of one, at system power-up, or with an LSB RESET command. This bit is not cleared in a reset sequence caused by setting NRST.
RSVD	<27:1>	R0	**Reserved.** Read as zero.
CEEN	<0>	R/W, 0	**Correctable Error Detection Enable.** When set, enables detection of correctable errors.

LMMR0–7—Memory Mapping Registers

Address BB + 0200 to BB + 03C0
Access R/W

Eight LMMR registers define the memory configuration for all memory modules installed in the system. They are copies of the equivalent AMR registers in memory modules installed in the system. Each LMMR register is associated with the LSB module whose node ID matches the three lower bits of the LMMR address. Thus, LMMR0 is associated with node 0, LMMR1 is associated with node 1, and so on. LMMR registers are loaded during system initialization when the memory modules are initialized and configured.

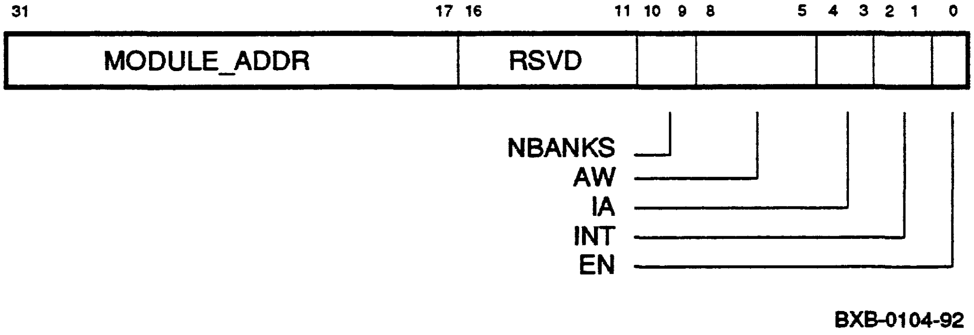


Table 9-6 LMMR Register Bit Definitions

Name	Bit(s)	Type	Function
MODULE_ADDR	<31:17>	R/W	**Module Address.** Specifies the most significant bits of the base address of the memory region spanned by the memory module associated with this register (LMMR0–LMMR7). These bits correspond to bits <39:25> of the byte address or D<34:20> of the command cycle.
RSVD	<16:11>	R0	**Reserved.** Read as zero.

Table 9-6 LMMR Register Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>NBANKS</td><td><10:9></td><td>R/W</td><td><b>Number of Banks.</b> Specifies the number of individual memory banks (1, 2, 4, or 8) contained on the memory module associated with this register (LMMR0–7). The value of this field determines how many bits of the memory address (0, 1, 2, or 3) are inserted into the bank number.
<table>
<tr><th>LMMR
<10:9></th><th>Banks per
Module</th><th>Bits in Bank
Number</th></tr>
<tr><td>00</td><td>1</td><td>0</td></tr>
<tr><td>01</td><td>2</td><td>1</td></tr>
<tr><td>10</td><td>4</td><td>2</td></tr>
<tr><td>11</td><td>8</td><td>3</td></tr>
</table></td></tr>
<tr><td>AW</td><td><8:5></td><td>R/W</td><td><b>Address Width.</b> Specifies the number of valid bits in MODULE_ADDR (LMMR<31:17>), starting from the MSB. The remaining bits of MODULE_ADDR are ignored.</td></tr>
<tr><td>IA</td><td><4:3></td><td>R/W</td><td><b>Interleave Address.</b> Specifies which interleave, within a group of interleaved modules, is served by the module associated with this register (LMMR0–7).</td></tr>
<tr><td>INT</td><td><2:1></td><td>R/W</td><td><b>Interleave.</b> Specifies the number of memory modules interleaved with this module (1, 2, or 4). This value determines the number of bits in the INT field (0, 1, or 2, starting from the LSB) that are compared to the LSBs of the memory address.
<table>
<tr><th>LMMR
<2:1></th><th>Modules
Interleaved</th><th>Address Bits
Compared</th></tr>
<tr><td>00</td><td>1</td><td>0</td></tr>
<tr><td>01</td><td>2</td><td>1</td></tr>
<tr><td>10</td><td>4</td><td>2</td></tr>
<tr><td>11</td><td>Reserved</td><td>Reserved</td></tr>
</table></td></tr>
<tr><td>EN</td><td><0></td><td>R/W</td><td><b>Enable.</b> When set, indicates that the module associated with this register (LMMR0–7) is installed, and it is a memory module.</td></tr>
</table>

LBESR0-3—Bus Error Syndrome Registers

Address BB + 0600 06C0
Access R

The LBESR registers contain the syndrome computed from the LSB Data and ECC fields received during the cycle when an error was detected. The syndrome is the bit-by-bit difference between the ECC check code generated from the received data and the ECC field received over the bus. The LBESR registers lock only on the first occurrence of an ECC error (LBER<CE> or LBER<UCE>). Subsequent ECC errors set LBER<CE2> or LBER<UCE2> until software clears those error bits.

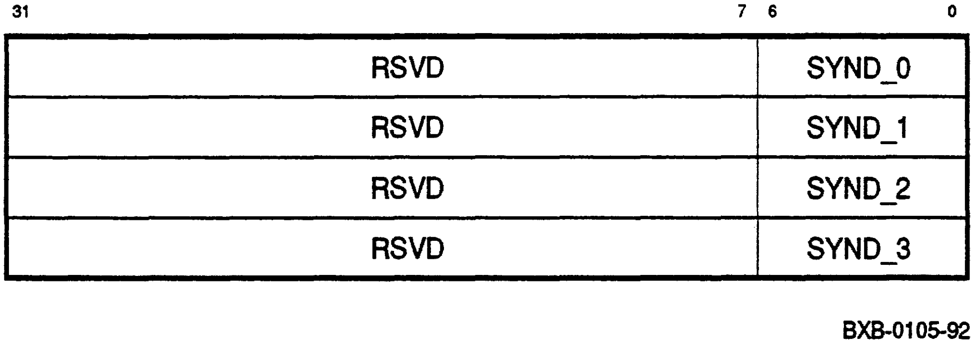


Table 9-7 LBESR Register Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<31:7>	R0	**Reserved.** Read as zero.
SYND_0	<6:0>	R	**Syndrome 0.** Syndrome computed from D<31:0> and ECC<6:0> during error cycle.
SYND_1	<6:0>	R	**Syndrome 1.** Syndrome computed from D<63:32> and ECC<13:7> during error cycle.
SYND_2	<6:0>	R	**Syndrome 2.** Syndrome computed from D<95:33> and ECC<20:14> during error cycle.
SYND_3	<6:0>	R	**Syndrome 3.** Syndrome computed from D<127:96> and ECC<27:21> during error cycle.

Syndrome Values

A syndrome of zero indicates no ECC error for the given longword. Table 9-8 gives the syndromes for all single-bit errors. Any non-zero syndrome not listed in Table 9-8 indicates a double-bit error.

Table 9-8 Syndromes for Single-Bit Errors

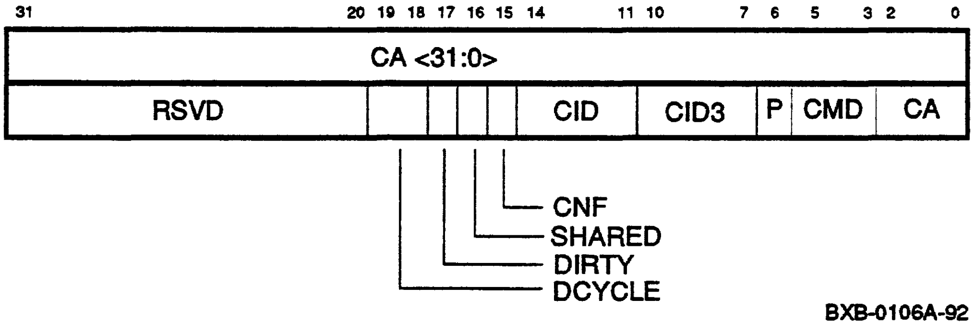
Bit	Syndrome (Hex)	Bit	Syndrome (Hex)
Data<0>	4F	Data<20>	16
Data<1>	4A	Data<21>	19
Data<2>	52	Data<22>	1A
Data<3>	54	Data<23>	1C
Data<4>	57	Data<24>	62
Data<5>	58	Data<25>	64
Data<6>	5D	Data<26>	67
Data<7>	23	Data<27>	68
Data<8>	25	Data<28>	6B
Data<9>	26	Data<29>	6D
Data<10>	29	Data<30>	70
Data<11>	2A	Data<31>	75
Data<12>	2C	ECC<0>	01
Data<13>	31	ECC<1>	02
Data<14>	34	ECC<2>	04
Data<15>	0E	ECC<3>	08
Data<16>	0B	ECC<4>	10
Data<17>	13	ECC<5>	20
Data<18>	15	ECC<6>	40
Data<19>			

LBECR0,1—Bus Error Command Registers

Address BB + 0700 and BB + 0740
Access R

The LBECR registers save the contents of the LSB command and address fields during the command cycle when an error is detected. The following errors detected by the CPU module lock the LBECR registers:

LSB uncorrectable ECC error (LBER<1>)
LSB correctable ECC error (LBER<3>)
LSB command parity error (LBER<5>)
LSB CSR data parity error (LBER<7>)
LSB nonexistent address error (LBER<12>)
LSB arbitration drop error (LMERR<10>
LEVI P-map parity error (LMERR<3:0>)
LEVI B-cache tag parity error (LMERR<4>)
LEVI B-cache status parity error (LMERR<5>)
LEVI B-map parity error (LMERR<6>)



BXB-0106A-92

Table 9-9 LBECR Register Bit Definitions

Name	Bit(s)	Type	Function
CA	<31:0>	R	**Command/Address.** Contents of D<31:0> during the command cycle.
RSVD	<31:20>	R0	**Reserved.** Read as zero.

Table 9-9 LBECR Register Bit Definitions (Continued)

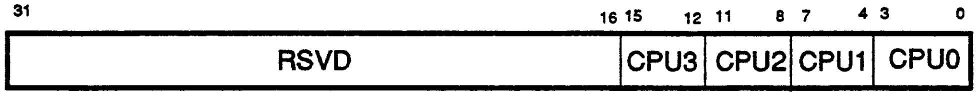
<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>DCYCLE</td><td>&lt;19:18&gt;</td><td>R</td><td><b>Data Cycle.</b> Indicates which data cycle had data error.
<table>
<tr><th>LBECR &lt;19:18&gt;</th><th>Data Cycle in Error</th></tr>
<tr><td>00</td><td>0</td></tr>
<tr><td>01</td><td>1</td></tr>
<tr><td>10</td><td>2</td></tr>
<tr><td>11</td><td>3</td></tr>
</table></td></tr>
<tr><td>DIRTY</td><td>&lt;17&gt;</td><td>R</td><td><b>Dirty.</b> Set when DIRTY is asserted for the current command.</td></tr>
<tr><td>SHARED</td><td>&lt;16&gt;</td><td>R</td><td><b>Shared.</b> Set when SHARED is asserted for the current command.</td></tr>
<tr><td>CNF</td><td>&lt;15&gt;</td><td>R</td><td><b>Confirmation.</b> Set when CNF is asserted for the current command.</td></tr>
<tr><td>CID</td><td>&lt;14:11&gt;</td><td>R</td><td><b>Commander ID.</b> Contents of REQ<3:0> during command cycle.</td></tr>
<tr><td>CID3</td><td>&lt;10:7&gt;</td><td>R</td><td><b>Commander ID 3.</b> This field is the duplicate of CID (bits <14:11>). It reads the same as CID. In some early versions of the KN7AA module, CID3 reads as zero.</td></tr>
<tr><td>P</td><td>&lt;6&gt;</td><td>R</td><td><b>Parity.</b> Contents of D<38> during command cycle.</td></tr>
<tr><td>CMD</td><td>&lt;5:3&gt;</td><td>R</td><td><b>Command.</b> Contents of D<37:35> during command cycle. CMD is decoded as follows:
<table>
<tr><th>Command</th><th>Function</th></tr>
<tr><td>000</td><td>Read</td></tr>
<tr><td>001</td><td>Write</td></tr>
<tr><td>010</td><td>Reserved</td></tr>
<tr><td>011</td><td>Write Victim</td></tr>
<tr><td>100</td><td>Read CSR</td></tr>
<tr><td>101</td><td>Write CSR</td></tr>
<tr><td>110</td><td>Reserved</td></tr>
<tr><td>111</td><td>Private</td></tr>
</table></td></tr>
<tr><td>CA</td><td>&lt;2:0&gt;</td><td>R</td><td><b>Command/Address.</b> Contents of D<34:32> during command cycle.</td></tr>
</table>

LIOINTR—I/O Interrupt Register

Address BSB + 0000
Access R/W

The LIOINTR register is used by the LSB I/O module to signal interrupts from the LSB I/O system to processors.

NOTE: A maximum of four processors can receive interrupts regardless of the system configuration. In a multiprocessor system with more than four CPU modules, only CPU0 to CPU3 can receive interrupts.



BXB-0109-92

Table 9-10 LIOINTR Register Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<31:16>	R0	**Reserved.** Read as zero.
CPU3	<15:12>	W1S	**CPU3 I/O Interrupt**. When a bit is set in this field, an interrupt is posted to CPU3.
CPU2	<11:8>	W1S	**CPU2 I/O Interrupt**. When a bit is set in this field, an interrupt is posted to CPU2.
CPU1	<7:4>	W1S	**CPU1 I/O Interrupt**. When a bit is set in this field, an interrupt is posted to CPU1.
CPU0	<3:0>	W1S	**CPU0 I/O Interrupt**. When a bit is set in this field, an interrupt is posted to CPU0.

Interrupt Mapping

Each interrupt target is assigned four bits of interrupt in the LIOINTR register corresponding to the four I/O interrupt levels. A given CPU only looks at the four bits that correspond to its target assignment. This allows interrupts to be targeted to a single CPU or up to four CPUs, depending on the data supplied in the bus CSR write transaction from the I/O module.

This register appears in LSB broadcast space. Writes that address this location are accepted without regard to node ID. Thus, all CPUs accept writes to the register. The register bits are write one to set (W1S). Multiple writes with a value of one to a given bit in this register post an equal number of interrupts to the targeted CPU. Reads to this location are undefined. Any given CPU implements only four bits of this register.

Table 9-11 shows the mapping of LSB interrupt levels to DECchip 21064 interrupt levels.

Table 9-11 LSB Interrupt Mapping

LSB Interrupt Level	DECchip 21064 IPL (Dec)
3	IPL 23
2	IPL 22
1	IPL 21
0	IPL 20

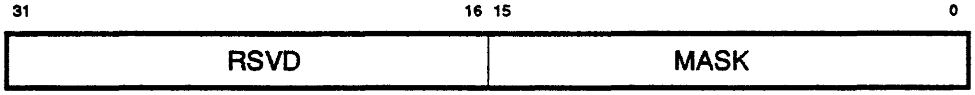
When any of the four interrupt-pending bits is set, the LEVI gate array correspondingly asserts the IOINTR<3:0> signals. The CPU module then uses these signals to assert the appropriate interrupt request to the DECchip 21064. The LEVI-A gate array also watches for LSB CSR reads to the LILID0–3 registers in the IOP module. When an LSB CSR read for LILID0 is asserted on the LSB bus, the LEVI-A gate array correspondingly deasserts IOINTR<0>. The LEVI-A gate array performs the same function on LILID3, LILID2, and LILID1.

NOTE: At least one CPU module must reside in slots 0 to 3.

LIPINTR—Interprocessor Interrupt Register

Address BSB + 0040
Access R/W

The LIPINTR register is used by the CPU modules to signal interprocessor interrupts.



BXB-0120-92

Table 9-12 LIPINTR Register Bit Definitions

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>RSVD</td><td>&lt;31:16&gt;</td><td>R0</td><td><b>Reserved.</b> Read as zero.</td></tr>
<tr><td>MASK</td><td>&lt;15:0&gt;</td><td>W1S, 0</td><td><b>Interprocessor Interrupt Mask.</b> When a given bit is set, an interprocessor interrupt is posted to a specific processor. Bits are mapped to specific CPUs within a multiprocessor system as follows:
<table>
<tr><th>LIPINTR Bits</th><th>DECchip 21064 CPU</th></tr>
<tr><td>&lt;15:8&gt;</td><td>Not used.</td></tr>
<tr><td>&lt;7&gt;</td><td>CPU7</td></tr>
<tr><td>&lt;6&gt;</td><td>CPU6</td></tr>
<tr><td>&lt;5&gt;</td><td>CPU5</td></tr>
<tr><td>&lt;4&gt;</td><td>CPU4</td></tr>
<tr><td>&lt;3&gt;</td><td>CPU3</td></tr>
<tr><td>&lt;2&gt;</td><td>CPU2</td></tr>
<tr><td>&lt;1&gt;</td><td>CPU1</td></tr>
<tr><td>&lt;0&gt;</td><td>CPU0</td></tr>
</table></td></tr>
</table>

Interprocessor Interrupt

When a processor wishes to post an interrupt to another processor, it simply writes to the LIPINTR register to set the relevant bit. The bits in LIPINTR<7:0> are write one to set (W1S).

This register appears in LSB broadcast space. Writes that address this location are accepted without regard to node ID. Thus, all CPUs accept writes to the register. Reads to this location are undefined.

The contents of LIPINTR<7:0> are qualified by the node ID. If a given CPU node is selected, the LEVI-A gate array asserts the IPINTR signal for one processor external clock. The CPU module ORs this signal and issues the appropriate interrupt request to the DECchip 21064.

LMODE—Mode Register

Address BB + 0C00
Access R/W

The LMODE register contains mode setup for an operational CPU module (as opposed to the LDIAG register which provides mode setup for a CPU module while running diagnostics).

NOTE: Pass 1 or 2 and Pass 3 LEVI bit definitions of the LMODE register are given in separate tables. See bits <19:16> for the LEVI revision.

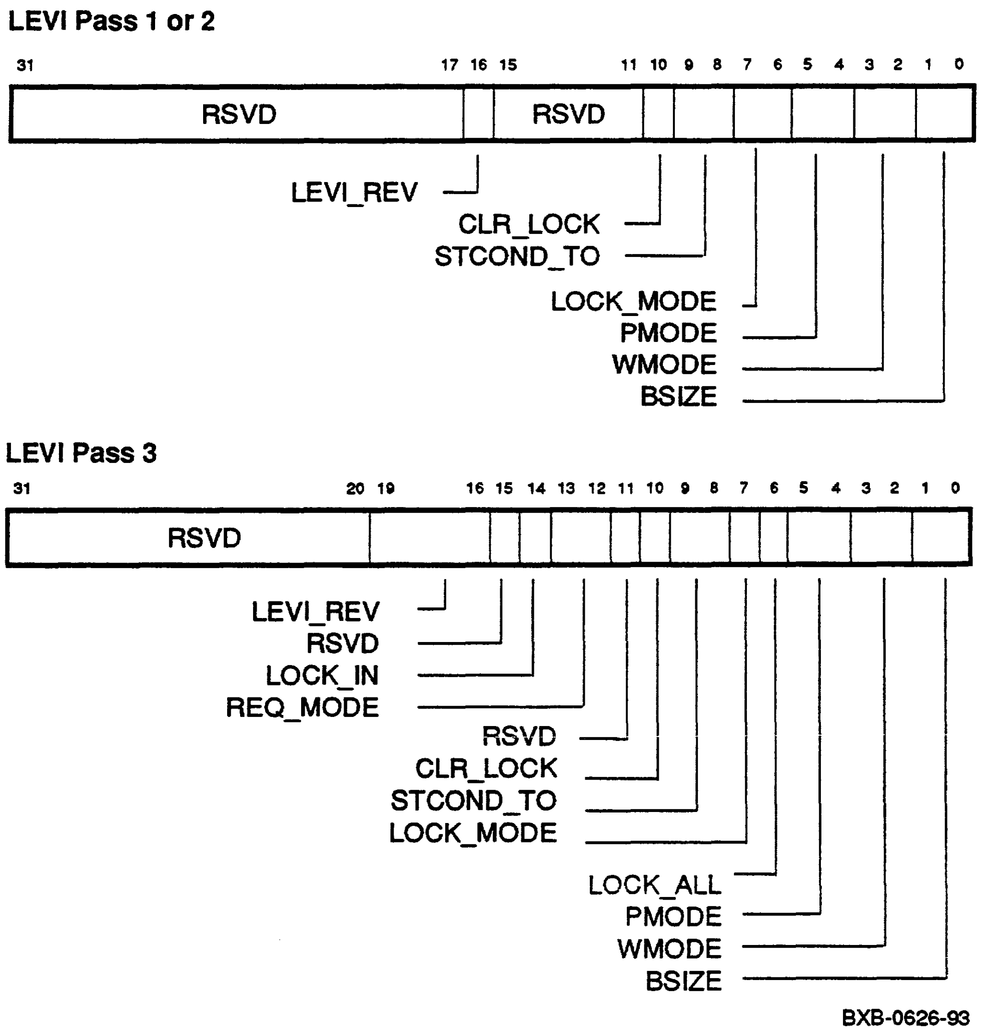


Table 9-13 LMODE Register Pass 1 and Pass 2 LEVI Bit Definitions

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>RSVD</td><td><31:17></td><td>R0</td><td><b>Reserved.</b> Read as zero.</td></tr>
<tr><td>LEVI_REV</td><td><16></td><td>R, X</td><td><b>LEVI Revision.</b> When clear, indicates pass 1 LEVI-A. When set, indicates pass 2 LEVI-A. See Table 9-14 for pass 1, pass 2, and pass 3 LEVI_A codes.</td></tr>
<tr><td>RSVD</td><td><15:11></td><td>R0</td><td><b>Reserved.</b> Read as zero.</td></tr>
<tr><td>CLR_LOCK</td><td><10></td><td>W, 0</td><td><b>Clear Lock.</b> When set, forces LEVI to deassert LSB_LOCKOUT and clear any relevant saved state irrespective of the state of LOCK_TIME, and so on.</td></tr>
<tr><td>STCOND_TO</td><td><9:8></td><td>R/W, 0</td><td><b>Store Conditional Timeout.</b> Unused on the KN7AA module. Should be written with zeros.</td></tr>
<tr><td>LOCK_MODE</td><td><7:6></td><td>R/W, 0</td><td><b>Lock Mode.</b> Unused on the KN7AA module. Should be written with zeros.</td></tr>
<tr><td>PMODE</td><td><5:4></td><td>R/W, 0</td><td><b>P-Cache Mode.</b> Allows LEVI to work with CPU chips with varying internal cache organizations. The value of this field for the KN7AA module is 01 (bin), which denotes an 8K D-cache and an 8K I-cache.</td></tr>
<tr><td>WMODE</td><td><3:2></td><td>R/W, 0</td><td><b>Write Mode.</b> Selects the behavior of LEVI in response to LSB writes.
<table>
<tr><th>LMODE
<3:2></th><th>LEVI Behavior</th></tr>
<tr><td>00</td><td>Use results of P-map lookup to determine invalidate/update.</td></tr>
<tr><td>01</td><td>Invalidate the B-cache.</td></tr>
<tr><td>10</td><td>Update the B-cache.</td></tr>
<tr><td>11</td><td>LEVI behavior undefined.</td></tr>
</table></td></tr>
<tr><td>BSIZE</td><td><1:0></td><td>R/W, 0</td><td><b>B-Cache Size.</b> Tells LEVI about the size of the B-cache.
<table>
<tr><th>LMODE
<1:0></th><th>B-Cache Size</th></tr>
<tr><td>00</td><td>4 Mbytes</td></tr>
<tr><td>01</td><td>LEVI behavior undefined.</td></tr>
<tr><td>10</td><td>LEVI behavior undefined.</td></tr>
<tr><td>11</td><td>LEVI behavior undefined.</td></tr>
</table></td></tr>
</table>

Table 9-14 LMODE Register Pass 3 LEVI Bit Definitions

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>RSVD</td><td><31:20></td><td>R0</td><td><b>Reserved.</b> Read as zero.</td></tr>
<tr><td>LEVI_REV</td><td><19:16></td><td>R, X</td><td><b>LEVI Revision.</b> Indicate revision of LEVI-A.
<table>
<tr><th>LMODE
<19:16></th><th>LEVI Revision</th></tr>
<tr><td>0000</td><td>Pass 1 LEVI-A</td></tr>
<tr><td>0001</td><td>Pass 2 LEVI-A</td></tr>
<tr><td>0011</td><td>Pass 3 LEVI-A</td></tr>
<tr><td>All else</td><td>Reserved</td></tr>
</table></td></tr>
<tr><td>RSVD</td><td><15></td><td>R0</td><td><b>Reserved.</b> Reads as zero.</td></tr>
<tr><td>LOCK_IN</td><td><14></td><td>R/W, 0</td><td><b>Lock In.</b> When set, LEVI-A asserts LSB LOCKOUT if LEVI-A signal R_CRD is asserted. This bit should be set along with LMODE<LOCK_ALL>. The use of LOCK_IN is intended for system debug only and should normally be cleared.</td></tr>
<tr><td>REQ_MODE</td><td><13:12></td><td>R/W, 0</td><td><b>Request Mode.</b> Determine the CPU module configurations by controlling the number of LSB REQ lines used by LEVI-A for arbitration. This field should allow no more than are allowed by Gbus$WHAMI<REQ_MODE>.
<table>
<tr><th>LMODE
<13:12></th><th>CPUs Allowed in LSB Slots</th></tr>
<tr><td>00</td><td>0 to 3</td></tr>
<tr><td>11</td><td>0 to 7</td></tr>
<tr><td>All else</td><td>Reserved</td></tr>
</table></td></tr>
<tr><td>RSVD</td><td><11></td><td>R0</td><td><b>Reserved.</b> Read as zero.</td></tr>
<tr><td>CLR_LOCK</td><td><10></td><td>W, 0</td><td><b>Clear Lock.</b> When set, forces LEVI to deassert LSB_LOCKOUT and clear any relevant saved state irrespective of the state of LOCK_TIME, and so on.</td></tr>
<tr><td>STCOND_TO</td><td><9:8></td><td>R/W, 0</td><td><b>Store Conditional Timeout.</b> Unused on the KN7AA module. Should be written with zeros.</td></tr>
</table>

Table 9-14 LMODE Register Pass 3 LEVI Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>LOCK_MODE</td><td>&lt;7&gt;</td><td>R/W, 0</td><td><b>Lock Mode.</b> Unused on the KN7AA module. Should be written with zero.</td></tr>
<tr><td>LOCK_ALL</td><td>&lt;6&gt;</td><td>R/W, 0</td><td><b>Lock All.</b> When set, prevents all LSB transactions (except secondary and victim writes) if LSB LOCKOUT is asserted by another node. This bit should be set along with LMODE<LOCK_IN>. The use of LOCK_ALL is intended for debug only. This bit should normally be cleared.</td></tr>
<tr><td>PMODE</td><td>&lt;5:4&gt;</td><td>R/W, 0</td><td><b>P-Cache Mode.</b> Allows LEVI to work with CPU chips with varying internal cache organizations. The value of this field for the KN7AA module is 01 (bin), which denotes an 8K D-cache and an 8K I-cache.</td></tr>
<tr><td>WMODE</td><td>&lt;3:2&gt;</td><td>R/W, 0</td><td><b>Write Mode.</b> Selects the behavior of LEVI in response to LSB writes.
<table>
<tr><th>LMODE
&lt;3:2&gt;</th><th>LEVI Behavior</th></tr>
<tr><td>00</td><td>Use results of P-map lookup to determine invalidate/update.</td></tr>
<tr><td>01</td><td>Invalidate the B-cache.</td></tr>
<tr><td>10</td><td>Update the B-cache.</td></tr>
<tr><td>11</td><td>LEVI behavior undefined.</td></tr>
</table></td></tr>
<tr><td>BSIZE</td><td>&lt;1:0&gt;</td><td>R/W, 0</td><td><b>B-Cache Size.</b> Tells LEVI about the size of the B-cache.
<table>
<tr><th>LMODE
&lt;1:0&gt;</th><th>B-Cache Size</th></tr>
<tr><td>00</td><td>4 Mbytes</td></tr>
<tr><td>01</td><td>1 Mbyte</td></tr>
<tr><td>10</td><td>16 Mbytes</td></tr>
<tr><td>11</td><td>LEVI behavior undefined.</td></tr>
</table></td></tr>
</table>

LMERR—Module Error Register

Address BB + 0C40
Access R/W

The LMERR register provides module-specific error information. If any bits are set in this register, NSES (LBER<18>) is also set.

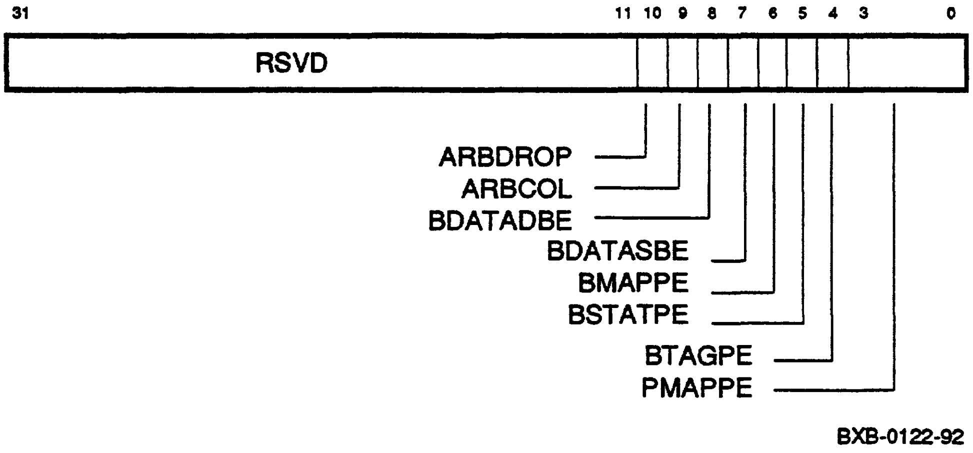


Table 9-15 LMERR Register Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<31:11>	R0	**Reserved.** Read as zero.
ARBDROP	<10>	W1C, 0	**Arbitration Drop.** Set when the LEVI arbitration logic detects an LSB cycle in which a node has failed to assert a command after having gained access to the LSB bus. When ARBDROP is set, the LSB command and address are latched in the LBECR register.
ARBCOL	<9>	W1C, 0	**Arbitration Collision.** Set when the LEVI arbitration logic detects an attempt to arbitrate for the LSB bus in an illegal time slot.

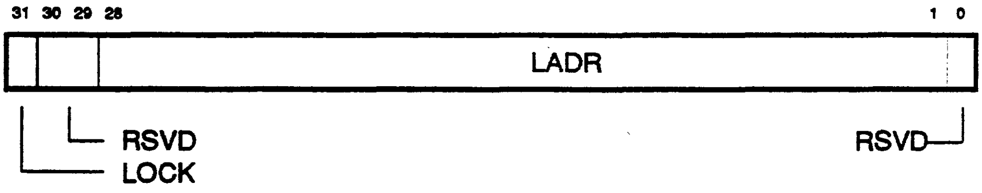
Table 9-15 LMERR Register Bit Definitions (Continued)

Name	Bit(s)	Type	Function
BDATADBE	<8>	W1C, 0	**B-Cache Data Double-Bit Error.** When set, indicates that the LEVI chips have detected a double-bit ECC error when unloading the B-cache RAMs. This bit is set when data being transmitted on the LSB bus incurs a double-bit error. The address and the associated ECC syndrome are latched in the LBECR and LBESR registers, respectively.
BDATASBE	<7>	W1C, 0	**B-Cache Data Single-Bit Error.** When set, indicates that the LEVI chips have detected a single-bit ECC error when unloading the B-data RAMs. This bit is set when data being transmitted on the LSB bus incurs a single-bit error. The address and the associated ECC syndrome are latched in the LBECR and LBESR registers, respectively.
BMAPPE	<6>	W1C, 0	**B-Map Parity Error.** When set, indicates that the LEVI-A chip has detected bad parity when reading the B-map RAMs. The associated address is latched in the LBECR register.
BSTATPE	<5>	W1C, 0	**B-Cache Status Store Parity Error.** When set, indicates that the LEVI-A chip has detected bad parity when reading the B-stat RAM. The associated address is latched in the LBECR register for LSB probes. For LEVI probes due to processor misses (LDx_L and so on), BIU_STAT<0> is set and the address is latched in the FILL_ADDR register.
BTAGPE	<4>	W1C, 0	**B-Cache Tag Store Parity Error.** When set, indicates that the LEVI chips have detected bad parity when reading the B-tag RAMs. The associated address is latched in the LBECR register for LSB probes. For LEVI probes due to processor misses (LDx_L and so on), BIU_STAT<0> is set and the address is latched in the FILL_ADDR register.
PMAPPE	<3:0>	W1C, 0	**P-Map Parity Error.** A bit is set in this field if the LEVI-A chip detects bad parity when reading one of the four internal P-map RAM structures. The associated address is latched in the LBECR register.

LLOCK—Lock Address Register

Address BB + 0C80
Access R

The LLOCK register contains the physical address and lock bit of the most recently executed LDxL instruction that referenced memory space.



BXB-0126-92

Table 9-16 LLock Register Bit Definitions

Name	Bit(s)	Type	Function
LOCK	<31>	W1C, 0	**Lock.** When set, indicates that the LLOCK register contains a valid address used in the most recent memory space LDx_L instruction executed by the processor and that no LSB writes that reference the same 64-byte LSB block have occurred. This bit is used to determine the response to a subsequent STx_C instruction. Software can clear this bit explicitly with an LSB write to the 64-byte block referenced in LLOCK<28:1>.
RSVD	<30:29>	R0	**Reserved.** Read as zero.
LADR	<28:1>	R, 0	**Lock Address**. Lock address bits <33:6>.
RSVD	<0>	R0	**Reserved.** Reads as zero.

LDIAG—Diagnostic Control Register

Address	BB + 0D00
Access	R/W

The LDIAG register allows a diagnostic program to manipulate various sections of the CPU module for complete testing.

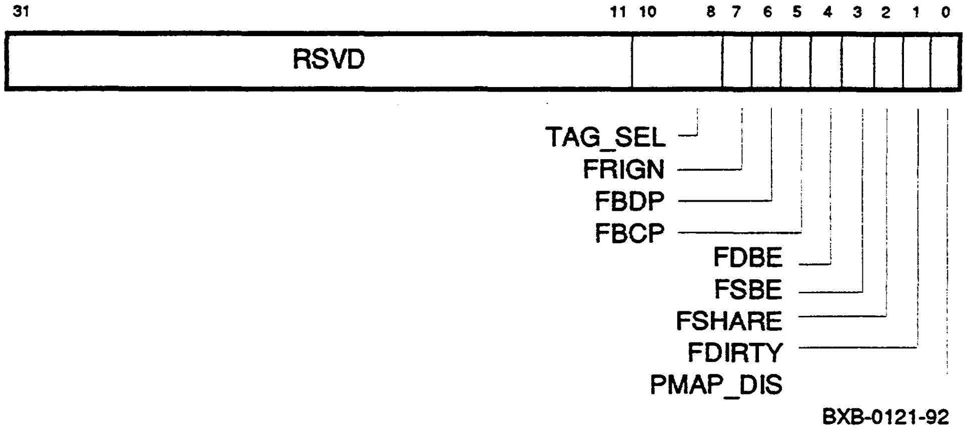


Table 9-17 LDIAG Register Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<31:11>	R0	**Reserved.** Read as zero.
TAG_SEL	<10:8>	R/W, 0	**Tag Select.** Specifies which tag store is to be read/written when Gbus$LtagRW is accessed. LDIAG<10:8> — Tag Store Selected 100 — B-cache 010 — B-map 001 — P-map When LDIAG is being used to read a tag, only one bit in TAG_SEL is allowed to be set. If more than one bit is set in TAG_SEL when Gbus$LtagRW is written, all specified tags are written.

Table 9-17 LDIAG Register Bit Definitions (Continued)

Name	Bit(s)	Type	Function
FRIGN	<7>	R/W, 1	**Force LSB Ignore.** When set, forces the LEVI gate arrays to ignore all LSB bus traffic except transactions initiated by this node. When FRIGN transitions from set to clear, the LEVI gate array sets LSB ERR to allow all LSB arbitration to resync.
FBDP	<6>	R/W, 0	**Force Bad Data Parity.** When set, forces the LEVI-A gate array to assert bad parity on the LSB during CSR data cycles.
FBCP	<5>	R/W, 0	**Force Bad Command Parity.** When set, forces the LEVI-A gate array to assert bad parity on the LSB during CSR command cycles.
FDBE	<4>	R/W, 0	**Force Double-Bit Error.** Allows a diagnostic program to force the LEVI gate arrays to load data into the B-cache with double-bit ECC errors. When set, LEVI inverts every ECC bit for each longword loaded into the B-cache from the LSB bus. This bit is only relevant when the LEVI gate arrays are loading the B-cache data store (fills).
FSBE	<3>	R/W, 0	**Force Single-Bit Error.** Allows a diagnostic program to force the LEVI gate arrays to load data into the B-cache with single-bit ECC errors. When set, LEVI inverts one ECC bit for each longword loaded into the B-cache from the LSB bus. This bit is only relevant when the LEVI gate arrays are loading the B-cache data store (fills).
FSHARE	<2>	R/W, 0	**Force Share.** When set, the LEVI-A chip responds to all LSB memory transactions from other nodes with assertion of SHARED.
FDIRTY	<1>	R/W, 0	**Force Dirty.** When set, the LEVI-A chip responds to all LSB memory space read transactions that hit in the B-map with assertion of DIRTY and supplies the data from the B-cache to the LSB.
PMAP_DIS	<0>	R/W, 0	**P-Map Disable.** On LEVI-A pass 3 chips, when set, disables the P-map and causes LEVI-A to behave as if all lockups hit in the P-map. This bit is don't care for earlier revisions of LEVI-A. It should be written with zero.

Diagnostic Notes

The following notes offer additional information for performing diagnostics on the CPU module.

- **How to Make the B-Cache Emulate Main Memory**
  The CPU module can be made to present its cache as main memory to the LSB environment by setting BIU_CTL<FHIT>, LDIAG<FDIRTY>, and LMODE<WMODE>=10 (bin). The selection of this mode is possible under the following two conditions: (1) Only a single CPU module is placed in this mode; (2) No memory module is present in the system.

- **How to Read/Write Tags**
  The combinations of LDIAG, LTAGA, LTAGW, and Gbus$LtagRW registers allow diagnostic programs or error recovery programs to read or write any tag store on the CPU module. LDIAG<TAG_SEL> selects the tag store of interest; LTAGA selects the location in the tag store; LTAGW supplies the value to be written into the tag; and Gbus$LtagRW provides the mechanism. The use of the Gbus register allows LEVI to perform the tag access without the need for any special setup (that is, FRIGN). Even though the Gbus registers are specified to be a byte in length, reads from Gbus$LtagRW return a full longword of data, since no physical Gbus location is actually being read. Gbus address space is used for convenience only.

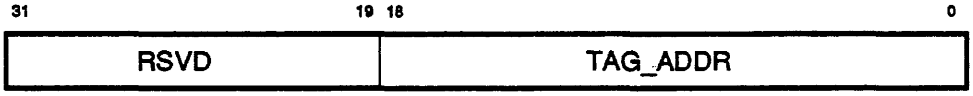
  Writing Tags
  1. Write LDIAG<TAG_SEL> to select the tag store.
  2. Write LTAGA to select the location.
  3. Write LTAGW to specify the value to be written.
  4. Write Gbus$LtagRW with any random data. This action triggers LEVI to perform the tag write as set up.

  Reading Tags
  1. Write LDIAG<TAG_SEL> to select the tag store.
  2. Write LTAGA to select the location.
  3. Read Gbus$LtagRW. This action triggers LEVI to perform the tag read as set up. The data returned is the value from the selected tag location in the format specified by the LTAGW register.

LTAGA—Tag Address Register

Address BB + 0D40
Access R/W

The LTAGA register provides a means by which a diagnostic program can specify the location to be accessed in the CPU cache data and tag RAM structures.



BXB-0123-92

Table 9-18 LTAGA Register Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<31:19>	R0	**Reserved.** Read as zero.
TAG_ADDR	<18:0>	R/W, 0	**Tag Address.** Specifies the location (tag address bits <23:5>) to be accessed in the tag store selected by LDIAG<TAG_SEL>.

LTAGW—Tag Write Data Register

Address BB + 0D80
Access R/W

The LTAGW register provides a means by which a diagnostic program can specify the value to be loaded into the CPU caches and tag RAM structures.

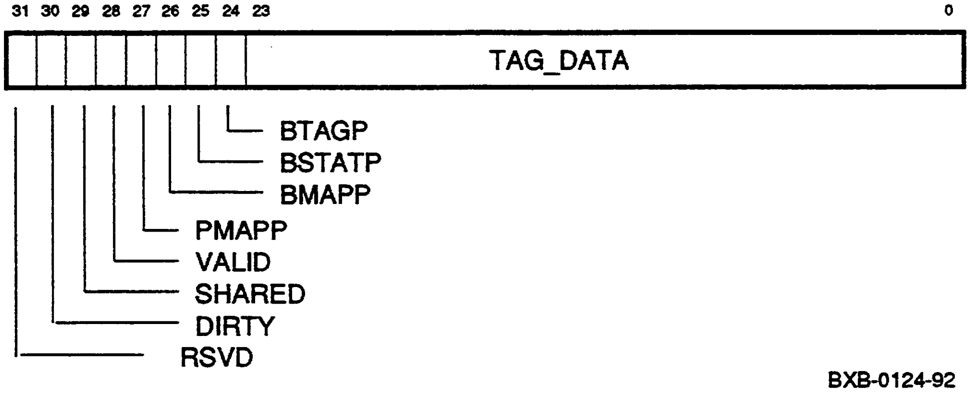


Table 9-19 LTAGW Register Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<31>	R0	**Reserved.** Reads as zero.
DIRTY	<30>	R/W, 0	**Dirty.** Loaded into the Dirty field (if any) of the B-stat store specified in LDIAG<TAG_SEL> when WTAG (LDIAG<4>) is set.
SHARED	<29>	R/W, 0	**Shared.** Loaded into the Shared field (if any) of the B-stat store specified in LDIAG<TAG_SEL> when WTAG (LDIAG<4>) is set.
VALID	<28>	R/W, 0	**Valid.** Loaded into the Valid field (if any) of the B-stat store specified in LDIAG<TAG_SEL> when WTAG (LDIAG<4>) is set.
PMAPP	<27>	R/W, 0	**P-Map Parity.** Specifies the value to be loaded in the B-map parity location when the Gbus$LtagRW register is written. PMAPP covers tag data bits <23:10> and the valid bit (even parity).

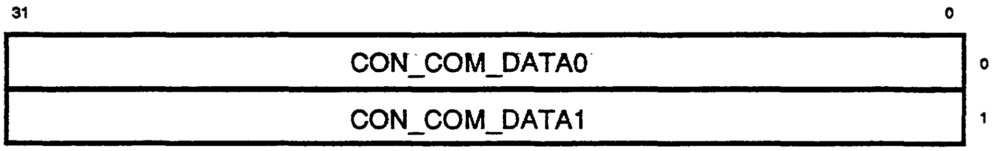
Table 9-19 LTAGW Register Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>BMAPP</td><td>&lt;26&gt;</td><td>R/W, 0</td><td><b>B-Map Parity.</b> Specifies the value to be loaded in the B-map parity location when the Gbus$LtagRW register is written. BMAPP covers tag data bits <33:22> and the valid bit (even parity).</td></tr>
<tr><td>BSTATP</td><td>&lt;25&gt;</td><td>R/W, 0</td><td><b>B-Stat Parity.</b> Specifies the value to be loaded in the B-stat parity location when the Gbus$LtagRW register is written. BSTATP covers the Shared, Dirty, and Valid bits.</td></tr>
<tr><td>BTAGP</td><td>&lt;24&gt;</td><td>R/W, 0</td><td><b>B-Tag Parity.</b> Specifies the value to be loaded in the B-tag parity location when the Gbus$LtagRW register is written. BTAGP covers tag data bits <33:22> (even parity).</td></tr>
<tr><td>TAG_DATA</td><td>&lt;23:0&gt;</td><td>R/W, 0</td><td><b>Tag Data.</b> Loaded into the tag store specified in LDIAG<TAG_SEL> when the Gbus$LtagRW register is written. Mapping is performed as follows:
<table>
<tr><th>LTAGW&lt;23:0&gt;</th><th>Tag</th><th>RAM Structure</th></tr>
<tr><td>&lt;23:12&gt;</td><td>&lt;33:22&gt;</td><td>B-cache tag</td></tr>
<tr><td>&lt;23:12&gt;</td><td>&lt;33:22&gt;</td><td>B-map tag</td></tr>
<tr><td>&lt;13:0&gt;</td><td>&lt;23:10&gt;</td><td>P-map tag</td></tr>
</table>
</td></tr>
</table>

LCON0,1—Console Communication Registers

Address BB + 0E00 and BB + 0E40
Access R/W

The LCON register provides a nonmemory communication location for the KN7AA console firmware. The value contained in this register has no direct effect on any CPU module hardware.



BXB-0129-92

Table 9-20 LCON Register Bit Definitions

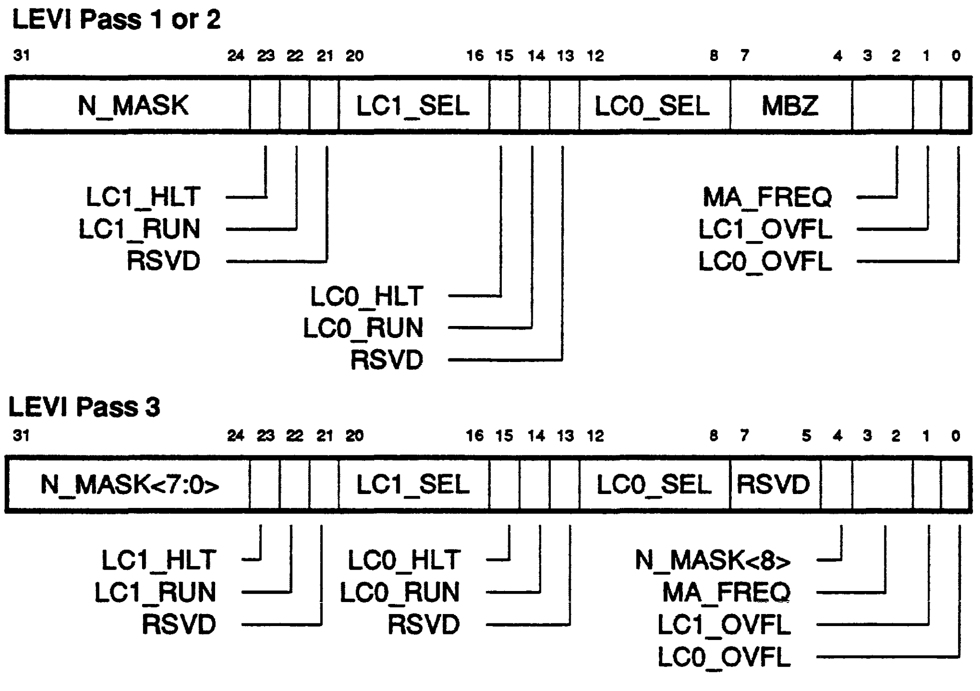
Name	Bit(s)	Type	Function
CON_COM_DATA0	<31:0>	R/W, 0	**Console Communication Data 0.** Data stored in the LCON0 register.
CON_COM_DATA1	<31:0>	R/W, 0	**Console Communication Data 1.** Data stored in the LCON1 register.

LPERF—Performance Counter Control Register

Address	BB + 0F00
Access	R/W

The LPERF register defines how the LEVI performance registers (LCNTR0, LCNTR1, and LMISSADDR) behave. Each counter register has an event select field, control bits, and an overflow bit. Some of the events to be counted are subject to the node ID mask. The LMISSADDR register loads miss addresses based on the value of the Miss Address Frequency field

NOTE: Pass 1 or 2 and Pass 3 LEVI bit definitions of the LMISSADDR register are given in separate tables. See LMODE<LEVI_REV> for the LEVI revision.



BXB-0634-93

Table 9-21 LPERF Register Pass 1 and Pass 2 LEVI Bit Definitions

Name	Bit(s)	Type	Function
N_MASK	<31:24>	R/W, 0	**Node Mask.** When a bit is set in this field, LSB reads, LSB writes, or victim writes are counted for the associated node. The bits are in one-to-one correspondence, so that bit <31> is associated with node 7, bit <30> with node 6, and so on, except for bit <24>, which is associated with node 0 and the IOP. More than one bit may be set, allowing transactions from multiple nodes to be counted. This field applies to both LCNTR registers.
LC1_HLT	<23>	W, 0	**LCNTR1 Halt.** Write one to disable LCNTR1 counting.
LC1_RUN	<22>	W, 0	**LCNTR1 Run.** Write one to enable LCNTR1 counting.
RSVD	<21>	R0	**Reserved.** Reads as zero.
LC1_SEL	<20:16>	R/W, 0	**LCNTR1 Select.** Selects event for the LCNTR1 register.

LPERF <20:16>[1]	Event to Count in LCNTR1
00000	Misses due to reads
00001	Misses due to writes
00010	Misses due to shared blocks
00011	LDL_X instructions
00100	STC_X failures
00101	LSB B-map hits
00110	Bank conflict delays —LSB cycles
00111	Arbitration losses
01000	Victim buffer hits —wrapped only
01001	CSR reads
01010	CSR writes
01011	LMBPR writes
01100	LSB interrupts
01101	Gbus reads
01110	Gbus writes
01111	LSB reads —subject to node mask
10000	LSB writes —subject to node mask
10001	Victim writes —subject to node mask
10010	Stall cycles
10011	Total memory latency
10100	Carry out from LCNTR0
Other	Reserved

[1] When counting victim buffer hits, LC1_SEL = 01000 counts only hits for which LEVI has to swap (or wrap) the first and second hexwords to satisfy the read request in the proper order. LC0_SEL = 01000 counts all (wrapped or unwrapped) hits.

Table 9-21 LPERF Register Pass 1 and Pass 2 LEVI Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>LC0_HLT</td><td>&lt;15&gt;</td><td>W, 0</td><td><b>LCNTR0 Halt.</b> Writing one disables LCNTR0 counting.</td></tr>
<tr><td>LC0_RUN</td><td>&lt;14&gt;</td><td>W, 0</td><td><b>LCNTR0 Run.</b> Writing one enables LCNTR0 counting.</td></tr>
<tr><td>RSVD</td><td>&lt;13&gt;</td><td>R0</td><td><b>Reserved.</b> Reads as zero.</td></tr>
<tr><td>LC0_SEL</td><td>&lt;12:8&gt;</td><td>R/W, 0</td><td><b>LCNTR0 Select.</b> Selects event for the LCNTR0 register.
<table>
<tr><th>LPERF
&lt;12:8&gt;[1]</th><th>Event to Count in LCNTR0</th></tr>
<tr><td>00000</td><td>Misses due to reads</td></tr>
<tr><td>00001</td><td>Misses due to writes</td></tr>
<tr><td>00010</td><td>Misses due to shared blocks</td></tr>
<tr><td>00011</td><td>LDL_X instructions</td></tr>
<tr><td>00100</td><td>STC_X failures</td></tr>
<tr><td>00101</td><td>LSB B-map hits</td></tr>
<tr><td>00110</td><td>Bank conflict delays —LSB cycles</td></tr>
<tr><td>00111</td><td>Arbitration losses</td></tr>
<tr><td>01000</td><td>Victim buffer hits —wrapped or unwrapped</td></tr>
<tr><td>01001</td><td>CSR reads</td></tr>
<tr><td>01010</td><td>CSR writes</td></tr>
<tr><td>01011</td><td>LMBPR writes</td></tr>
<tr><td>01100</td><td>LSB interrupts</td></tr>
<tr><td>01101</td><td>Gbus reads</td></tr>
<tr><td>01110</td><td>Gbus writes</td></tr>
<tr><td>01111</td><td>LSB reads —subject to node mask</td></tr>
<tr><td>10000</td><td>LSB writes —subject to node mask</td></tr>
<tr><td>10001</td><td>Victim writes —subject to node mask</td></tr>
<tr><td>10010</td><td>Stall cycles</td></tr>
<tr><td>10011</td><td>Total memory latency</td></tr>
<tr><td>10100</td><td>Carry out from LCNTR1</td></tr>
<tr><td>Other</td><td>Reserved</td></tr>
</table>
[1] When counting victim buffer hits, LC1_SEL = 01000 counts only hits for which LEVI has to swap (or wrap) the first and second hexwords to satisfy the read request in the proper order. LC0_SEL = 01000 counts all (wrapped or unwrapped) hits.</td></tr>
<tr><td>RSVD</td><td>&lt;7:4&gt;</td><td>R0</td><td><b>Reserved.</b> Read as zero.</td></tr>
</table>

Table 9-21 LPERF Register Pass 1 and Pass 2 LEVI Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>MA_FREQ</td><td><3:2></td><td>R/W, 0</td><td><b>Miss Address Frequency.</b> Determines how often the LMISSADDR register loads the last miss address.
<table>
<tr><th>LPERF
<3:2></th><th>Miss Frequency</th></tr>
<tr><td>00</td><td>Load every 32nd miss address</td></tr>
<tr><td>01</td><td>Load every 64th miss address</td></tr>
<tr><td>10</td><td>Load every 128th miss address</td></tr>
<tr><td>11</td><td>Load every 256th miss address</td></tr>
</table></td></tr>
<tr><td>LC1_OVFL</td><td><1></td><td>R, 0</td><td><b>LCNTR1 Overflow.</b> Records overflow from the LCNTR1 register. Clears when the LCNTR1 register is reset.</td></tr>
<tr><td>LC0_OVFL</td><td><0></td><td>R, 0</td><td><b>LCNTR0 Overflow.</b> Records overflow from the LCNTR0 register. Clears when the LCNTR0 register is reset.</td></tr>
</table>

Table 9-22 LPERF Register Pass 3 LEVI Bit Definitions

Name	Bit(s)	Type	Function
N_MASK	<31:24>	R/W, 0	**Node Mask.** When a bit is set in this field, LSB reads, LSB writes, or victim writes are counted for the associated node. The bits are in one-to-one correspondence, so that bit <31> is associated with node 7, bit <30> with node 6, and bit <24> for node 0. The IOP module (node 8) is associated with LPERF<4>. More than one bit may be set, allowing transactions from multiple nodes to be counted. This field applies to both LCNTR registers.
LC1_HLT	<23>	W, 0	**LCNTR1 Halt.** Write one to disable LCNTR1 counting.
LC1_RUN	<22>	W, 0	**LCNTR1 Run.** Write one to enable LCNTR1 counting.
RSVD	<21>	R0	**Reserved.** Reads as zero.
LC1_SEL	<20:16>	R/W, 0	**LCNTR1 Select.** Selects event for the LCNTR1 register.

LPERF <20:16>[1]	Event to Count in LCNTR1
00000	Our node LSB read due to RBlock (miss)
00001	Our node LSB R/W due to WBlock (miss/shd)
00010	Our node LSB write due to WBlock (shared)
00011	Our node LDxL requests (hit or miss)
00100	Our node STxC failures (hit, miss, or shared)
00101	Another node write this B-map hit
00110	Bank conflict delays—complex
00111	Arbitration losses—complex
01000	Other node LSB read, this VB hit (wrapped)
01001	Our node LSB READ_CSR
01010	Our node LSB WRT_CSR
01011	Our node LSB WRT_CSR to LMBPR (mailbx)
01100	Any node WRT_CSR to broadcast space
01101	Our node RBlock to Gbus space
01110	Our node WBlock to Gbus space
01111	LSB reads —subject to N_MASK<8:0>
10000	LSB writes —subject to N_MASK<8:0>
10001	Victim writes —subject to N_MASK<8:0>
10010	LSB cycles with STALL asserted
10011	Total memory latency—complex
10100	Carry out from LCNTR0

[1] When counting victim buffer hits, LC1_SEL = 01000 counts only hits for which LEVI has to swap (or wrap) the first and second hexwords to satisfy the read request in the proper order. LC0_SEL = 01000 counts all (wrapped or unwrapped) hits.

Table 9-22 LPERF Register Pass 3 LEVI Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td></td><td></td><td></td><td>
<table>
<tr><th>LPERF
&lt;20:16&gt;</th><th>Event to Count in LCNTR1[1]</th></tr>
<tr><td>10101</td><td>All LSB cmds (subject to N_MASK<8:0&gt;)</td></tr>
<tr><td>10110</td><td>LSB cycles without STALL asserted</td></tr>
<tr><td>10111</td><td>LSB cycles (all)</td></tr>
<tr><td>11000</td><td>Mem-space RBlk assert time (RBlk latency)</td></tr>
<tr><td>11001</td><td>Mem-space WBlk assert time (WBlk latency)</td></tr>
<tr><td>11010</td><td>Mem-space LDxL assert time (LDxL latency)</td></tr>
<tr><td>11011</td><td>Mem-space STxC assert time (STxC latency)</td></tr>
<tr><td>11100</td><td>Our node LSB READ due to LDxL (miss)</td></tr>
<tr><td>11101</td><td>Our node LSB R/W due to STxC (miss o shrd)</td></tr>
<tr><td>11110</td><td>Our node STxC requests (hit, miss, shrd, fail)</td></tr>
<tr><td>11111</td><td>Our node MB requests (Barrier)</td></tr>
</table>
[1] When counting victim buffer hits, LC1_SEL = 01000 counts only hits for which LEVI has to swap (or wrap) the first and second hexwords to satisfy the read request in the proper order. LC0_SEL = 01000 counts all (wrapped or unwrapped) hits.
</td></tr>
<tr><td>LC0_HLT</td><td>&lt;15&gt;</td><td>W, 0</td><td><b>LCNTR0 Halt.</b> Writing one disables LCNTR0 counting.</td></tr>
<tr><td>LC0_RUN</td><td>&lt;14&gt;</td><td>W, 0</td><td><b>LCNTR0 Run.</b> Writing one enables LCNTR0 counting.</td></tr>
<tr><td>RSVD</td><td>&lt;13&gt;</td><td>R0</td><td><b>Reserved.</b> Reads as zero.</td></tr>
</table>

Table 9-22 LPERF Register Pass 3 LEVI Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>LC0_SEL</td><td><12:8></td><td>R/W, 0</td><td>**LCNTR0 Select.** Selects event for the LCNTR0 register.
<table>
<tr><th>LPERF
<12:8>[1]</th><th>Event to Count in LCNTR0</th></tr>
<tr><td>00000</td><td>Our node LSB read due to RBlock (miss)</td></tr>
<tr><td>00001</td><td>Our node LSB R/W due to WBlock (miss/shd)</td></tr>
<tr><td>00010</td><td>Our node LSB write due to WBlock (shared)</td></tr>
<tr><td>00011</td><td>Our node LDxL requests (hit or miss)</td></tr>
<tr><td>00100</td><td>Our node STxC failures (hit, miss, or shared)</td></tr>
<tr><td>00101</td><td>Another node write this B-map hit</td></tr>
<tr><td>00110</td><td>Bank conflict delays—complex</td></tr>
<tr><td>00111</td><td>Arbitration losses—complex</td></tr>
<tr><td>01000</td><td>Other node LSB read, this VB hit</td></tr>
<tr><td>01001</td><td>Our node LSB READ_CSR</td></tr>
<tr><td>01010</td><td>Our node LSB WRT_CSR</td></tr>
<tr><td>01011</td><td>Our node LSB WRT_CSR to LMBPR</td></tr>
<tr><td>01100</td><td>(mailbx)</td></tr>
<tr><td>01101</td><td>Any node WRT_CSR to broadcast space</td></tr>
<tr><td>01110</td><td>Our node RBlock to Gbus space</td></tr>
<tr><td>01111</td><td>Our node WBlock to Gbus space</td></tr>
<tr><td>10000</td><td>LSB reads —from any node</td></tr>
<tr><td>10001</td><td>LSB writes —from any node</td></tr>
<tr><td>10010</td><td>Victim writes —from any node</td></tr>
<tr><td>10011</td><td>LSB cycles with STALL asserted</td></tr>
<tr><td>10100</td><td>Total memory latency—complex</td></tr>
<tr><td>10101</td><td>Carry out from LCNTR1</td></tr>
<tr><td>10110</td><td>All LSB cmds (from any node)</td></tr>
<tr><td>10111</td><td>LSB cycles without STALL asserted</td></tr>
<tr><td>11000</td><td>LSB cycles (all)</td></tr>
<tr><td>11001</td><td>Mem-space RBlk assert time (RBlk latency)</td></tr>
<tr><td>11010</td><td>Mem-space WBlk assert time (WBlk latency)</td></tr>
<tr><td>11011</td><td>Mem-space LDxL assert time (LDxL latency)</td></tr>
<tr><td>11100</td><td>Mem-space STxC assert time (STxC latency)</td></tr>
<tr><td>11101</td><td>Our node LSB READ due to LDxL (miss)</td></tr>
<tr><td>11110</td><td>Our node LSB R/W due to STxC (miss or shd)</td></tr>
<tr><td>11111</td><td>Our node STxC requests (hit, miss, shrd, fail)</td></tr>
<tr><td></td><td>Our node MB requests (Barrier)</td></tr>
</table>

[1] When counting victim buffer hits, LC1_SEL = 01000 counts only hits for which LEVI has to swap (or wrap) the first and second hexwords to satisfy the read request in the proper order. LC0_SEL = 01000 counts all (wrapped or unwrapped) hits.</td></tr>
</table>

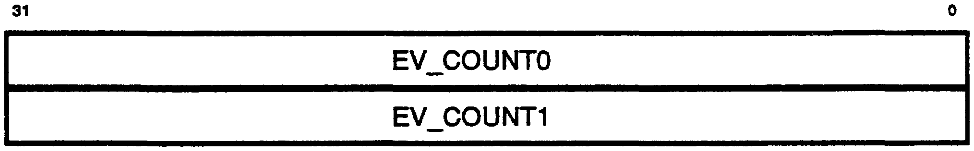
Table 9-22 LPERF Register Pass 3 LEVI Bit Definitions (Continued)

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>RSVD</td><td>&lt;7:5&gt;</td><td>R0</td><td><b>Reserved.</b> Read as zero.</td></tr>
<tr><td>N_MASK&lt;8&gt;</td><td>&lt;4&gt;</td><td>R/W, 0</td><td><b>Node Mask&lt;8&gt;.</b> When set, LCNTR1 counts node masked events from node 8, the IOP. This bit is an extension of the N_MASK field (LPERF<34:24&gt;).</td></tr>
<tr><td>MA_FREQ</td><td>&lt;3:2&gt;</td><td>R/W, 0</td><td><b>Miss Address Frequency.</b> Determines how often the LMISSADDR register loads the last miss address.
<table>
<tr><th>LPERF
&lt;3:2&gt;</th><th>Miss Frequency</th></tr>
<tr><td>00</td><td>Load every 32nd miss address</td></tr>
<tr><td>01</td><td>Load every 64th miss address</td></tr>
<tr><td>10</td><td>Load every miss address</td></tr>
<tr><td>11</td><td>Load every 256th miss address</td></tr>
</table></td></tr>
<tr><td>LC1_OVFL</td><td>&lt;1&gt;</td><td>R, 0</td><td><b>LCNTR1 Overflow.</b> Records overflow from the LCNTR1 register. Clears when the LCNTR1 register is reset.</td></tr>
<tr><td>LC0_OVFL</td><td>&lt;0&gt;</td><td>R, 0</td><td><b>LCNTR0 Overflow.</b> Records overflow from the LCNTR0 register. Clears when the LCNTR0 register is reset.</td></tr>
</table>

LCNTR0,1—Performance Counter Registers

Address BB + 0F40 and BB + 0F80
Access R

The LCNTR registers count events selected by the Select (LC0_SEL and LC1_SEL) and N_MASK fields of the LPERF register. Each register is a 32-bit counter with an associated overflow bit (LPERF<0> for LCNTR0 and LPERF<1> for LCNTR1). With the correct values of the Select fields, the LCNTR registers can be cascaded to form a single 64-bit register. The two counters are enabled and disabled independently through their associated LPERF control bits. The LCNTR registers can be read while the counters are stopped or running. The registers can also be stopped and restarted without resetting to zero. A write to an LCNTR register resets the counter and the associated overflow bit in the LPERF register, and sets the counter to the stopped state. Writes to LCNTR registers are ignored.



BXB-0228-92

Table 9-23 LCNTR Register Bit Definitions

Name	Bit(s)	Type	Function
EV_COUNT0	<31:0>	R, 0	**Event Count 0.** Number of events (selected through LPERF<LC0_SEL>) that occurred while LCNTR0 was enabled. Writes ignored.
EV_COUNT1	<31:0>	R, 0	**Event Count 1.** Number of events (selected through LPERF<LC1_SEL>) that occurred while LCNTR1 was enabled. Writes ignored.

LMISSADDR—Last Miss Address Register

Address BB + 0FC0
Access R/W

The LMISSADDR register captures every *n*th B-cache miss address determined by LPERF<MA_FREQ>. The miss may be due to a read, a write, or a shared block.

NOTE: Pass 1 or 2 and Pass 3 LEVI bit definitions of the LMISSADDR register are given in separate tables. See LMODE<LEVI_REV> for the LEVI revision.

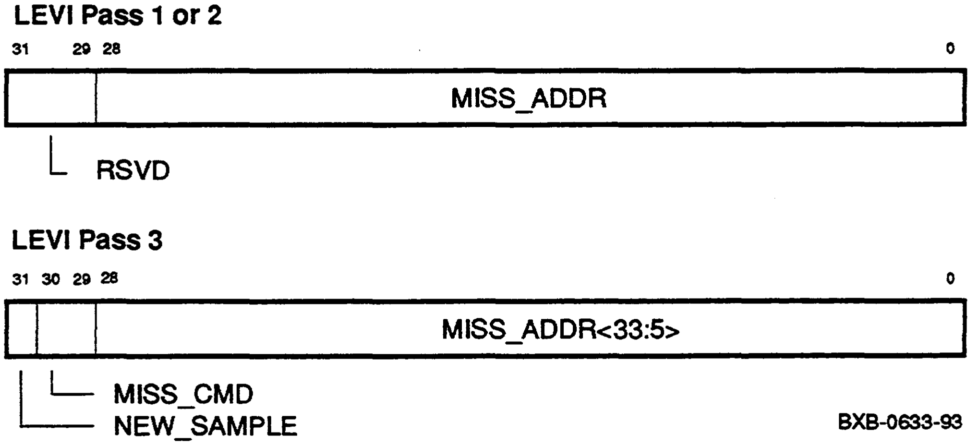


Table 9-24 LMISSADDR Register Pass 1 and Pass 2 LEVI Bit Definitions

Name	Bit(s)	Type	Function
RSVD	<31:29>	R0	**Reserved.** Read as zero.
MISS_ADDR	<28:0>	R/W, 0	**Missed Address.** Block address of one of the last *n* B-cache misses. LPERF<MA_FREQ> determines the value of *n*.

Table 9-25 LMISSADDR Register Pass 3 LEVI Bit Definitions

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Type</th><th>Function</th></tr>
<tr><td>NEW_SAMPLE</td><td><31></td><td>R0</td><td><b>New Sample.</b> Set every time a new address is loaded into the LMISSADDR register; cleared after the register is read.</td></tr>
<tr><td>MISS_CMD</td><td><30:29></td><td>R</td><td><b>Miss Command.</b> Indicates how the B-cache miss was generated.
<table>
<tr><th>LMISSADDR
<30:29></th><th>Miss Command</th></tr>
<tr><td>00</td><td>I-stream read miss: LSB read due to an I-stream RBlock or LDxL.</td></tr>
<tr><td>01</td><td>D-stream read miss: LSB read due to a D-stream RBlock or LDxL.</td></tr>
<tr><td>10</td><td>Write miss: LSB read due to WBlock or STxC.</td></tr>
<tr><td>11</td><td>Write to shared block: LSB write due to WBlock or STxC.</td></tr>
</table></td></tr>
<tr><td>MISS_ADDR</td><td><28:0></td><td>R</td><td>Block address of one of the last <i>n</i> B-cache misses. LPERF<MA_FREQ> determines the value of <i>n</i>.</td></tr>
</table>

Chapter 10

Privileged Architecture Library Code

This chapter describes the DECchip 21064 privileged architecture library code (PALcode). It covers the following topics:

- PALcode
- PALmode Environment
- Invoking PALcode
- PALcode Entry Points
- PALmode Restrictions
- Implementation of Architecturally Reserved Opcodes

10.1 PALcode

The Alpha AXP architecture defines an innovative feature called PALcode that allows many different physical implementations to coexist, each one adhering to the same programming interface specification. PALcode has characteristics that make it appear to be a combination of microcode, ROM BIOS, and system service routines, though the analogy to any of these other items is not exact. PALcode exists for several major reasons:

- There are some necessary support functions that are too complex to implement directly in a processor chip's hardware, yet cannot be handled by a normal operating system software routine. Routines to fill the translation buffer, acknowledge interrupts, and dispatch exceptions are some examples. In some architectures, these functions are handled by microcode, but the Alpha AXP architecture is careful not to mandate the use of microcode for reasonable chip implementations.
- There are functions that must run atomically, yet involve long sequences of instructions that may need complete access to all the underlying computer hardware. An example of this is the sequence that returns from an exception or interrupt.
- There are some instructions that are necessary for backward compatibility or ease of programming; however, these are not used often enough to dedicate them to hardware, or are so complex that they would jeopardize the overall performance of the computer. For example, an instruction that does a VAX-style interlocked memory access might be familiar to someone used to programming on a CISC machine, but is not included in the Alpha AXP architecture. Another example is the emulation of an instruction that has no direct hardware support in a particular chip implementation.

In each of these cases, PALcode routines are used to provide the function. The routines are nothing more than programs invoked at specified times, and read in as I-stream code in the same way that all other Alpha AXP code is read. Once invoked, however, PALcode runs in a special mode.

10.2 PALmode Environment

PALcode runs in a special environment called PALmode, defined as follows:

- I-stream memory mapping is disabled. Because the PALcode is used to implement translation buffer fill routines, I-stream mapping clearly cannot be enabled.
- The program has privileged access to all of the computer hardware. Most of the functions handled by PALcode are privileged and need control of the lowest levels of the system.
- Interrupts are disabled. If a long sequence of instructions needs to be executed atomically, interrupts cannot be allowed.

One important aspect of PALcode is that it uses normal Alpha AXP instructions for most of its operations; that is, the same instruction set that nonprivileged Alpha AXP programmers use. There are a few extra instructions that are available only in PALmode which will cause an OPCDEC exception (see Table <pal_entry_points>) if attempted while not in PALmode. The Alpha AXP architecture allows some flexibility in what these special PALmode instructions do. On the DECchip 21064 the special PALmode-only instructions perform the following functions:

- Read or write internal processor registers (HW_MFPR, HW_MTPR)
- Perform memory load or store operations without invoking the normal memory management routines (HW_LD, HW_ST)
- Return from an exception or interrupt (HW_REI).

Refer to Section 10.6 for detailed information on these special PALmode instructions.

When executing in PALmode, there are certain restrictions for using the privileged instructions because PALmode gives the programmer complete access to many of the internal details of the DECchip 21064.

CAUTION: It is possible to cause unintended side effects by writing what appears to be perfectly acceptable PALcode. As such, PALcode is not something that many users will want to change.

Refer to Section 10.5 for additional information on PALmode restrictions.

10.3 Invoking PALcode

PALcode is invoked at specific entry points, under certain well-defined conditions. PALcode can be thought of as a series of callable routines, with each routine indexed as an offset from a base address. The base address of the PALcode is programmable (stored in the PAL_BASE IPR) and is normally set by the system reset code.

When an event occurs that needs to invoke PALcode, the DECchip 21064 first drains the pipeline. The current PC is loaded into the EXC_ADDR

IPR, and the appropriate PALcode routine is dispatched. These operations occur under direct control of the chip hardware. The machine is now operatng in PALmode. When the HW_REI instruction is executed at the end of the PALcode routine, the hardware executes a jump to the address contained in the EX_ADDR IPR. The least significant bit is used to indicate PALmode to the hardware. Generally, upon return from a PALcode routine, the least significant bit is clear, in which case the hardware will load the new PC, enables interrupts, enables memory mapping, and dispatches back to the user.

10.3.1 Categories of Hardware-Initiated PALcode

The most basic use of PALcode is to handle complex hardware events. PALcode is called automatically when the particular hardware event is sensed. This use of PALcode is similar to other architectures' use of microcode. There are several major categories of hardware-initiated PALcode:

- When the DECchip 21064 is reset, it enters PALmode and executes the RESET PALcode. The system remains in PALmode until an HW_REI instruction is executed and EXC_ADDR<0> is cleared. It then continues execution in non-PALmode (native mode), as just described. It is during this initial RESET PALcode execution that the rest of the low level system initialization is performed, including any modification to the PAL_BASE IPR.
- When a system hardware error is detected by the DECchip 21064, the DECchip invokes one of several PALcode routines, depending upon the type of error. Errors such as machine checks, exceptions, reserved or privileged instruction decode, and data fetch errors are handled in this manner.
- When the DECchip 21064 senses an interrupt, it dispatches the acknowledgment of the interrupt to a PALcode routine that does the necessary information gathering, then handles the situation appropriately for the given interrupt.
- When a D-stream or I-stream translation buffer miss occurs, one of several PALcode routines is called to perform the TB fill. The memory management algorithms or even the existence of a virtual to physical page mapping is flexible. In the simplest case, this could be an automatic one-to-one translation from virtual to physical address. On a normal operating system these routines would consult page tables and perform the translation and fill based upon the PTE contents.

These elements are all very basic hardware-related functions, and would be difficult to implement efficiently using normal operating system service routines.

10.3.2 CALL_PAL Instruction

The other mechanism used to invoke PALcode is the CALL_PAL instruction. This is a special instruction that dispatches to PALcode at a specific entry point using the same set of steps as the hardware-activated PALcode. That is, the pipeline is drained, the PC is saved, and the appropriate dispatch to an offset from the PALcode base is performed. The only difference is that the dispatch is controlled by the program through an instruction, rather than through a hardware event or error. Also, PAL_CALL instructions place PC + 4 in the EXC_ADDR IPR.

The CALL_PAL instruction format includes a single parameter, the function field, that defines which CALL_PAL routine to invoke. Only a subset of all the possible CALL_PAL function values are supported with hardware dispatches in the DECchip 21064. These dispatches are described in Section 10.4. CALL_PAL routines can perform different functions for different operating systems running on the DECchip 21064. Unlike the basic hardware-generated PALcode, the CALL_PAL operations are largely optional and based upon what the system implementation needs.

There is a subtle difference between the two basic uses of PALcode: hardware-dispatched and CALL_PAL-dispatched. The hardware-invoked PALcode functions are necessary in some form for almost any useful computer system. For example, when the DECchip 21064 detects a serious system error, it will dispatch to the machine check (MCHK) PALcode entry point. The exact PALcode that resides at this entry point can do whatever is reasonable, based upon system needs.

The CALL_PAL instruction is totally under the control of the executing program for dispatch. If the program never executes one of the instructions that is included in the CALL_PAL list, then none of that PALcode will ever be run. Even here, the PALcode that does run once invoked, is executing in PALmode and is under the same restrictions as the hardware-activated PALcode.

The DECchip 21064 supports hardware dispatch for both privileged and nonprivileged CALL_PAL instructions. That is, some of the functions that are passed to the CALL_PAL instruction are considered special. The designation of privileged or nonprivileged refers to whether the user can call that particular CALL_PAL, and not the mode that it eventually runs in. Without exception, every CALL_PAL instruction will dispatch to PALcode that runs in PALmode. Only kernel users can call privileged CALL_PAL instructions.

The difference between privileged and nonprivileged CALL_PAL instructions is that the former can only be executed in kernel mode. Otherwise, they are vectored to offset 13E0 (OPCDEC) from the PAL_BASE IPR.

These are both CALL_PAL instructions, dispatched in exactly the same way, and when executed enter PALmode, do their function, and return to the user. The only difference is that before execution, a check is made to determine if the user is in the correct mode. If a nonkernel mode user attempts to execute a privileged CALL_PAL instruction, an OPCDEC PALcode routine is run instead of the CALL_PAL function. In addition, if a CALL_PAL function code that is not supported by the DECchip 21064 hardware dispatch is attempted, an OPCDEC exception is taken.

10.4 PALcode Entry Points

PALcode entry points are prioritized. Table <pal_entry_points> lists the entry points from the highest priority (first row, RESET) to the lowest. The table indicates only the entry point offset, bits <13:0>. The high-order bits of the new PC (bits <33:14>) are provided by the PAL_BASE IPR. The value in this IPR at power-up is zero.

NOTE: PALcode at entry points of higher priority than DTB_MISS must unlock possible MMCSR IPR and VA IPR locks.

Table 10-1 PALcode Entry Points

Entry Name	Offset From PAL_BASE IPR	Cause
RESET	0000	
MCHK	0020	Uncorrected hardware error.
ARITH	0060	Arithmetic exception.
INTERRUPT	00E0	Includes corrected hardware error.
D-stream errors	01E0, 08E0 09E0, 11E0	See Table 10-2.
ITB MISS	03E0	ITB miss.
ITB_ACV	03E0	I-stream access violation.
CALL_PAL	2000, 2040, 2080, 20C0 through 3FC0	128 locations based on instruction 7, 5..0. See the next table entry.
OPCDEC	13E0	Reserved or privileged opcode. Reserved opcodes are listed in Table 4-2 and marked RSVD. The privileged opcodes include both the HW_x instructions and the privileged CAL_PALL instructions. Any attempt to issue a privileged instruction while the processor is not in kernel mode (PS<CM1:CM0> is not equal to zero) causes a trap to the OPCDEC exception.
FEN	17E0	Floating-point operation attempted with: 1. FP instructions disabled by way of ICCSR<FPE> 2. FP IEEE round to +/– infinity 3. FP IEEE with datatype field other than S,T,Q

PALcode functions are implemented by way of the CALL_PAL instruction. CALL_PAL instructions cause exceptions in the hardware. As with all exceptions, the EXC_ADDR IPR is loaded by hardware with a possible return address.

CALL_PAL exceptions do not load the EXC_ADDR IPR with the address of the CALL_PAL instruction. They load the EXC_ADDR IPR with the address of the instruction following the CALL_PAL. PALcode supporting the desired PALmode function need not increment the EXC_ADDR IPR before executing an HW_REI instruction to return to native mode. This feature requires special handling in the arithmetic trap and machine check PALcode flows. See the description of the EXC_ADDR IPR for more complete information.

To improve speed of execution, a limited number of CALL_PAL instructions are directly supported in hardware with dispatches to specific address offsets.

The DECchip 21064 provides the first 64 privileged and 64 unprivileged CALL_PAL instructions with regions of 64 bytes. This produces hardware PALcode entry points described as follows:

```
Privileged CALL_PAL Instructions [00000000:0000003F]

Offset(Hex) = 2000 + ([5:0] shift left 6)

Unprivileged CALL_PAL Instructions [00000080:000000BF]

Offset(Hex) = 3000 + ([5:0] shift left 6)
```

The CALL_PAL instructions that do not fall within the ranges [0000:003F] and [0080:00BF] result in an OPCDEC exception.

CALL_PAL instructions that fall within the range [00000000:0000003F] while the DECchip 21064 is not executing in kernel mode will result in an OPCDEC exception.

The hardware recognizes four classes of D-stream memory management errors.

- Bad virtual address (incorrect sign extension)
- DTB_MISS
- Alignment error
- ACV, FOR, FOW

The following errors get mapped into four PALcode entry points:

- UNALIGN
- DTB_MISS PALmode
- DTB_MISS native mode
- D_FAULT

Table 10-2 shows the priorities of these entry points with respect to each other. A particular D-stream memory reference may generate errors that fall into more than one of the four error classes that the hardware recognizes.

Table 10-2 D-Stream Error PALcode Entry Points

BAD_VA	DTB_MISS	UNALIGN	PAL	Other	Offset (Hex)
1	x	1	x	x	11E0 UNALIGN
1	x	0	x	x	01E0 D_FAULT
0	1	x	1	x	09E0 DTB_MISS PAL
0	1	x	0	x	08E0 DTB_MISS Native
0	0	1	x	x	11E0 UNALIGN
0	0	0	x	1	01E0 D_FAULT

10.5 PALmode Restrictions

Many of the PALmode restrictions involve waiting *n* cycles before using the results of a PALcode instruction. Inserting *n* instructions between the two time-sensitive instructions is the typical method of waiting for *n* cycles. Because the DECchip 21064 can dual-issue instructions, it is possible to write code that requires 2**n* + 1 instructions to wait *n* cycles. Due to the resource requirements of individual instructions and the DECchip 21064 hardware design, multiple copies of the same instruction cannot be dual issued. This is used in some of the following examples. Explanations of PALmode restrictions follow:

- As a general rule, HW_MTPR instructions require at least four cycles to update the selected IPR. At least three cycles of delay must be inserted before using the result of the register update.

  The following instructions will pipeline correctly and do not require software timing except for accesses of the TB IPRs:

  — Multiple reads

  — Multiple writes

  — Read followed by write

  These cycles can be guaranteed by either including seven instructions, which do not use the IPR in transition, or proving through the dual-issue rules and/or state of the machine, that at least three cycles of delay will occur. Multiple copies of a HW_MTPR instruction (used as a no-op instruction) can be used to pad cycles after the original HW_MTPR. Multiple copies of the same instruction will never dual issue. Because of this, the maximum number of instructions necessary to ensure at least three cycles of delay is three, as shown in Example 10-1.

Example 10-1 Code for a Delay of Three Cycles

```
HW_MTPR Rx, HIER        ; Write to HIER
HW_MFPR R31, 0          ; NOP mxpr instruction
HW_MFPR R31, 0          ; NOP mxpr instruction
HW_MFPR R31, 0          ; NOP mxpr instruction
HW_MFPR Ry, HIER        ; Read from HIER
```

The HW_REI instruction uses the ITB if the EXC_ADDR IPR contains a nonPALmode VPC (VPC<0> = 0). By the previous rule, it is implied that at least 3 cycles of delay must be included after writing the ITB before executing a HW_REI instruction to exit PALmode.

The following are exceptions to the general rule:

— HW_MFPR instructions reading a PAL_TEMP IPR can never occur exactly two cycles after an HW_MTPR instruction writing a PAL_TEMP IPR. The solution results in code shown in Example 10-2.

Example 10-2 Reading PAL_TEMP After a Write to PAL_TEMP

```
HW_MTPR Rx, PAL_R0      ; Write PAL temp [0]
HW_MFPR R31, 0          ; NOP mxpr instruction
HW_MFPR R31, 0          ; NOP mxpr instruction
HW_MFPR R31, 0          ; NOP mxpr instruction
HW_MFPR Ry, PAL_R0      ; Read PAL temp [0]
```

This code guarantees three cycles of delay after the write before the read. It is also possible to make use of the cycle immediately following an HW_MTPR instruction to execute an HW_MFPR instruction to the same (accomplishing a swap) or a different PAL_TEMP IPR. The swap operation only occurs if the HW_MFPR instruction immediately follows the HW_MTPR. This timing requires great care and knowledge of the pipeline to ensure that the second instruction does not stall for one or more cycles. Use of the slot to accomplish a read from a different PAL_TEMP IPR requires that the second instruction will not stall for exactly one cycle. This is much easier to insure. An HW_MFPR instruction can stall for a single cycle as a result of a write-after-write conflict.

— The EXC_ADDR IPR can be read by an HW_REI instruction only two cycles after the HW_MTPR. This is equivalent to one intervening cycle of delay. This translates to code shown in Example 10-3.

Example 10-3 Reading the EXC_ADDR IPR

```
HW_MTPR Rx, EXC_ADDR    ; Write EXC_ADDR
HW_MFPR R31, 0    ; NOP cannot dual issue with either
HW_REI            ; Return
```

- An HW_MTPR operation to the DTBIS IPR cannot be sourced from bypassed path. All data being moved to the DTBIS IPR must be sourced directly from the register file. One way to ensure this is to provide at least three cycles of delay before using the result of any integer operation (except MUL) as the source of an HW_MTPR DTBIS. This is shown in Example 10-4.

NOTE: Note: MUL should not be used as a source of DTBIS data.

Example 10-4 Using Result of Integer Operation as Source of HW_MTPR DTBIS

```
ADDQ R1,R2,R3        ; Source for DTBIS address
ADDQ R31,R31,R31     ; Cannot dual issue with above,
                     ; 1st cycle of delay
ADDQ R31,R31,R31     ; 2nd cycle of delay
ADDQ R31,R31,R31     ; 3rd cycle of delay
ADDQ R31,R31,R31     ; May dual issue with below, else
                     ; 4th cycle of delay
HW_MTPR R3,DTBIS     ; R3 must be in register file, no
                     ; bypass possible
```

- At least one cycle of delay must occur after an HW_MTPR TB_CTL before an HW_MTPR ITB_PTE or an HW_MFPR ITB_PTE. This must be done to allow setup of the ITB large page or small page decode.
- The first cycle (the first one or two instructions) at all PALcode entry points cannot execute a conditional branch instruction or any other instruction that uses the JSR stack hardware. This includes the following instructions:
  - — JSR
  - — JMP
  - — RET
  - — JSR_COROUTINE
  - — BSR
  - — HW_REI
  - — All box opcode except BR
- Table 10-3 lists the number of cycles required after an HW_MTPR instruction before a subsequent HW_REI instruction for the specified IPRs. These cycles can be ensured by inserting one HW_MFPR R31,0 instruction or other appropriate instruction(s) for each cycle of delay required after the HW_MTPR.

Table 10-3 HW_MTPR Restrictions

IPR	Cycles Between HW_MTPR and HW_REI
DTBIS, ASM, ZAP	0
ITBIS, ASM, ZAP	2
xIER	3
xIRR	3
ICCSR<FPE>	4
PS	4

- When loading the CC IPR, bits <3:0> must be loaded with zero. Loading nonzero values in these bits can cause an inaccurate count.

- An HW_MTPR DTBIS cannot be combined with an HW_MTPR ITBIS instruction. The hardware will not clear the ITB if both the Ibox and Abox IPRs are simultaneously selected. Two instructions are needed to clear each TB individually, as shown in Example 10-5.

Example 10-5 Clearing the ITB and DTB

```
HW_MTPR Rx,ITBIS
HW_MTPR Ry,DTBIS
```

- Three cycles of delay are required between:
  - HW_MTPR xIER and HW_MFPR xIRR
  - HW_MTPR xIRR and HW_MFPR xIRR
  - MTPS and LD or ST
  - MTPS and HW_MFPR xIRR
  - HW_MTPR ALT_MODE and HW_LD/HW_ST ALT_MODE
- The following operations are disabled in the cycle immediately following an HW_REI instruction:
  - HW_MxPR ITB_TAG
  - HW_MxPR ITB_PTE
  - HW_MxPR ITB_PTE_TEMP

  This rule implies that it is not a good idea to ever allow exceptions while updating the ITB. The ITB IPR will not be written if:

  - An exception interrupts flow of the ITB miss routine and attempts to REI back.
  - The return address begins with an HW_MxPR instruction to an ITB IPR.
  - The REI is predicted correctly to avoid any delay between the two instructions.

  Example 10-6 shows the code for this operation.

Example 10-6 Write to ITB Ignored Following REI

```
HW_REI               ; return from interrupt
HW_MTPR R1,ITB_TAG   ; attempts to execute very next
                     ; cycle, instr ignored
```

- The following registers can only be accessed in PALmode:
  - ITB_TAG
  - ITB_PTE
  - ITB_PTE_TEMP

  If the instruction HW_MTPR or HW_MFPR is applied to these IPRs while not in PALmode, the instruction will be ignored even if ICCSR<HWE> is set.

- When writing the PAL_BASE IPR, exceptions cannot occur. An exception occurring simultaneously with a write to the PAL_BASE IPR can leave the register in a metastable state. All asynchronous exceptions but reset can be avoided under conditions shown in Example 10-7.

Example 10-7 Conditions for Avoiding Asynchronous Exceptions

```
PALmode ................... blocks all interrupts
machine checks disabled ..... blocks I/O error exceptions
  (by way of the ABOX_CTL reg or MB isolation)
Not under trap shadow ....... avoids arithmetic traps

The trap shadow is defined as:

less than  3 cycles after a non-mul integer operate that
   may overflow
less than 22 cycles after a MULL/V instruction
less than 24 cycles after a MULQ/V instruction
less than  6 cycles after a non-div fp operation that may
   causea trap
less than 34 cycles after a DIVF or DIVS that may cause a
   trap
less than 63 cycles after a DIVG or DIVT that may cause a
   trap
```

- The sequence HW_MTPR PTE, HW_MTPR TAG is not allowed. At least two null cycles must occur between HW_MTPR PTE and HW_MTPR TAG.
- The MCHK exception service routine must check the EXC_SUM IPR for simultaneous arithmetic errors. Arithmetic traps will not trigger exceptions a second time after returning from exception service for the machine check.
- Three cycles of delay must be inserted between HW_MFPR DTB_PTE and HW_MFPR DTB_PTE_TEMP as shown in Example 10-8.

Example 10-8 Delay Between HW_MFPR DTB_PTE and HW_MFPR DTB_PTE_TEMP

```
HW_MFPR Rx,DTB_PTE      ; reads DTB_PTE into DTB_PTE_TEMP
                        ; IPR
HW_MFPR R31,0           ; 1st cycle of delay
HW_MFPR R31,0           ; 2nd cycle of delay
HW_MFPR R31,0           ; 3rd cycle of delay
HW_MFPR Ry,DTB_PTE_TEMP ; read DTB_PTE_TEMP into register
                        ; file Ry
```

- Three cycles of delay must be inserted between HW_MFPR ITB_PTE and HW_MFPR ITB_PTE_TEMP, as shown in Example 10-9.

Example 10-9 Delay Between HW_MFPR ITB_PTE and HW_MFPR ITB_PTE_TEMP

```
HW_MFPR Rx,DTB_PTE      ; reads DTB_PTE into DTB_PTE_TEMP
                        ; IPR
HW_MFPR R31,0           ; 1st cycle of delay
HW_MFPR R31,0           ; 2nd cycle of delay
HW_MFPR R31,0           ; 3rd cycle of delay
HW_MFPR Ry,DTB_PTE_TEMP ; read DTB_PTE_TEMP into register
                        ; file Ry
```

- The content of the destination register for HW_MFPR Rx, DTB_PTE or HW_MFPR Rx, ITB_PTE is UNPREDICTABLE.
- Two HW_MFPR DTB_PTE instructions cannot be issued in consecutive cycles. This implies that more than one instruction can be necessary between the HW_MFPR instructions if dual issue is possible. Similar restrictions apply to the ITB_PTE IPR.
- Reading the EXC_SUM and BC_TAG IPRs require special timing. Refer to Chapter 3 for specific information.
- DMM errors occurring one cycle before HW_MxPR instructions to the ITB_PTE will not stop the TB pointer from incrementing to the next TB entry even though the HW_MxPR instruction will be aborted by the DMM error. This restriction only affects performance and not functionality.
- PALcode that writes multiple ITB entries must write the entry that maps the address contained in the EXC_ADDR IPR last.
- HW_STC instructions cannot be followed, for two cycles, by any load instruction that may miss in the D-cache.
- Updates to the ASN field of the ICCSR IPR require at least 10 cycles of delay before entering native mode that can reference the ASN during I-cache access. If the ASN field is updated in kernel mode by way of the HWE bit of the ICCSR IPR, it is sufficient that all I-stream references during this time be made to pages with the ASM bit set to avoid use of the ASN.
- HW_MTPR instructions that update the TB_CTL IPR cannot follow an HW_MTPR instruction that updates the DTB_PTE or ITB_PTE IPR by one cycle.
- The HW_MTPR instructions that update the following IPRs require delays as shown in Table 10-4:

  — ICCSR (ASN field)

  — FLUSH_IC

  — FLUSH_IC_ASM

  The purpose of the delay is to ensure that the update occurs before the first instruction fetch in native mode, since the pipeline may currently contain instructions that were fetched before the update (which would remain valid during a pipeline stall). It is necessary that at least one instruction be issued during each cycle of the delay to ensure that the pipeline is cleared of all instructions fetched prior to the update.

If the update is performed in kernel mode through the use of the HWE bit of the ICCSR, it is sufficient that all I-stream references during this time be made to pages with the ASM bit set to avoid use of the ASN.

Table 10-4 HW_MTPR Cycle Delay

IPR	Cycle Delay
ICCSR (ASN field only)	8
FLUSH_IC	9
FLUSH_IC_ASM	9

- Machine check exceptions taken while in PALmode can load the EXC_ADDR IPR with a restart address one instruction earlier than the correct restart address. Some HW_MxPR instructions may have already completed execution even if the restart address indicates the HW_MxPR as the return instruction. Reexecution of some HW_MxPR instructions can alter the machine state TB pointers, EXC_ADDR IPR mask.

  The mechanism used to stop instruction flow during machine check exceptions causes the machine check exception to appear as a D-stream fault on the following instruction in the hardware pipeline. In the event that the following instruction is a HW_MxPR, a D-stream fault will not abort execution in all cases. The EXC_ADDR will be loaded with the address of the HW_MxPR instruction as if it were aborted. An HW_REI to this restart address will incorrectly reexecute this instruction.

  Machine check service routines should check for MxPR instructions at the return address before continuing.

10.5.1 TB Miss Flows

This section describes hardware-specific details to aid the PALcode programmer in writing ITB and DTB fill routines. These flows highlight the trade-offs and restrictions between PALcode and hardware. The PALcode source that is released with the DECchip 21064 should be consulted before any new flows are written. The discussions assume a working knowledge of the Alpha AXP memory management architecture (see Chapters 11 and 12).

10.5.1.1 ITB Miss

When the Ibox encounters an ITB miss it:

1. Latches the VPC of the target instruction-stream reference in the EXC_ADDR IPR.
2. Flushes the pipeline of any instructions following the instruction which caused the ITB miss.
3. Waits for any other instructions that may be in progress to complete.
4. Enters PALmode.

5. Jumps to the ITB miss PALcode entry point.

The recommended PALcode sequence for translating the address and filling the ITB is as follows:

1. Create some scratch area in the integer register file by writing the contents of a few integer registers to the PAL_TEMP register file.
2. Read the target virtual address from the EXC_ADDR IPR.
3. Fetch the PTE (this can take multiple reads) using a physical-mode HW_LD instruction. If this PTE's valid bit is clear, report TNV or ACV as appropriate.
4. The *Alpha Architecture Reference Manual* states that translation buffers cannot contain invalid PTEs; the PTE's valid bit must be explicitly checked by PALcode. Since the ITB's PTE RAM does not hold the FOE bit, the PALcode must also explicitly check this condition. If the PTE's valid bit is set and FOE bit is clear, PALcode can fill an ITB entry.
5. Write the original virtual address to the TB_TAG IPR using HW_MTPR. This writes the TAG into a PAL_TEMP IPR and not the actual tag field in the ITB.
6. Write the PTE to the TB_CTL to select between the large page or small page TB regions. Wait at least one cycle before executing the next step.
7. Write the PTE to the ITB_PTE IPR using HW_MTPR. This HW_MTPR causes both the TAG and PTE fields in the ITB to be written.

   NOTE: It is not necessary to delay issuing the HW_MTPR to the ITB_PTE after the MTPR to the ITB_TAG is issued.
8. Restore the contents of any modified integer registers to their original state using the HW_MFPR instruction.
9. Restart the instruction stream using the HW_REI instruction.

10.5.1.2 DTB Miss

When the Abox encounters a DTB miss, it:

1. Latches the referenced virtual address in the VA IPR and other information about the reference in the MMCSR IPR.

   Locks the VA and MMCSR IPR against further modifications.

   Latches the PC of the instruction that generated the reference in the EXC_ADDR IPR.
2. Drains the machine as described in Section 10.5.1.1.
3. Jumps to the DTB miss PALcode entry point.

Unlike ITB misses, DTB misses can occur while the CPU is executing in PALmode. The recommended PALcode sequence for translating the address and filling the DTB is as follows:

1. Create some scratch area in the integer register file by writing the contents of a few integer registers to the PAL_TEMP register file.

2. Read the requested virtual address from the VA IPR. The act of reading this register unlocks the VA and MMCSR IPRs. The MMCSR IPR is updated only when D-stream memory management errors occur. It will retain information about the instruction that generated the DTB miss. This can be useful later.
3. Fetch the PTE. This operation can require multiple reads. If the Valid bit of the PTE is clear, a Translation Not Valid (TNV) or Access Violation (ACV) must be reported unless the instruction which caused the DTB miss was FETCH or FETCH_M. This can be checked by way of the opcode field of the MMCSR IPR. If the value in this field is 18 (hex), then a FETCH or FETCH_M instruction caused this DTB miss. As mandated in the *Alpha Architecture Reference Manual,* the subsequent TNV or ACV should not be reported. Therefore, PALcode should:
    a. Read the value in EXC_ADDR IPR
    b. Increment the value by four
    c. Write the value back to EXC_ADDR IPR
4. Write the register that holds the contents of the PTE to the DTB_CTL. This has the effect of selecting one of the four possible granularity hint sizes.
5. Write the original virtual address to the TB_TAG IPR. This writes the TAG into a PAL_TEMP IPR and not the actual tag field in the DTB.
6. Write the PTE to the DTB_PTE IPR. This HW_MTPR causes both the TAG and PTE fields in the DTB to be written.

    NOTE: It is not necessary to delay issuing the HW_MTPR to the DTB_PTE after the MTPR to the DTB_TAG is issued.
7. Restore the contents of any modified integer registers.
8. Restart the instruction stream using the HW_REI instruction.

10.6 Implementation of Architecturally Reserved Opcodes

PALcode uses the Alpha AXP instruction set for most of its operations. The DECchip 21064 maps the architecturally reserved opcodes (PAL19, PAL1B, PAL1D, PAL1E, and PAL1F) to:

- A move-to and a move-from processor register (HW_MTPR, HW_MFPR)
- A special load and store (HW_LD, HW_ST)
- A return from PALmode exception or interrupt (HW_REI)

These instructions produce an OPCDEC exception (see Table 10-1) if executed while not in the PALmode environment. If ICCSR<HWE> is set, these instructions can be executed in kernel mode.

Register checking and bypassing logic is provided for PALcode instructions as it is for nonPALcode instructions when using general purpose registers.

NOTE: Explicit software timing is required for accessing the hardware-specific IPRs and the PAL_TEMPs. These constraints are described in Section 10.5.

10.6.1 HW_MFPR and HW_MTPR Instructions

The IPR specified by the PAL, ABX, IBX, and index field is written/read with the data from the specified integer register.

CAUTION: Writing / reading IPRs can produce side effects.

Coding restrictions (see Section 10.5) are associated with accessing various registers. Separate bits are used to access the following:

Abox IPRs
Ibox IPRs
PAL_TEMPs

It is possible for an HW_MTPR instruction to write multiple registers in parallel if they both have the same index.

Figure 10-1 shows the format of the HW_MFPR and HW_MTPR instructions. Table 10-5 describes the HW_MFPR and HW_MTPR instruction fields.

Figure 10-1 HW_MFPR and HW_MTPR Instruction Format

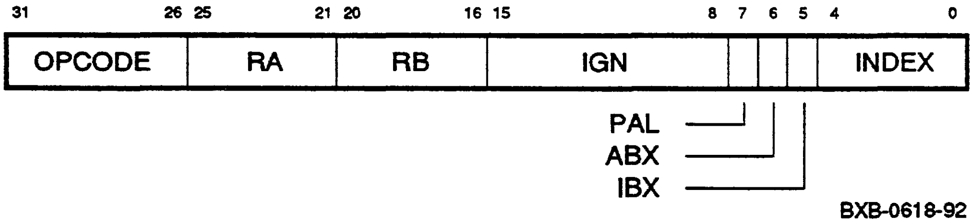


Table 10-5 HW_MFPR and HW_MTPR Field Descriptions

Field	Function
OPCODE	Is either 25 (HW_MFPR) or 29 (HW_MTPR).
RA/RB	Contain the source (HW_MTPR) or destination (HW_MFPR) IPR number. The RA and RB fields must always be identical.
PAL	If set, this HW_MFPR or HW_MTPR instruction is referencing a PAL temporary register, PAL_TEMP.
ABX	If set, this HW_MFPR or HW_MTPR instruction is referencing a register in the Abox.
IBX	If set, this HW_MFPR or HW_MTPR instruction is referencing a register in the Ibox.

10.6.2 HW_LD and HW_ST Instructions

PALcode uses the HW_LD and HW_ST instructions to access memory outside the realm of normal Alpha AXP memory management. The effective address of these instructions is calculated as follows:

```
addr <- (SEXT(DISP) + RB) AND NOT (QW | 11 (bin))
```

Figure 10-2 shows the format of the HW_LD and HW_ST instructions. Table 10-6 describes the fields of the HW_LD and HW_ST instruction fields.

Figure 10-2 HW_LD and HW_ST Instruction Format

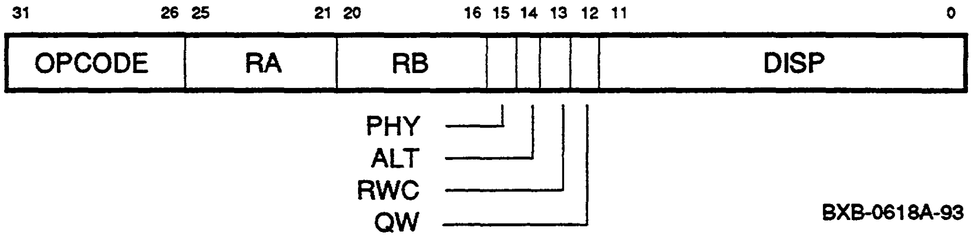


Table 10-6 HW_LD and HW_ST Instruction Field Descriptions

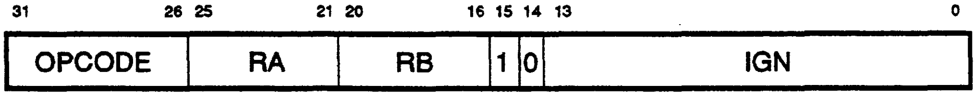
Field	Function
OPCODE	Is either 27 (HW_LD) or 31 (HW_ST).
RA/RB	Contain register numbers, interpreted in the normal fashion for loads and stores.
PHY	If clear, the effective address of the HW_LD or HW_ST is a virtual address. If set, then the effective address of the HW_LD or HW_ST is a physical address.
ALT	For virtual-mode HW_LD and HW_ST instructions, this bit selects the processor mode bits that are used for memory management checks. If ALT is clear, the current mode bits of the PS IPR are used; if ALT is set, the mode bits in the ALT_MODE IPR are used.
	Physical-mode load-lock and store-conditional variants of the HW_LD and HW_ST instructions may be created by setting both the PHY and ALT bits.
RWC	The RWC (read-with-write check) bit, if set, enables both read and write access checks on virtual HW_LD instructions.
QW	The quadword bit specifies the data length. If it is set, the length is quadword. If it is clear, the length is longword.
DISP	The DISP field holds a 12-bit signed byte displacement.

10.6.3 HW_REI Instruction

The HW_REI instruction uses the address in the Ibox EXC_ADDR IPR to determine the new virtual program counter (VPC). Bit <0> of the EXC_ADDR IPR indicates the state of the PALmode bit on the completion of the HW_REI. If EXC_ADDR<0> is set, then the processor remains in PALmode. This allows PALcode to transition from PALmode to non-PALmode. The HW_REI instruction can also be used to jump from PALmode to PALmode. This allows PAL instruction flows to take advantage of the D-stream mapping hardware in the DECchip 21064, including traps.

Figure 10-3 shows the format of the HW_REI instruction. Table 10-7 describes the HW_LD and HW_ST instruction fields.

Figure 10-3 HW_REI Instruction Format



BXB-0618B-93

Table 10-7 HW_REI Instruction Field Descriptions

Field	Function
OPCODE	Contains 30.
RA/RB	Contain register numbers, which should be R31. Otherwise, a stall will occur.

Positions <15,14> in the HW_REI instruction contain the branch prediction hint bits. The DECchip 21064 pushes the contents of the EXC_ADDR IPR on the JSR prediction stack. Bit <15> must be set to pop the stack to avoid misalignment.

The next address and PALmode bit are calculated as follows:

```
VPC <- EXC_ADDR AND {NOT 3}
PALmode <- EXC_ADDR[0]
```

Chapter 11

OpenVMS AXP System Support

This chapter discusses memory management performed by the OpenVMS AXP operating system and gives the structure of a process within the OpenVMS AXP environment. Consult the *Alpha Architecture Reference Manual* (hereafter referred to as *AARM* in this chapter) for a thorough discussion of these topics.

11.1 OpenVMS Memory Management

Memory management is the control of the allocation and use of physical memory. It is implemented by a combination of hardware and software. Typically, in a multiprogramming system, several processes may reside in physical memory at the same time. OpenVMS Alpha uses memory protection and multiple address spaces to ensure that one process will not affect either other processes or the operating system.

To improve further software reliability, four hierarchical access modes provide memory access control. They are, from most to least privileged: kernel, executive, supervisor, and user. Protection is specified at the individual page level, where a page may be inaccessible, read only, or read/write for each of the four access modes. Accessible pages can be restricted to have only data or instruction access.

A program uses virtual addresses to access its data and instructions. However, before these virtual addresses can be used to access memory, they must be translated into physical addresses. Memory management software maintains tables of mapping information (page tables) that keep track of where each virtual page is located in physical memory. The processor uses this mapping information when it translates virtual addresses to physical addresses.

Therefore, memory management provides both memory protection and memory mapping mechanisms. The OpenVMS Alpha memory management architecture is designed to meet several goals:

- Provide a large address space for instructions and data.
- Allow programs to run on hardware with physical memory smaller than the virtual memory used.
- Provide convenient and efficient sharing of instructions and data.
- Allow sparse use of a large address space without excessive page table overhead.
- Contribute to software reliability.

- Provide independent read and write access protection.

11.1.1 Virtual Address Space

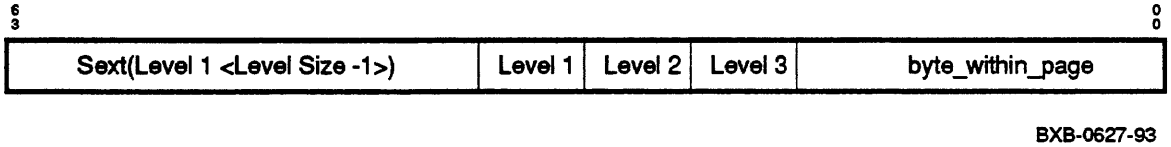
A virtual address is a 64-bit unsigned integer specifying a byte location within the virtual address space. Implementations support subsets of the address space in one of four sizes (43, 47, 51, or 55 bits) as a function of page size. The minimal virtual address size supported is 43 bits. If an implementation supports less than 64-bit virtual addresses, it must check that all the VA<63:vaSize> bits are equal to VA<vaSize−1>. This gives two disjoint ranges for valid virtual addresses. For example, for a 43-bit virtual address space valid virtual address ranges are 0 to 3FF FFFF FFFF and FFFF FC00 0000 0000 to FFFF FFFF FFFF FFFF. Accesses to virtual addresses outside the valid virtual address ranges for an implementation cause an access violation exception.

The virtual address space is broken into pages, which are the units of relocation, sharing, and protection. The page size ranges from 8 Kbytes to 64 Kbytes. System software should, therefore, allocate regions with differing protection on 64-Kbyte virtual address boundaries to ensure image compatibility across all Alpha implementations.

Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. The operating system controls the virtual-to-physical address mapping tables and saves the inactive parts of the virtual address space on external storage media.

The processor generates a 64-bit virtual address for each instruction and operand in memory. The virtual address consists of three level-number fields, and a byte_within_page field. Figure 11-1 shows the virtual address format.

Figure 11-1 Virtual Address Format



The byte_within_page field can be either 13, 14, 15, or 16 bits depending on a particular implementation. Thus, the allowable page sizes are 8 Kbytes, 16 Kbytes, 32 Kbytes, and 64 Kbytes. Each level-number field contains 0-*n* bits, where *n* is, for example, 9 with an 8-Kbyte page size. The level-number fields are the same size for a given implementation.

The level-number fields are a function of the page size; all page table entries at any given level do not exceed one page. The PFN field in the PTE is always 32 bits wide. Thus, as the page size grows, the virtual and physical address size also grows.

Table 11-1 Virtual Address Options

Page Size (Kbytes)	Byte Offset (Bits)	Level Size (Bits)	Virtual Address (Bits)	Physical Address (Bits)
8	13	10	43	45
16	14	11	47	46
32	15	12	51	47
64	16	13	55	48

11.1.2 Physical Address Space

Physical addresses are at most 48 bits. A processor may choose to implement a smaller physical address space by not implementing some number of high order bits. The two most significant implemented physical address bits select a caching policy or implementation dependent type of address space. Implementations will use these bits as appropriate for their systems. For example, in a workstation with a 30-bit physical address space, bit <29> might select between memory and non-memory like regions, and bit <28> could enable or disable caching.

11.1.3 Memory Management Control

Memory management is always enabled. Implementations must provide an environment for PALcode to service exceptions and to initialize and boot the processor. For example, PALcode might run with I-stream mapping disabled and use the privileged CALL_PAL LDQP and STQP instructions to access data stored in physical addresses.

11.1.4 Page Table Entries

The processor uses a quadword PTE (Figure 11-2) to translate virtual addresses to physical addresses. A PTE contains hardware and software control information and the physical page frame number (PFN). Fields in the page table entry are interpreted as shown in Table 11-2.

Figure 11-2 Page Table Entry

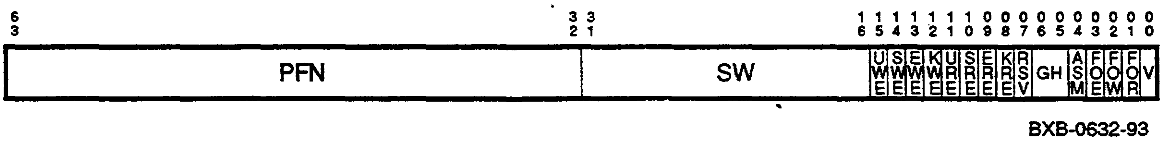


Table 11-2 Page Table Entry Bit Definitions

Name	Bit(s)	Function
PFN	<63:32>	**Page Frame Number.** The PFN field always points to a page boundary. If <V> is set, the PFN is concatenated with VA<byte_within_page> to obtain the physical address; see Section 11.1.7. If <V> is clear, this field may be used by software.
RSVD	<31:16>	**Reserved.** To be used by software.
UWE	<15>	**User Write Enable.** Enables writes from user mode. If this bit is a 0 and a STORE is attempted while in user mode, an access violation occurs. This bit is valid even when <V>=0. *NOTE: If a write enable bit is set and the corresponding read enable bit is not, the operation of the processor is UNDEFINED.*
SWE	<14>	**Supervisor Write Enable.** Enables writes from supervisor mode. If this bit is a 0 and a STORE is attempted while in supervisor mode, an access violation occurs. This bit is valid even when <V>=0.
EWE	<13>	**Executive Write Enable.** Enables writes from executive mode. If this bit is a 0 and a STORE is attempted while in executive mode, an access violation occurs. This bit is valid even when <V>=0.
KWE	<12>	**Kernel Write Enable.** Enables writes from kernel mode. If this bit is a 0 and a STORE is attempted while in kernel mode, an access violation occurs. This bit is valid even when <V>=0.
URE	<11>	**User Read Enable.** Enables reads from user mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in user mode, an access violation occurs. This bit is valid even when <V>=0.
SRE	<10>	**Supervisor Read Enable.** Enables reads from supervisor mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in supervisor mode, an access violation occurs. This bit is valid even when <V>=0.
ERE	<9>	**Executive Read Enable.** Enables reads from executive mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in executive mode, an access violation occurs. This bit is valid even when <V>=0.
KRE	<8>	**Kernel Read Enable.** Enables reads from kernel mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in kernel mode, an access violation occurs. This bit is valid even when <V>=0.
RSVD	<7>	**Reserved.** To be used by Digital in the future. *NOTE: This reserved bit will be used by future hardware systems and should not be used by software even if PTE<V> is clear.*

Table 11-2 Page Table Entry Bit Definitions (Continued)

Name	Bit(s)	Function
GH	<6:5>	**Granularity Hint.** Software may set these bits to a non-zero value to supply a hint to translation buffer implementations that a block of pages can be treated as a single larger page: • The block is an aligned group of 8^N pages, where N is the value of PTE<6:5>, for example, a group of 1, 8, 64, or 512 pages starting at a virtual address with page_size + 3*N low-order zeros. • The block is a group of physically contiguous pages that are aligned both virtually and physically. Within the block, the low 3*N bits of the PFNs describe the identity mapping, and the high 32-3*N PFN bits are all equal. • Within the block, all PTEs have the same values for bits <15:0>, that is, protection, fault, granularity, and valid bits. Hardware may use this hint to map the entire block with a single TB entry instead of 8, 64, or 512 separate TB entries. Note that it is UNPREDICTABLE which PTE values within the block are used if the granularity bits are set inconsistently. *NOTE: A granularity hint might be appropriate for a large memory structure such as a frame buffer or nonpaged pool that in fact is mapped into contiguous virtual pages with identical protection, fault, and valid bits.*
ASM	<4>	**Address Space Match.** When set, this PTE matches all Address Space Numbers. For a given VA, ASM must be set consistently in all processes, otherwise the address mapping is UNPREDICTABLE.
FOE	<3>	**Fault On Execute.** When set, a fault on execute exception occurs on an attempt to execute an instruction in the page.
FOW	<2>	**Fault On Write.** When set, a fault on write exception occurs on an attempt to write any location in the page.
FOR	<1>	**Fault On Read.** When set, a fault on read exception occurs on an attempt to read any location in the page.
V	<0>	**Valid.** Indicates the validity of the the PFN field. When <V> is set, the PFN field is valid for use by hardware. If <V> is clear, the PFN field is reserved for use by software. <V> does not affect the validity of PTE<15:1>.

11.1.5 Changes to Page Table Entries

The operating system changes PTEs as part of its memory management functions. For example, the operating system may set or clear the valid bit, change the PFN field as pages are moved to and from external storage media, or modify the software bits. The processor hardware never changes PTEs.

Software must guarantee that each PTE is always consistent within itself. Changing a PTE one field at a time may give incorrect system operation, for example, setting PTE<V> with one instruction before establishing

PTE<PFN> with another. Execution of an interrupt service routine between the two instructions could use an address that would map using the inconsistent PTE. Software can solve this problem by building a complete new PTE in a register and then moving the new PTE to the page table using a Store Quadword instruction (STQ).

Multiprocessing makes the problem more complicated. Another processor could be reading (or even changing) the same PTE that the first processor is changing. Such concurrent access must produce consistent results. Software must use some form of software synchronization to modify PTEs that are already valid. Once a processor has modified a valid PTE, it is possible that other processors in a multiprocessor system may have old copies of that PTE in their translation buffer. Software must inform other processors of changes to PTEs.

Software may write new values into invalid PTEs using STQ instructions. Hardware must ensure that aligned quadword reads and writes are atomic operations. The following procedure must be used to change any of the PTE bits <15:0> of a shared valid PTE (PTE<0>=1) such that an access that was allowed before the change is not allowed after the change.

1. The PTE<0> is cleared without changing any of the PTE bits <63:32> and <15:1>.
2. All processors do a TBIS for the VA mapped by the PTE that changed. The VA used in the TBIS must assume that the PTE granularity hint bits are zero.
3. After all processors have done the TBIS, the new PTE may be written changing any or all fields.

NOTE: This procedure allows the QUEUE instructions that have probed to check that all can complete, to service a TB miss. The QUEUE instruction will use the PTE even though the V bit is clear, if during its initial probe flow the V bit was set.

11.1.6 Memory Protection

Memory protection is the function of validating whether a particular type of access is allowed to a specific page from a particular access mode. Access to each page is controlled by a protection code that specifies, for each access mode, whether read or write references are allowed.

The processor uses the following to determine whether an intended access is allowed:

- The virtual address used to index page tables
- The intended access type (read data, write data, or instruction fetch)
- The current access mode from the Processor Status

If the access is allowed and the address can be mapped (the PTE is valid), the result is the physical address corresponding to the specified virtual address.

For protection checks, the intended access is read for data loads and instruction fetch, and write for data stores.

If an operand is an address operand, then no reference is made to memory. Hence, the page need not be accessible nor mapped to a physical page.

11.1.6.1 Processor Access Modes

There are four processor modes:

- Kernel
- Executive
- Supervisor
- User

The access mode of a running process is stored in the Current Mode bits of the Processor Status (PS). Refer to the *AARM* for details.

11.1.6.2 Protection Code

Every page in the virtual address space is protected according to its use. A program may be prevented from reading or writing portions of its address space. Associated with each page is a protection code that describes the accessibility of the page for each processor mode. The code allows a choice of read or write protection for each processor mode.

- Each mode's access can be read/write, read-only, or no-access.
- Read and write accessibility are specified independently.
- The protection of each mode can be specified independently.

The protection code is specified by 8 bits in the PTE; see Table 11-2.

The OpenVMS Alpha architecture allows a page to be designated as execute only by setting the read enable bit for the access mode and by setting the fault on read and write bits in the PTE.

11.1.6.3 Access Violation Fault

An access violation fault occurs if an illegal access is attempted, as determined by the current processor mode and the page's protection field.

11.1.7 Address Translation

The page tables can be accessed from physical memory or (to reduce overhead) through a mapping to a linear region of the virtual address space. All implementations must support the virtual access method and are expected to use it as the primary access method to enhance performance.

11.1.7.1 Physical Access for Page Table Entries

Physical address translation is performed by accessing entries in a three-level page table structure. The Page Table Base Register (PTBR) contains the physical PFN of the highest level (Level 1) page table. Bits <level1> of the virtual address are used to index into the first level page table to obtain the physical PFN of the base of the second level (Level 2) page table. Bits <level2> of the virtual address are used to index into the second level page table to obtain the physical PFN of the base of the third level (Level 3) page table. Bits <level3> of the virtual address are used to index the third level page table to obtain the physical PFN of the page being referenced. The PFN is concatenated with VA <byte_within_page> to obtain the physical address of the location being accessed.

If part of any page table resides in I/O space or in nonexistent memory, the operation of the processor is UNDEFINED.

If the first-level or second-level PTE is valid, the protection bits are ignored; the protection code in the third-level PTE is used to determine accessibility.

If a first-level or second-level PTE is invalid, an access violation occurs if the PTE<KRE> equals zero. An access violation on a first-level or second-level PTE implies that all lower-level page tables mapped by that PTE do not exist.

NOTE: *This mapping scheme does not require multiple contiguous physical pages. There are no length registers. With a page size of 8 Kbytes, 3 pages (24 Kbytes) map 8 Mbytes of virtual address space; 1026 pages (approximately 8 Mbytes) map an 8-Gbyte address space; and 1,049,601 pages (approximately 8 Gbytes) map the entire 8 Tbyte (2^{43}-byte) address space.*

The algorithm to generate a physical address from a virtual address follows:

```
   IF {SEXT(VA<63:VA_SIZE>) NEQ SEXT (VA<VA_SIZE-1>} THEN
         {initiate access violation fault}

    ! Read Physical

      level1_pte ¬ ({PTBR * page_size} + {8 *
VA<level1_number>})

      IF level1_pte<V> EQ 0 THEN

         IF level1_pte<KRE> EQ 0 THEN
            {initiate access violation fault}

         ELSE
            {initiate translation not valid fault}

      ! Read Physical

      level2_pte ¬
         ({level1_pte<PFN> * page_size} + {8 *
VA<level2_number>})

      IF level2_pte<V> EQ 0 THEN
         IF level2_pte<KRE> EQ 0 THEN
            {initiate access violation fault}
         ELSE
            {initiate translation not valid fault}

      ! Read Physical

      level3_pte ¬
         ({level2_pte<PFN> * page_size} + {8 *
VA<level3_number>})

      IF {{{level3_pte<UWE> EQ 0} AND {write access} AND
{PS<CM> EQ 3}} OR
          {{level3_pte<URE> EQ 0} AND {read  access} AND
{PS<CM> EQ 3}} OR
          {{level3_pte<SWE> EQ 0} AND {write access} AND
{PS<CM> EQ 2}} OR
```

```
         {{level3_pte<SRE> EQ 0} AND {read  access} AND
{PS<CM> EQ 2}} OR
         {{level3_pte<EWE> EQ 0} AND {write access} AND
{PS<CM> EQ 1}} OR
         {{level3_pte<ERE> EQ 0} AND {read  access} AND
{PS<CM> EQ 1}} OR
         {{level3_pte<KWE> EQ 0} AND {write access} AND
{PS<CM> EQ 0}} OR
         {{level3_pte<KRE> EQ 0} AND {read  access} AND
{PS<CM> EQ 0}}}
     THEN
        {initiate access violation fault}
     ELSE
        IF level3_pte<V> EQ 0 THEN
            {initiate translation not valid fault}

     IF {level3_pte<FOW> EQ 1} AND { write   access} THEN
        {initiate Fault On Write fault}
     IF {level3_pte<FOR> EQ 1} AND { read    access} THEN
        {initiate Fault On Read fault}
     IF {level3_pte<FOE> EQ 1} AND { execute access} THEN
        {initiate Fault On Execute fault}

     Physical_Address ¬
        {level3_pte<PFN> * page_size} OR
          VA<byte_within_page>
```

11.1.7.2 Virtual Access for Page Table Entries

To reduce the overhead associated with the address translation in a three-level page table structure, the page tables are mapped into a linear region of the virtual address space. The virtual address of the base of the page table structure is set on a systemwide basis and is contained in the VPTB.

When a native mode DTB or ITB Miss occurs, the TBMISS flows attempt to load the level 3 page table entry using a single virtual mode load instruction.

The algorithm involving the manipulation of the missing VA is:

```
    tmp ¬ left_shift(VA, {64 - {{lg(PageSize) *4} -9 }})
    tmp ¬
         right_shift(tmp,{64 - {{lg(PageSize)*4} -9} +
lg(PageSize) -3})
    tmp ¬ VPTB OR tmp
    tmp<2:0> ¬  0
```

At this point, tmp contains the VA of the level 3 PTE. An LDQ from that VA will result in the acquisition of the PTE needed to satisfy the initial TBMISS condition.

However, in the PALcode environment, if a TBMISS occurs during an attempt to fetch the level 3 PTE, then it is necessary to use the longer sequence of three dependent loads.

The *AARM* describes the VPTB used to contain the virtual address of the base of the page table structure.

The mapping of the page tables necessary for the correct function of the algorithm is done as follows:

1. Select a $2^{3*lg(page\_size/8)+3}$ byte-aligned region (an address with 3*lg(page_size/8)+3 low order zeros) in the virtual address space. This value will be written into the VPTB.
2. Create a level 1 PTE to map the page tables as follows:

```
Level1_PTE         ← 0      ! Init all fields to 0
Level1_PTE<63:32> ← PFN of Level1 Pagetable
             ! Set PFN to PFN of level1 pagetable
Level1_PTE<8>      ← 1 ! Kernel Read Enable  (KRE)
Level1_PTE<0>      ← 1      ! Valid bit
```

3. Write the created level 1 PTE into the level 1 page table entry that corresponds to the VPTB value.
4. Set all level 1 and level 2 valid PTEs to allow kernel read access.
5. Write the VPTB with the selected base value.

NOTE: *No validity checks need be made on the value stored in the VPTB in a running system. Therefore, if the VPTB contains an invalid address, the operation is UNDEFINED.*

11.1.8 Translation Buffer

To save actual memory references when repeatedly referencing the same pages, hardware implementations include a translation buffer to remember successful virtual address translations and page states.

When the process context is changed, a new value is loaded into the Address Space Number (ASN) with a Swap Privileged Context instruction (CALL_PAL SWPCTX). This causes address translations for pages with PTE<ASM> clear to be invalidated on a processor that does not implement address space numbers. Additionally, when the software changes any part (except for the Software field) of a valid PTE, it must also move a virtual address within the corresponding page to the TBIS register (see *AARM)* with the MTPR instruction.

NOTE: *Some implementations may invalidate the entire translation buffer on an MTPR to TBIS. In general, implementations may invalidate more than the required translations in the TB.*

The entire translation buffer can be invalidated by doing a write to the TBIA (CALL_PAL MTPR_TBIA, and all ASM=0 entries can be invalidated by doing a write to TBIAP (CALL_PAL MTPR_TBIAP). See *AARM*.

The translation buffer must not store invalid PTEs. Therefore, software is not required to invalidate translation buffer entries when making changes for PTEs that are already invalid.

The TBCHK (see *AARM*) is available for interrogating the presence of a valid translation in the translation buffer.

11.1.9 Address Space Numbers

The Alpha architecture allows a processor to optionally implement address space numbers (process tags) to reduce the need for invalidation of cached address translations for process-specific addresses when a context switch occurs. The supported ASN range is 0..MAX_ASN.

NOTE: If an ASN outside the range 0..MAX_ASN is assigned to a process, the operation of the processor is UNDEFINED.

The ASN for the current process is loaded by software in the ASN (see *AARM*) with a Swap Privileged Context instruction. ASNs are processor specific and the hardware makes no attempt to maintain coherency across multiple processors. In a multiprocessor system, software is responsible for ensuring the consistency of TB entries for processes that might be rescheduled on different processors.

NOTE: System software should not assume that the number of ASNs is a power of two. This allows, for example, hardware to use N TB tag bits to encode 2^N-3 ASN values, one value for ASM=1 PTEs, and one for invalid. There are several possible ways of using ASNs, and, in a multiprocessor system, there are several complications. Consider the case where a process that executed on processor–1 is rescheduled on processor–2. If a page is deleted or its protection is changed, the TB in processor–1 has stale data. One solution would be to send an interprocessor interrupt to all the processors on which this process could have run and cause them to invalidate the changed PTE. This results in significant overhead in a system with several processors. Another solution would be to have software invalidate all TB entries for a process on a new processor before it can begin execution if the process executed on another processor during its previous execution. This ensures the deletion of possibly stale TB entries on the new processor. A third solution would assign a new ASN whenever a process is run on a processor that is not the same as the last processor on which it ran.

11.1.10 Memory Management Faults

Five types of faults are associated with memory access and protection:

- **Access Violation (ACV)**
  Taken when the protection field of the third-level PTE that maps the data indicates that the intended page reference would be illegal in the specified access mode. An ACV fault is also taken if <KRE> is zero in an invalid first or second level PTE.
- **Fault On Read (FOR)**
  Occurs when a read is attempted with PTE<FOR> set.
- **Fault On Write (FOW)**
  Occurs when a write is attempted with PTE<FOW> set.
- **Fault On Execute (FOE)**
  Occurs when instruction execution is attempted with PTE<FOE> set.
- **Translation Not Valid (TNV)**
  Taken when a read or write reference is attempted through an invalid PTE in a first-, second-, or third-level page table.

Refer to the *AARM* for a detailed description of these faults.

Note that these five faults have distinct vectors in the system control block.

The ACV fault takes precedence over the TNV, FOR, FOW, and FOE faults.

The TNV fault takes precedence over the FOR, FOW, and FOE faults.

The faults FOR and FOW can occur simultaneously in the CALL_PAL queue instructions, in which case the order that the exceptions are taken is UNPREDICTABLE.

11.2 OpenVMS AXP Process Structure

A process is the basic entity that is scheduled for execution by the processor. A process represents a single thread of execution and consists of an address space and both hardware and software context.

The hardware context of a process is defined by:

- 31 integer registers and 31 floating-point registers
- Processor Status (PS)
- Program Counter (PC)
- 4 stack pointers
- AST Enable and AST Summary Registers (ASTEN and ASTSR)
- Page Table Base Register (PTBR)
- Address Space Number (ASN)
- Floating-Point Enable Register (FEN)
- Process Cycle Counter (PCC)
- Process Unique Value
- Data Alignment Trap (DAT)
- Performance Monitor Enable Register (PME)

NOTE: Consult the AARM for detailed discussions of the parameters appearing in the hardware context of a process.

The software context of a process is defined by operating system software and is system dependent.

A process may share the same address space with other processes or have an address space of its own. There is, however, no separate address space for system software, and therefore, the operating system must be mapped into the address space of each process.

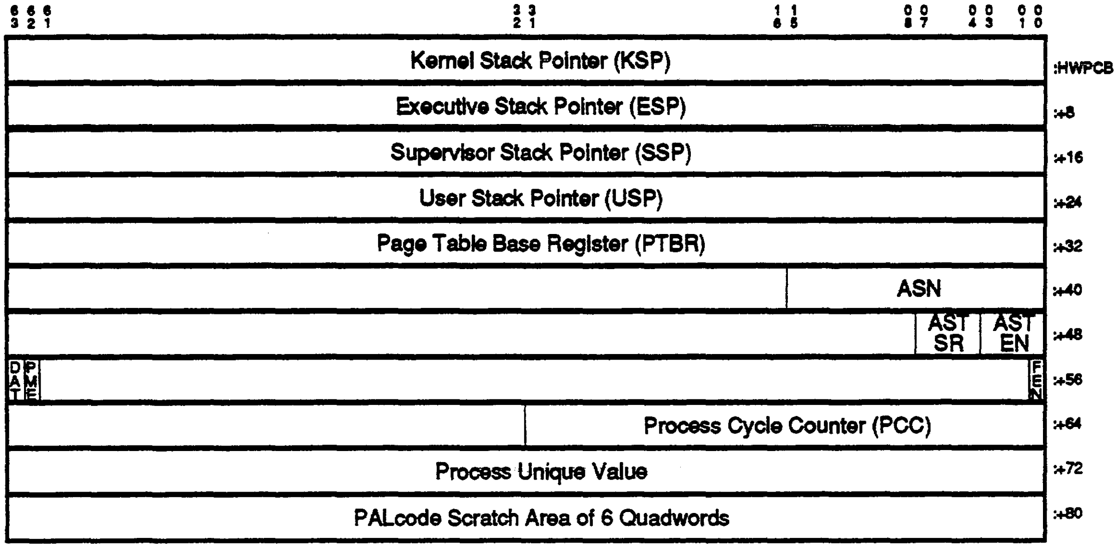
Saving the hardware context of the current process in memory followed by loading the hardware context for a new process is termed context switching. Context switching occurs as one process after another is scheduled by the operating system for execution.

11.2.1 Hardware Privileged Process Context

The hardware context of a process is defined by a privileged part which is context switched with the Swap Privileged Context instruction (SWPCTX), and a nonprivileged part, which is context switched by operating system software.

When a process is not executing, its privileged context is stored in a 128 byte naturally aligned memory structure called the Hardware Privileged Context Block (see Figure 11-3).

Figure 11-3 Hardware Privileged Context Block


63 62 61
32 31
16 15
08 07
04 03
01 00
Kernel Stack Pointer (KSP)
:HWPCB
Executive Stack Pointer (ESP)
:+8
Supervisor Stack Pointer (SSP)
:+16
User Stack Pointer (USP)
:+24
Page Table Base Register (PTBR)
:+32
ASN
:+40
AST SR
AST EN
:+48
DATE
PME
FEN
:+56
Process Cycle Counter (PCC)
:+64
Process Unique Value
:+72
PALcode Scratch Area of 6 Quadwords
:+80
BXB-0630-93

The Hardware Privileged Context Block (HWPCB) for the current process is specified by the PCBB.

The Swap Privileged Context instruction (SWPCTX) saves the privileged context of the current process into the HWPCB specified by the PCBB, loads a new value into the PCBB, and then loads the privileged context of the new process into the appropriate hardware registers.

The new value loaded into the PCBB, as well as the contents of the Privileged Context Block, must satisfy certain constraints or an UNDEFINED operation results:

- The physical address loaded into the PCBB must be 128-byte aligned and describes 16 contiguous quadwords that are in a memory-like region.
- The value of the PTBR must be the PFN of an existent page that is in a memory-like region.

It is the responsibility of the operating system to save and load the nonprivileged part of the hardware context.

The SWPCTX instruction returns ownership of the current HWPCB to operating system software and passes ownership of the new HWPCB from the operating system to the processor. Any attempt to write a HWPCB while ownership resides with the processor has UNDEFINED results. If the HWPCB is read while ownership resides with the processor, it is UNPREDICTABLE whether the original or an updated value of a field is read. The processor is free to update an HWPCB field at any time. The decision as to whether or not a field is updated is made individually for each field.

If ASNs are not implemented, the ASN field is not read or written by PALcode.

The FEN bit reflects the setting of the FEN.

The DAT bit controls whether data alignment traps that are fixed up in PALcode are reported to the operating system. If the bit is clear, the trap is reported. If the bit is set, after the fixup, return is to the user.

Setting the PME bit alerts any performance hardware or software in the system to monitor the performance of this process.

The Process Unique value is that value used in support of multithread implementations. The value is stored in the HWPCB when the process is not active. When the process is active, the value may be cached in hardware internal storage or kept in the HWPCB only.

11.2.2 Asynchronous System Traps (AST)

Asynchronous System Traps (ASTs) are a means of notifying a process of events that are not synchronized with its execution but must be dealt with in the context of the process with minimum delay.

ASTs interrupt process execution and are controlled by the ASTEN and ASTSR registers.

The ASTEN contains an enable bit for each of the four processor access modes. When the bit corresponding to an access mode is set, ASTs for that mode are enabled. The AST enable bit for an access mode may be changed by executing a Swap AST Enable instruction (SWASTEN) or by executing an MTPR instruction specifying ASTEN (MTPR ASTEN).

The ASTSR contains a pending bit for each of the four processor access modes. When the bit corresponding to an access mode is set, an AST is pending for that mode.

Kernel mode software may request an AST for a particular access mode by executing an MTPR instruction specifying ASTSR (MTPR ASTSR).

Hardware or PALcode monitors the state of ASTEN, ASTSR, PS<CM>, and PS<IPL>. If PS<IPL> is less than 2, and there is an AST pending and enabled for an access mode that is less than or equal to PS<CM> (that is, an equal or more privileged access mode), an AST is initiated at IPL 2.

ASTs that are pending and enabled for a less privileged access mode are not allowed to interrupt execution in a more privileged access mode.

11.2.3 Process Context Switching

Process context switching occurs as one process after another is scheduled for execution by operating system software. Context switching requires the hardware context of one process to be saved in memory followed by the loading of the hardware context for another process into the hardware registers.

The privileged hardware context is swapped with the CALL_PAL Swap Privileged Context instruction (SWPCTX). Other hardware context must be saved and restored by operating system software.

The sequence in which process context is changed is important since the SWPCTX instruction changes the environment in which the context switching software itself is executing. Also, although not enforced by hardware, it is advisable to execute the actual context switching software in an environment that cannot be context switched (that is, at an IPL high enough that rescheduling cannot occur).

The SWPCTX instruction is the only method provided for loading certain internal processor registers. The SWPCTX instruction always saves the privileged context of the old process and loads the privileged context of a new process. Therefore, a valid HWPCB must be available to save the privileged context of the old process as well as load the privileged context of the new process.

Chapter 12

DEC OSF/1 AXP System Support

This chapter discusses memory management performed by the DEC OSF/1 AXP operating system and gives the structure of a process within the DEC OSF/1 AXP environment. Consult the *Alpha Architecture Reference Manual* (hereafter referred to as *AARM* in this chapter) for a thorough discussion of these topics.

12.1 DEC OSF/1 AXP Memory Management

Memory management is the control of the allocation and use of physical memory. It is implemented by a combination of hardware and software. Typically, in a multiprogramming system, several processes may reside in physical memory at the same time. DEC OSF/1 AXP uses memory protection and multiple address spaces to ensure that one process will not affect either other processes or the operating system.

To improve further software reliability, the DEC OSF/1 AXP operating system provides two hierarchical access modes: kernel and user. Protection is specified at the individual page level, where a page may be inaccessible, read only, or read/write for the user mode. Accessible pages can be restricted to have only data or instruction access.

A program uses virtual addresses to access its data and instructions. However, before these virtual addresses can be used to access memory, they must be translated into physical addresses. Memory management software maintains tables of mapping information (page tables) that keep track of where each virtual page is located in physical memory. The processor uses this mapping information when it translates virtual addresses to physical addresses.

Therefore, memory management provides both memory protection and memory mapping mechanisms. The DEC OSF/1 AXP memory management architecture is designed to meet several goals:

- Provide a large address space for instructions and data.
- Allow programs to run on hardware with physical memory smaller than the virtual memory used.
- Provide convenient and efficient sharing of instructions and data.
- Allow sparse use of a large address space without excessive page table overhead.
- Contribute to software reliability.
- Provide independent read and write access protection.

12.1.1 Virtual Address Spaces

A virtual address is a 64-bit unsigned integer that specifies a byte location within the virtual address space. Implementations support the address space in one of four sizes (43, 47, 51, or 55 bits) as a function of page size. The minimal supported virtual address size is 43 bits. If an implementation supports less than 64-bit virtual addresses, it must check that all the VA<63:vaSize> bits are equal to VA<vaSize−1>). This gives two disjoint ranges for valid virtual addresses. For example, for a 43-bit virtual address space, valid virtual address ranges are 0 to 3FF FFFF FFFF and FFFF FC00 0000 0000 to FFFF FFFF FFFF FFFF. Access to virtual addresses outside of an implementation's valid virtual address range cause an access violation fault.

The virtual address space is divided into three segments. The two bits VA <vaSize−1:vaSize−2> select a segment as shown in Table 12-1.

Table 12-1 Virtual Address Space Segments

VA<vaSize−1:vaSize−2>	Name	Mapping	Access Control
0x	seg0	Mapped via TB	Programmed in PTE
10	kseg	PA<--sext(VA<vaSize−3:0>)	Kernel read/write
11	seg1	Mapped via TB	Programmed in PTE

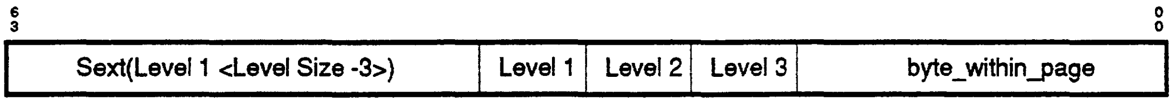
For kseg, the relocation, sharing, and protection are fixed. For seg0 and seg1, the virtual address space is broken into pages, which are the units of relocation, sharing, and protection. The page size ranges from 8 Kbytes to 64 Kbytes. Therefore, system software should allocate regions with differing protection on 64-Kbyte virtual address boundaries to ensure image compatibility across all Alpha implementations.

Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. The operating system controls the virtual-to-physical address mapping tables and saves the inactive (but used) parts of the virtual address space on external storage media.

12.1.1.1 Segment Seg0 and Seg1 Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. A seg0 or seg1 virtual address consists of three level-number fields and a byte_within_page field, as shown in Figure 12-1.

Figure 12-1 Virtual Address Format



BXB-0628-93

The byte_within_page field can be either 13, 14, 15, or 16 bits depending on a particular implementation. Thus, the allowable page sizes are 8, 16, 32, and 64 Kbytes. Each level-number field is 0-*n* bits long, where, for example, *n* is 9 for an 8K page size. Level-number fields are the same size for a given implementation.

The level-number fields are a function of the page size; all page table entries at any given level do not exceed one page. The PFN field in the PTE is always 32 bits wide. Thus, as the page size grows the virtual and physical address size also grows.

In Table 12-2 the physical address column is the maximum physical address supported by the smaller of seg0/seg1 or kseg, as indicated.

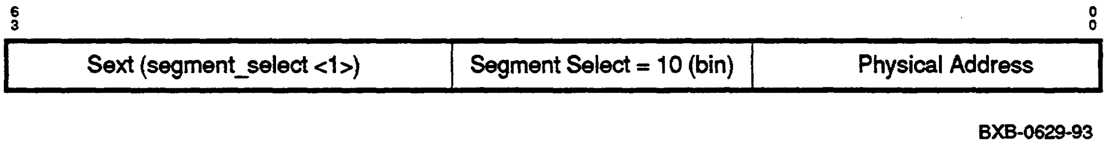
Table 12-2 Virtual Address Options

Page Size (Kbytes)	Byte Offset (Bits)	Level Size (Bits)	Virtual Address (Bits)	Physical Address (Bits)	Physical Address Limited by
8	13	10	43	41	kseg
16	14	11	47	45	kseg
32	15	12	51	47	seg0/seg1
64	16	13	55	48	seg0/seg1

12.1.1.2 Kseg Virtual Address Format

The processor generates a 64-bit virtual address for each instruction and operand in memory. A kseg virtual address consists of segment select field with a value of 10 (bin) and a physical address field. The segment select field is the two bits VA<vaSize–1:vaSize–2>. The physical address field is VA<vaSize–3:0>. The kseg virtual address format is shown in Figure 12-2.

Figure 12-2 Kseg Virtual Address Format



12.1.2 Physical Address Space

Physical addresses are at most vaSize–2 bits. This allows all of physical memory to be accessed via kseg. A processor may choose to implement a smaller physical address space by not implementing some number of high order bits. The two most significant implemented physical address bits select a caching policy or implementation-dependent type of address space. Implementations will use these bits as appropriate for their systems. For example, in a workstation with a 30-bit physical address space, bit <29> might select between memory and non-memory like regions, and bit <28> could enable or disable caching.

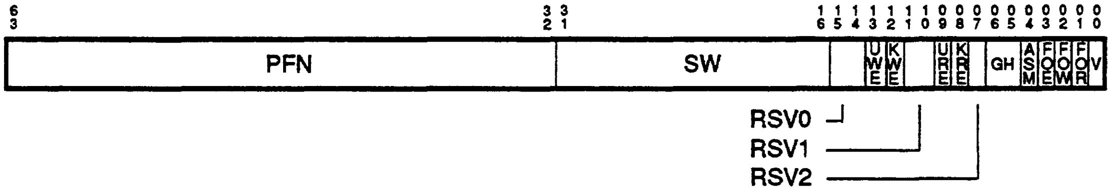
12.1.3 Memory Management Control

Memory management is always enabled. Implementations must provide an environment for PALcode to service exceptions and to initialize and boot the processor. For example, PALcode might run with I-stream mapping disabled.

12.1.4 Page Table Entries

The processor uses a quadword page table entry (PTE) to translate seg0 and seg1 virtual addresses to physical addresses. A PTE contains hardware and software control information and the physical page frame number (PFN). A PTE is a quadword with fields as shown in Figure 12-3. Table 12-3 gives the definitions of the PTE fields.

Figure 12-3 Page Table Entry (PTE)



BXB-0632A-93

Table 12-3 Page Table Entry Bit Definitions

Name	Bit(s)	Function
PFN	<63:32>	**Page Frame Number.** The PFN field always points to a page boundary. If V is set, the PFN is concatenated with the VA<byte_within_page> to obtain the physical address.
SW	<31:16>	**Software.** Reserved for software.
RSV0	<15:14>	**Reserved 0.** Reserved for hardware; SBZ.
UWE	<13>	**User Write Enable.** Enables writes from user mode. If this bit is a 0 and a STORE is attempted while in user mode, an access violation occurs. This bit is valid even when <V>=0.
KWE	<12>	**Kernel Write Enable.** Enables writes from kernel mode. If this bit is a 0 and a STORE is attempted while in kernel mode, an access violation occurs. This bit is valid even when <V>=0.
RSV1	<11:10>	**Reserved 1.** Reserved for hardware; SBZ.
URE	<9>	**User Read Enable.** Enables reads from user mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in user mode, an access violation occurs. This bit is valid even when <V>=0.
KRE	<8>	**Kernel Read Enable.** Enables reads from kernel mode. If this bit is a 0 and a LOAD or instruction fetch is attempted while in kernel mode, an access violation occurs. This bit is valid even when <V>=0.
RSV2	<7>	**Reserved 2.** Reserved for hardware; SBZ.

Table 12-3 Page Table Entry Bit Definitions (Continued)

Name	Bit(s)	Function
GH	<6:5>	**Granularity Hint.** Software may set these bits to a nonzero value to supply a hint to translation buffer implementations that a block of pages can be treated as a single larger page: 1. The block is an aligned group of 8^N pages, where N is the value of PTE<6:5>, e.g., a group of 1, 8, 64, or 512 pages starting at a virtual address with page_size + 3*N low-order zeros. 2. The block is a group of physically contiguous pages that are aligned both virtually and physically. Within the block, the low 3*N bits of the PFNs describe the identity mapping and the high 32–3*N PFN bits are all equal. 3. Within the block, all PTEs have the same values for bits <15:0>, i.e., protection, fault, granularity, and valid bits. Hardware may use this hint to map the entire block with a single TB entry, instead of 8, 64, or 512 separate TB entries.
ASM	<4>	**Address Space Match.** When set, this PTE matches all ASNs. For a given VA, ASM must be set consistently in all processes, otherwise the address mapping is UNPREDICTABLE.
FOE	<3>	**Fault On Execute.** When set, a fault on execute exception occurs on an attempt to execute an instruction in the page.
FOW	<2>	**Fault On Write.** When set, a fault on write exception occurs on an attempt to write any location in the page.
FOR	<1>	**Fault On Read.** When set, a fault on read exception occurs on an attempt to read any location in the page.
V	<0>	**Valid.** Indicates the validity of the the PFN field. When <V> is set, the PFN field is valid for use by hardware. When V is clear, the PFN field is reserved for use by software. <V> does not affect the validity of PTE<15:1>.

The operating system changes PTEs as part of its memory management functions. For example, the operating system may set or clear the V bit, change the PFN field as pages are moved to and from external storage media, or modify the software bits. The processor hardware never changes PTEs.

Software must guarantee that each PTE is always consistent within itself. Changing a PTE one field at a time can cause incorrect system operation, such as setting PTE<V> with one instruction before establishing PTE<PFN> with another. Execution of an interrupt service routine between the two instructions could use an address that would map using the inconsistent PTE. Software can solve this problem by building a complete new PTE in a register and then moving the new PTE to the page table by using an STQ instruction.

Multiprocessing makes the problem more complicated. Another processor could be reading (or even changing) the same PTE that the first processor is changing. Such concurrent access must produce consistent results. Software must use some form of software synchronization to modify PTEs that are already valid. Whenever a processor modifies a valid PTE, it is possible that other processors in a multiprocessor system may have old copies of that PTE in their translation buffer. Software must inform other processors of changes to PTEs. Hardware must ensure that aligned quadword reads and writes are atomic operations. Hardware must not cache invalid PTEs (PTEs with <V>=0) in translation buffers.

12.1.5 Memory Protection

Memory protection is the function of validating whether a particular type of access is allowed to a specific page from a particular access mode. Access to each page is controlled by a protection code that specifies, for each access mode, whether read or write references are allowed. The processor uses the following to determine whether an intended access is allowed:

- The virtual address, which is used either to select kseg mapping or provide the index into the page tables.
- The intended access type (read or write).
- The current access mode base on processor mode.

For protection checks, the intended access is read for data loads and instruction fetches, and write for data stores.

12.1.5.1 Processor Access Modes

There are two processor modes, user and kernel. The access mode of a running process is stored in PS<MODE>.

12.1.5.2 Protection Code

Every page in the virtual address space is protected according to its use. A program may be prevented from reading or writing portions of its address space. Associated with each page is a protection code that describes the accessibility of the page for each processor mode.

For seg0 and seg1, the code allows a choice of read or write protection for each processor mode. For each mode, access can be read/write, read only, or no access. Read and write accessibility and the protection for each mode are specified independently.

For kseg, the protection code is kernel read/write, user no access.

12.1.5.3 Access Violation Fault

An access violation memory management fault occurs if an illegal access is attempted, as determined by the current processor mode and the page's protection.

12.1.6 Address Translation for Seg0 and Seg1

The page tables can be accessed from physical memory, or (to reduce overhead) can be mapped to a linear region of the virtual address space. The following sections describe both access methods.

12.1.6.1 Physical Access for Seg0 and Seg1 PTEs

Seg0 and seg1 address translation can be performed by accessing entries in a three-level page table structure. The PTBR (see *AARM)* contains the physical PFN of the highest level (Level 1) page table. Bits <level1> of the virtual address are used to index into the first level page table to obtain the physical PFN of the base of the second level (Level 2) page table. Bits <level2> of the virtual address are used to index into the second level page table to obtain the physical PFN of the base of the third level (Level 3) page table. Bits <level3> of the virtual address are used to index the third level page table to obtain the physical PFN of the page being referenced. The PFN is concatenated with VA<byte_within_page> to obtain the physical address of the location being accessed.

If part of any page table does not reside in a memory-like region, or does reside in nonexistent memory, the operation of the processor is UNDEFINED.

If the first level or second level PTE is valid, the protection bits are ignored; the protection code in the third level PTE is used to determine accessibility. If a first level or second level PTE is invalid, an access-violation fault occurs if PTE<KRE>=0. An access violation fault on a first level or second level PTE implies that all lower level page tables mapped by that PTE do not exist.

The algorithm to generate a physical address from a seg0 or seg1 virtual address follows:

```
IF {SEXT(VA<vaSize-1:0>) neq VA} THEN
        { initiate access-violation fault}

level1_PTE ¬ ({PTBR * page_size} + {8 * VA<level1>}
! Read physical
IF level1_PTE<v> EQ 0 THEN
        IF level1_PTE<KRE> eq 0 THEN
                { initiate access-violation fault}
        ELSE
                { initiate translation-not-valid fault}

level2_PTE ¬ ({level1_PTE<PFN> * page_size} + {8 *
VA<level2>} )    ! Read physical

IF level2_PTE<v> EQ 0 THEN
        IF level2_PTE<KRE> eq 0 THEN
                { initiate access-violation fault}

        ELSE
                { initiate translation-not-valid fault}

level3_PTE ¬ ({level2_PTE<PFN> * page_size} + {8 *
VA<level3>} )    ! Read physical

IF {{{level3_PTE<UWE> eq 0} AND {write access} AND
{ps<mode> EQ 1} } OR
    {{level3_PTE<URE> eq 0} AND {read access} AND
{ps<mode> EQ 1} } OR
    {{level3_PTE<KWE> eq 0} AND {write access} AND
{ps<mode> EQ 0} } OR
    {{level3_PTE<KRE> eq 0} AND {read access} AND
{ps<mode> EQ 0} } }
        THEN
```

```
                {initiate memory-management fault}
        ELSE
                IF level3_PTE<v> EQ 0 THEN
                        {initiate memory-management fault}
IF { level3_PTE<FOW> eq 1} AND {write access} THEN
        {initiate memory-management fault}
IF { level3_PTE<FOR> eq 1} AND {read access} THEN
        {initiate memory-management fault}
IF { level3_PTE<FOE> eq 1} AND {execute access} THEN
        {initiate memory-management fault}
Physical_address ¬ {level3_PTE<PFN> * page_size} OR
VA<byte_within_page>
```

12.1.6.2 Virtual Access for Seg0 or Seg1 PTEs

The page tables can be mapped into a linear region of the virtual address space, reducing the overhead for seg0 and seg1 PTE accesses. The mapping is done as follows:

1. Select a $2^{3*lg(pageSize/8)+3}$ byte-aligned region (an address with 3*lg(pageSize/8)+3 low-order zeros) in the seg0 or seg1 address space. Set the virtual page table pointer (VPTPTR) with a write virtual page table pointer instruction (wrvptptr) to the selected value.
2. Create a level1 PTE to map the page tables as follows:

```
level1_PTE = 0            ! Initialize all fields to 0
level1_PTE<63:32> = pfn_of_Level_1_pagetable
                          ! Set the PFN to the PFN of
the level one pagetable
level1_PTE<8>  = 1        ! Set the kernel read enable
bit
level1_PTE<0>  = 1        ! Set the valid bit
```

3. Set the level1 page table entry that corresponds to the VPTB to the created level1_PTE.
4. Set all level 1 and level 2 valid PTEs to allow kernel read access. With this setup in place, the algorithm to fetch a seg0 or seg1 PTE is:

```
tmp ← left_shift (va, {64 - {{lg(pageSize) *4} - 9}} )
tmp ← right_shift (tmp, {64 - {{lg(pageSize) *4} - 9}
+ lg(pageSize) - 3} )
tmp ← VPTB OR tmp
tmp<2:0> ← 0
level3_PTE ← (tmp)       ! Load PTE using its virtual
                          ! address
```

The virtual access method is used by PALcode for most TB fills.

12.1.7 Translation Buffer

To save actual memory references when repeatedly referencing the same pages, hardware implementations include a translation buffer to remember successful virtual address translations and page states. When the process context is changed, a new value is loaded into the ASN with a swap process context (swpctx) instruction. This causes address translations for pages with PTE<ASM> clear to be invalidated on a processor that does not implement address space numbers.

Additionally, when the software changes any part (except the software field) of a valid PTE, it must also execute a CALL_PAL tbi instruction. The entire translation buffer can be invalidated by tbia, and all ASM=0 entries can be invalidated by tbiap. The translation buffer must not store invalid PTEs. Therefore, the software is not required to invalidate translation buffer entries when making changes for PTEs that are already invalid.

12.1.8 Address Space Numbers

The Alpha architecture allows a processor to optionally implement address space numbers (process tags) to reduce the need for invalidation of cached address translations for process-specific addresses when a context switch occurs.

The ASN for the current process is loaded by software in the ASN with an swpctx instruction. ASNs are processor specific and the hardware makes no attempt to maintain coherency across multiple processors. In a multiprocessor system, software is responsible for ensuring the consistency of TB entries for processes that might be rescheduled on different processors.

NOTE: System software should not assume that the number of ASNs is a power of two. This allows, for example, hardware to use N TB tag bits to encode 2^N *–3 ASN values, one value for ASM=1 PTEs, and one for invalid. There are several possible ways of using ASNs. There are several complications in a multiprocessor system. Consider the case where a process that executed on processor–1 is rescheduled on processor–2. If a page is deleted or its protection is changed, the TB in processor–1 has stale data. One solution would be to send an interprocessor interrupt to all the processors on which this process could have run and cause them to invalidate the changed PTE. This results in significant overhead in a system with several processors. Another solution would be to have software invalidate all TB entries for a process on a new processor before it can begin execution if the process executed on another processor during its previous execution. This ensures the deletion of possibly stale TB entries on the new processor. A third solution would assign a new ASN whenever a process is run on a processor that is not the same as the last processor on which it ran.*

12.1.9 Memory Management Faults

On a memory-management fault, the fault code (MMCSR) is passed in a1 to specify the type of fault encountered, as shown in Table 12-4.

Table 12-4 Memory Management Fault Type Codes

Fault	MMCSR Value
Translation not valid	0
Access violation	1
Fault on read	2
Fault on execute	3
Fault on write	4

Faults are taken as follows:

- A translation-not-valid fault is taken when a read or write reference is attempted through an invalid PTE in a first, second, or third level page table.
- An access violation fault is taken on a reference to a seg0 or seg1 address when the protection field of the third level PTE that maps the data indicates that the intended page reference would be illegal in the specified access mode. An access violation fault is also taken if <KRE> is a zero in an invalid first or second level PTE. An access violation fault is generated for any access to a kseg address when the mode is user (PS<MODE>=1).
- A fault on read (FOR) occurs when a read is attempted with PTE<FOR> set.
- A fault on execute (FOE) occurs when an instruction fetch is attempted with PTE<FOE> set.
- A fault on write (FOW) occurs when a write is attempted with PTE<FOW> set.

12.2 DEC OSF/1 AXP Process Structure

A process is a single thread of execution. It is the basic entity that can be scheduled and is executed by the processor. A process consists of an address space and both software and hardware context. The hardware context of a process is defined by the the following:

- 30 integer registers (excluding R31 and SP)
- 31 floating-point registers (excluding F31)
- Program Counter (PC)
- User stack pointer (USP) and kernel stack pointer (KSP) per process
- Processor Status (PS)
- Address Space Number (ASN)
- Process Cycle Counter (PCC)
- Page Table Base Register (PTBR)
- Process Unique value

NOTE: Consult the AARM for detailed discussions of the parameters appearing in the hardware context of a process.

While a process is executing, some of its hardware context is being updated in the internal registers. When a process is not being executed, its hardware context is stored in memory in a software structure termed the process control block (PCB). Saving the process context in the PCB and loading new values from another PCB for a new context is termed context switching. Context switching occurs as one process after another is scheduled for execution.

The PCB holds the state of a process, as shown in Figure 12-4.

Figure 12-4 Process Control Block (PCB)

63 … 32	31 … 01	00	
Kernel Stack Pointer (KSP)			:+00
User Stack Pointer (USP)			:+08
Page Table Base Register (PTBR)			:+16
Address Space Number (ASN)	Cycle Counter (PCC)		:+24
Process Unique Value			:+32
		FEN	:+40
Reserved to Digital			:+48
Reserved to Digital			:+56

BXB-0631-93

The contents of the PCB are loaded and saved by the swpctx instruction. The PCB must be quadword aligned and should be 64-byte aligned for best performance. Kernel mode code can read the PTBR, the ASN, and the FEN for the current process from the PCB. Kernel mode code must use the rdusp/wrusp instructions to access the USP. The PCC must be read with the rpcc instruction. The unique value can be accessed with the rdunique/wrunique instruction.

Chapter 13

Initialization

The KN7AA CPU module can be initialized in three ways:

- **Power-Up Sequence.** When the LSB system is powered up, the CPU module generates a local reset signal.
- **System Reset.** Whenever the LSB RESET signal is asserted, the CPU module is initialized. LSB RESET can be asserted by any node or from the control panel keyswitch through CCL_RESET.
- **Node Reset.** A single CPU module can be reset by setting LCNR<NRST>.

The processor chip (DECchip 21064) can be reset independently of the other components on the CPU module through the serial I/O port.

13.1 Initialization Overview

A CPU reset causes the console code to first invoke the on-board self-test sequence. Self-test begins by testing a very small portion of the CPU logic and gradually expands the scope of testing until all hardware functions of the module have been verified.

Action subsequent to the completion of the CPU module self-test depends on the state of LCNR<RSTSTAT>. If the state of this bit indicates a power-up or system reset, CPU module self-test is followed by CPU-based testing of other system components.

Once all appropriate testing has been completed, the KN7AA console program determines a primary processor. The primary processor is then responsible for displaying the results of all testing, configuring the LSB system (memory, registers, and so on), and creating the software data structures necessary to communicate between processors and the operating system. The console program then enters its input loop.

If the CPU module reset was caused by a node reset, no additional system components are tested. CPU registers are set to their firmware-initialized state, but no other LSB system configuration is performed. There is no change in the primary processor, and the console program enters its input loop.

13.2 Self-Test

CPU self-test is a layered process that is called as the first part of the console entry sequence. It starts with a simple load of code from the serial

ROM (SROM) into the P-cache and augments itself through additional ROM-resident code that is copied to the B-cache and runs from the B-cache. The process completes by returning a GO/NOGO status to the console entry sequence. The following subsections summarize the various stages in the CPU self-test. A complete description and flowchart of self-test sequences can be found in the *Advanced Troubleshooting* manuals.

13.2.1 SROM Operation

Following the deassertion of reset to the DECchip 21064, the contents of the SROM are loaded into the internal cache, the PC is pointed to location zero and instruction execution is started. This code performs the following:

- A quick internal test of the processor chip
- Tests the external B-cache tag, status, and data store RAMs and associated control
- Determines that access to the Gbus resources is operational
- Copies the balance of CPU self-test and the CPU console program from the Gbus ROMs to the B-cache and, following a checksum verification, transfers control to it.

At any point in this process, the SROM code can signal failures through the Gbus$LEDs register.

13.2.2 CPU Module Self-Test

Following the transfer of control from the SROM code to the main body of CPU self-test in the B-cache, the CPU module is tested thoroughly. Highlights include:

- Test of all Gbus resources including the UARTs and the watch chip
- LEVI tests including LSB transfers
- Tests of all RAM structures

13.2.3 Additional Power-Up Testing

If the CPU module self-test completes successfully, additional power-up testing is next performed to verify untested system components. Additional testing performed by all processors includes:

- Tests of the processor/memory LSB interface
- Tests of CPU multiprocessor logic
- Tests of the LSB I/O port module

The boot processor then performs tests on interfaces to the I/O port.

13.3 Console Entry

When the power-up test sequence is complete, the console code entry sequence continues. This section briefly describes the system-level initialization functions that are required to start the operating system, which are performed by the console code following self-test.

13.3.1 Boot Processor Arbitration

The console program determines the boot processor. A boot processor must also be determined on an interim basis, between phases of power-up test sequence, for the purpose of printing out test results.

The boot processor is selected dynamically. Any processor in a multiprocessor system can become the boot processor. By default, the CPU with the lowest LSB node number that has passed all of its power-up tests thus far, and is eligible, is selected as the boot processor.

If all processors fail self-test, or if all processors have been disabled through console commands from becoming boot processors, then no boot processor is assigned. In this case, a unique code is placed on the LEDs, and all processors monitor the console terminal lines waiting for a sequence to be typed by the operator, which would force one of the processors to become the boot processor.

The console **set cpu** command can be used to change the boot processor once the console is running. Refer to the *Console Reference Manual* for further information on the **set cpu** command.

13.3.2 Boot Processor System Setup

Following configuration of memory, the console creates data structures in memory that are required to communicate between processors and with the operating system. These data structures include:

- Hardware restart parameter block (HWRPB)
- A physical memory descriptor
- A bitmap of good and bad pages of physical memory
- Console routines block (CRB)
- Console terminal block (CTB)

13.3.3 Operating System Startup

The KN7AA console program's primary role in operating system startup is to load and transfer control to the primary bootstrap program. The method the console generally uses to load the primary bootstrap program is called bootblock booting. To begin the boot, the console reads the first logical block (LBN 0) on a disk. This is the bootblock, which contains information that points to the location of the primary bootstrap program on the disk. Using the same routines that read the bootblock, the console then uses this information to load the primary bootstrap program.

When booting from a network, the console must request the bootstrap image from an external server.

Once control is passed to the primary bootstrap, the console program's only remaining role is to allow access to console terminal routines and I/O routines. The console terminal routines allow the operating system software to send/receive characters to/from the console terminal. The I/O routines allow the primary bootstrap program to utilize the console boot drivers to load the secondary bootstrap or operating system software.

Chapter 14

Error Handling

Errors detected by or reported to the KN7AA processor can occur anywhere in the system. If errors occur during data movement within the DECchip 21064 or its environment, they are detected by the DECchip 21064. Errors that occur during data movement external to the DECchip 21064 (B-cache, LEVI interface, and other nodes) are detected by the LEVI interface and reported to the DECchip 21064.

The errors result in machine checks and are referred to error service routines in the PALcode. The system control block (SCB) specifies the entry points for the error service routines. The handler for machine checks executes in kernel mode, on the kernel stack, at IPL 31 (dec). Table 14-1 lists the error types and indicates their respective PALcode entry points.

Table 14-1 Error Entry Points to the PALcode Service Routines

Error Type	PALcode Entry Point (Byte Offset, Hex)
Processor Machine Check	670
System Machine Check	660
Processor Correctable Machine Check[1]	630

[1] The recovery method for this error is dependent on the DECchip 21064 revision number.

This chapter covers the following topics:

- Machine Check Overview
- DECchip 21064 Actions on Errors
- PALcode Error Handling

The chapter discusses error conditions caused by failures at the hardware level. It also presents parse trees for all machine check errors to help the programmer isolate the error to a particular fault. However, it is not the goal of this chapter to present an exhaustive discussion of error conditions. For further information on error handling, exceptions, interrupts, and machine checks, refer to the *Alpha Architecture Reference Manual*. Consult also the *DEC 7000 AXP System Advanced Troubleshooting* manual for information on software error flags.

14.1 Machine Check Overview

The machine check exception is an indication of a serious system error. Under certain conditions the error may be recoverable. Recoverability is a function of the PALcode, the saved error state, and type of error.

Machine checks occur because of one of the following classes of errors: B-cache probes, CPU fill ECC errors, or external LSB control related errors occurring synchronous to the outstanding DECchip 21064 EDAL (pin bus) request.

Notification of such errors happens in one of two ways. The probe and fill errors are internally generated machine checks done within the DECchip 21064. The external LSB control error notification is done by the LEVI interface using the cAck_h hard error response lines back to the outstanding DECchip 21064 command/request.

14.2 DECchip 21064 Actions on Errors

This section summarizes hardware flows for various error conditions handled by the DECchip 21064. When these errors occur during in-chip operations, they are detected by the DECchip 21064. If they occur in off-chip interactions, they may be recognized by the DECchip 21064. These errors may or may not be corrected by hardware.

The DECchip 21064 reports corrected hardware errors through the maskable corrected-read interrupt. ABOX_CTL<CRD_EN> controls whether error hardware generates an interrupt request for corrected errors. HIER<CRE> is used to mask pending corrected-read interrupt requests. Corrected-read interrupts are masked when the CPU is in PALmode.

The DECchip 21064 reports uncorrected hardware errors by generating a machine check trap to PALcode. ABOX_CTL <MCHK_EN> controls whether machine checks are generated by uncorrectable hardware errors. The DECchip 21064-recognized hardware errors occur during interactions between the DECchip 21064 BIU and off-chip hardware. These errors fall into three categories:

- Uncorrectable hardware errors recognized by system components while processing requests generated by the DECchip 21064, and communicated to the DECchip 21064 EDAL interface command acknowledge field (cAck_h [2:0]).
- B-cache tag probe errors recognized by the DECchip 21064 during DECchip 21064-controlled access of the B-cache.
  - — Tag address parity errors
  - — Tag control parity errors
- P-cache fill data errors recognized by the DECchip 21064. These errors could occur during the DECchip 21064 controlled reads of the B-cache or during external read transactions between the DECchip 21064 and system components.

Errors may be recognized by system level components outside the context of the DECchip 21064-generated requests. These errors and their handling depend upon the system implementation and are discussed in Section 14.3.

14.2.1 Response to Single Errors

For the purpose of illustration, this section describes the response by the DECchip 21064 to an error when its internal status registers are not already locked by some previous event.

Single-Bit I-Stream ECC Error

- Corrupted data put into I-cache; block gets validated
- Machine check if enabled by ABOX_CTL<MCHK_EN>
- BIU_STAT: FILL_ECC, FILL_IRD, and FILL_CRD set
- FILL_ADDR <33:5> & BIU_STAT<FILL_QW> give bad QW's address
- FILL_SYND contains syndrome bits associated with failing QW
- BIU_ADDR, BIU_STAT<6:0> locked; contents are UNPREDICTABLE
- BC_TAG holds results of B-cache tag probe if B-cache was enabled for this transaction

Single-Bit D-Stream ECC Error

- Corrupted data put into register file; D-cache invalidated
- Machine check if enabled by ABOX_CTL<MCHK_EN>
- BIU_STAT: FILL_ECC set; FILL_IRD clear; FILL_CRD set
- FILL_ADDR <33:5> & BIU_STAT<FILL_QW> give bad QW's address
- FILL_ADDR <4:2> contain PA bits<4:2> of location which the failing load instruction attempted to read
- FILL_SYND contains syndrome bits associated with failing quadword
- BIU_ADDR, BIU_STAT<6:0> locked; contents are UNPREDICTABLE
- BC_TAG holds results of B-cache tag probe if B-cache was enabled for this transaction

Double-Bit I-Stream ECC Error

- Corrupted data put into I-cache; block gets validated
- Machine check if enabled by ABOX_CTL<MCHK_EN>
- BIU_STAT: FILL_DPERR set; FILL_IRD set; FILL_CRD clear
- FILL_ADDR<33:5> & BIU_STAT<FILL_QW> give bad QW's address
- FILL_SYND identifies corrupted longword(s)
- BIU_ADDR, BIU_STAT<6:0> locked; contents are UNPREDICTABLE
- BC_TAG holds results of B-cache tag probe if B-cache was enabled for this transaction

Double-Bit D-Stream ECC Error

- Corrupted data put into register file; D-cache invalidated
- Machine check if enabled by ABOX_CTL<MCHK_EN>

- BIU_STAT: FILL_DPERR set; FILL_IRD clear; FILL_CRD clear
- FILL_ADDR<33:5> & BIU_STAT<FILL_QW> give bad QW's address
- FILL_ADDR<4:2> contain PA bits<4:2> of location which the failing load instruction attempted to read
- FILL_SYND identifies corrupted longword(s)
- BIU_ADDR, BIU_STAT<6:0> locked; contents are UNPREDICTABLE
- BC_TAG holds results of B-cache tag probe if B-cache was enabled for this transaction

BIU: Tag Address Parity Error

- Recognized at end of tag probe sequence
- Lookup uses predicted parity so transaction misses the B-cache
- BC_TAG holds results of B-cache tag probe
- Machine check if enabled by ABOX_CTL<MCHK_EN>
- BIU_STAT<BC_TPERR> set
- BIU_ADDR holds address

BIU: Tag Control Parity Error

- Recognized at end of tag probe sequence
- Transaction forced to miss B-cache
- BC_TAG holds results of B-cache tag probe
- Machine check if enabled by ABOX_CTL<MCHK_EN>
- BIU_STAT<BC_TCPERR> set
- BIU_ADDR holds address

BIU: System External Transaction Terminated with CACK_SERR

- CRD interrupt posted if enabled by ABOX_CTL<CRD_EN>
- BIU_STAT: BIU_SERR set; BIU_CMD holds cReq_h <2:0>
- BIU_ADDR holds address

BIU: System Transaction Terminated with CACK_HERR

- Machine check if enabled by ABOX_CTL<MCHK_EN>
- BIU_STAT<BIU_HERR> set; BIU_CMD holds cReq_h <2:0>
- BIU_ADDR holds address

BIU: I-Stream Parity Error (parity mode only)

- Data put into I-cache unchanged; block gets validated
- Machine check if enabled by ABOX_CTL<MCHK_EN>
- BIU_STAT: FILL_DPERR; set, FILL_IRD set; FILL_CRD clear

- FILL_ADDR<33:5> & BIU_STAT<FILL_QW> give bad QW's address
- FILL_SYND identifies failing longword(s)
- BIU_ADDR, BIU_STAT<6:0> locked; contents are UNPREDICTABLE
- BC_TAG holds results of B-cache tag probe if B-cache was enabled for this transaction

BIU: D-Stream Parity Error (parity mode only)

- Data put into D-cache unchanged, block gets validated
- Machine check if enabled by ABOX_CTL<MCHK_EN>
- BIU_STAT: FILL_DPERR set; FILL_IRD clear
- FILL_ADDR <33:5> & BIU_STAT<FILL_QW> give bad QW's address
- FILL_ADDR<4:2> contain PA bits<4:2> of location which the failing load instruction attempted to read
- FILL_SYND identifies failing longword(s)
- BIU_ADDR, BIU_STAT<6:0> locked; contents are UNPREDICTABLE
- BC_TAG holds results of B-cache tag probe if B-cache was enabled for this transaction

14.2.2 Response to Multiple Errors

This section describes the DECchip 21064 response to multiple hardware errors, that is, to errors that occur after an initial error and before execution of the PALcode exception handler associated with that initial error.

The DECchip 21064 error reporting hardware consists of two sets of independent error reporting registers.

- BIU_STAT<7:0> and BIU_ADDR contain information about the following hardware errors:
  - — Correctable or uncorrectable errors reported with cAck_h <2:0> by system components
  - — Tag probe parity errors in the tag address or tag control fields
- BIU_STAT<14:8>, FILL_ADDR, and FILL_SYND contain error information about data fill errors.

The BC_TAG register contains information that can relate to any of the error conditions listed above.

Both sets of error registers can contain information about either corrected or uncorrected hardware errors. When a hardware error occurs, information about that error is loaded into the appropriate set of error registers, and those registers are locked against further updates until PALcode explicitly unlocks them. If a second error occurs between the time that an initial error occurs and the time that software unlocks the associated error reporting registers, information about the second is lost.

When the DECchip 21064 recognizes the second error, it still posts the required corrected-read interrupt or machine check; however it does not overwrite information previously locked in an error reporting register. If the second hardware error is not correctable and the error reporting register normally associated with this second error is already locked, the DECchip 21064 will set a bit to indicate that information about an uncorrectable hardware error was lost. Both sets of error registers have a bit to report these fatal errors.

For example, BIU_STAT<FATAL1> is set by hardware to indicate that a tag probe parity error or HARD_ERROR-terminated external transaction occurred while BIU_STAT<6:0>, BIU_ADDR, and BC_TAG were already locked due to some previous error. If a SOFT_ERROR-terminated transaction occurs while these registers are locked, FATAL1 is not set, however. Similarly, BIU_STAT<FATAL2> is set by hardware to indicate that a primary cache fill received either a parity or single- or double-bit ECC error while BIU_STAT <13:8>, FILL_ADDR, FILL_SYND, and BC_TAG were already locked.

14.3 PALcode Error Handling

A PALcode error handling routine is invoked when a machine check is taken by the processor or other system components. This section discusses what these error routines are and offers guidance to the operating system programmer in trying to diagnose the fault. It covers the following topics:

- Error log packets
- Processor machine check 670 errors
- System machine check 660 errors
- Processor correctable machine check 630 errors

Parse trees accompanying the discussions help the programmer isolate errors to particular system faults.

14.3.1 Error Log Packets

Error information is entered by PALcode in the form frames. This section provides the error log formats for various PALcode entry points and stack frames for OpenVMS AXP and DEC OSF/1 AXP machine checks.

When a machine check/interrupt occurs, the PALcode gathers information to be included in the stack frame. Upon entry to the service routine, R4 points to the frame.

Subpackets may be appended to error log packets to provide additional information to help isolate a particular fault. The following subpackets are associated with error types discussed in this chapter:

- **Processor Machine Check 670**

  DLIST—Disabled Resource List
  LSB—LSB Bus Snapshot
  LMA—LSB Memory
  Log Adapter—LSB Adapter

- **System Machine Check 660**

  DLIST—Disabled Resource List
  LSB—LSB Bus Snapshot
  LMA—LSB Memory
  Log Adapter—LSB Adapter

- **Processor Correctable Machine Check 630**

  DLIST—Disabled Resource List
  LSB—LSB Bus Snapshot
  LMA—LSB Memory

Figure 14-1 shows the format of the 670/660 machine check error log packet.

Figure 14-1 670/660 Machine Check Error Log Packet Format

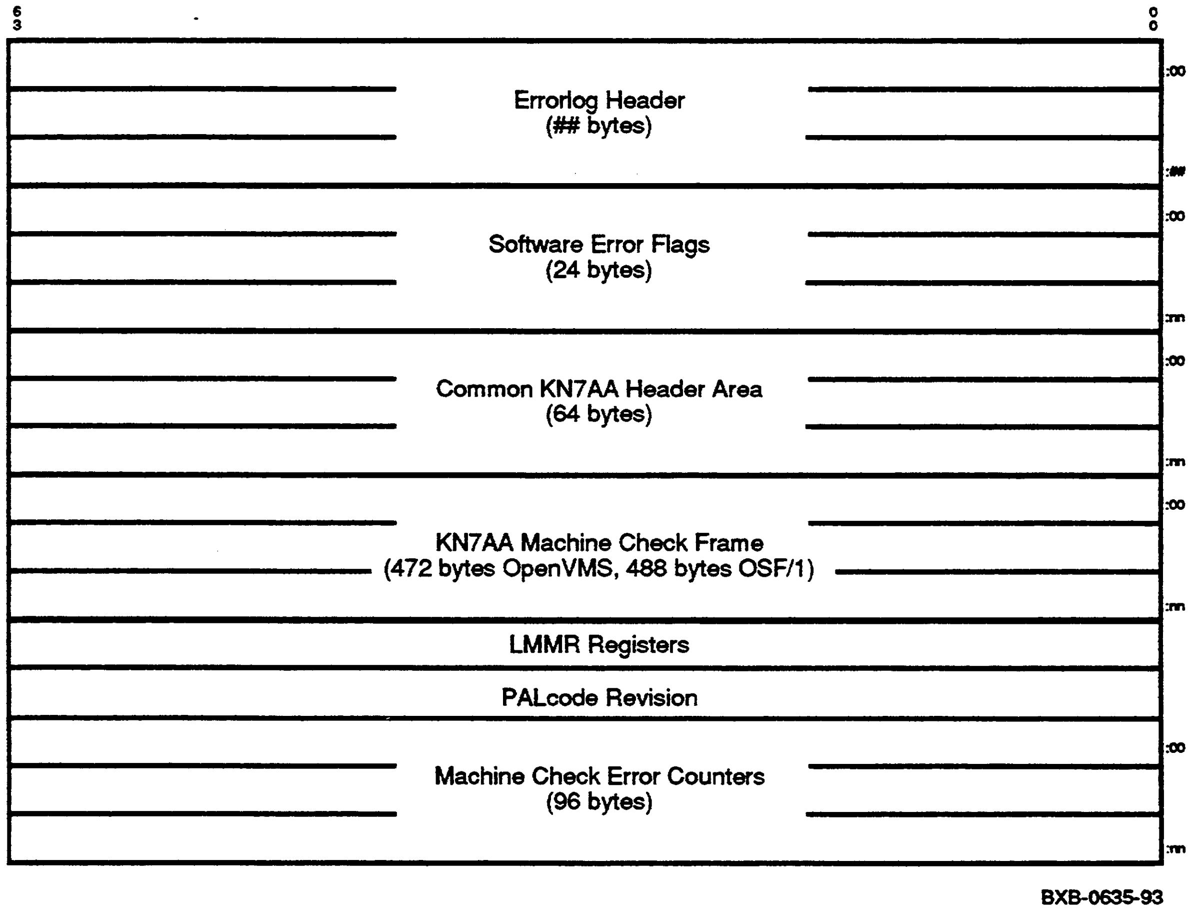


Figure 14-2 shows the stack frame of the 670/660 machine check.

Figure 14-2 670/660 Stack Frame

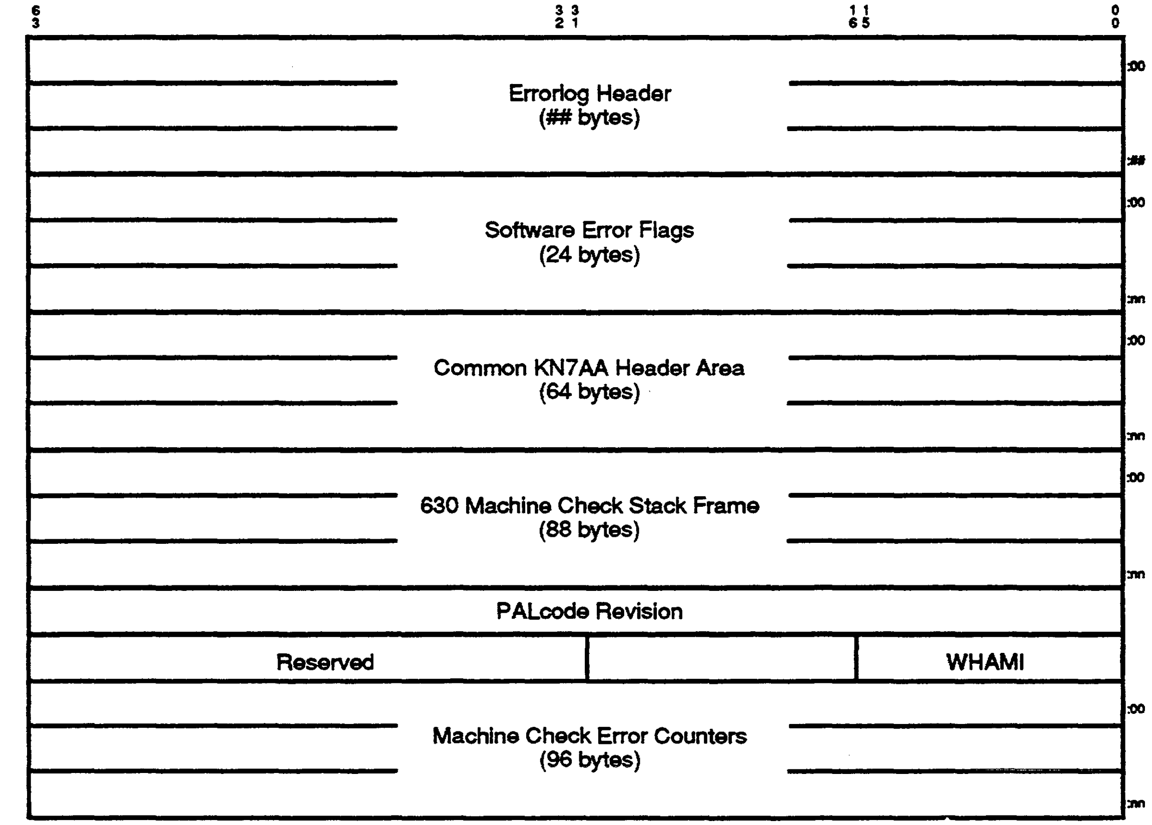
63 62 32 31 16 15 0

			Offset
R		Byte Count	: 0
Sys$$offset = [1A0]	Proc$$offset = [110]		:+ 8
Machine Check Frame Revision	Reason Mask		:+ 10
PAL Temps <1:31>			:+ 18
EXC_ADDR			:+110
EXC_SUM			:+118
EXC_MASK			:+120
ICCSR			:+128
PAL_BASE			:+130
HIER			:+138
HIRR			:+140
MM_CSR			:+148
DC_STAT			:+150
DC_ADDR			:+158
ABOX_CTL			:+160
BIU_STAT			:+168
BIU_ADDR			:+170
BIU_CTL			:+178
FILL_SYNDROME			:+180
FILL_ADDR			:+188
VA			:+190
BC_TAG			:+198
GBUS$: WHAMI <55:48>, PMASK <39:32>, INTR <23:16>, HALD <7:0>			:+1A0
LBER	LDEV		:+1A8
LMERR	LCNR		:+1B0
LBESR1	LBESR0		:+1B8
LBESR3	LBESR2		:+1C0
LBECR1	LBECR0		:+1C8
LLOCK	LMODE		:+1D0

BXB-0639A-93

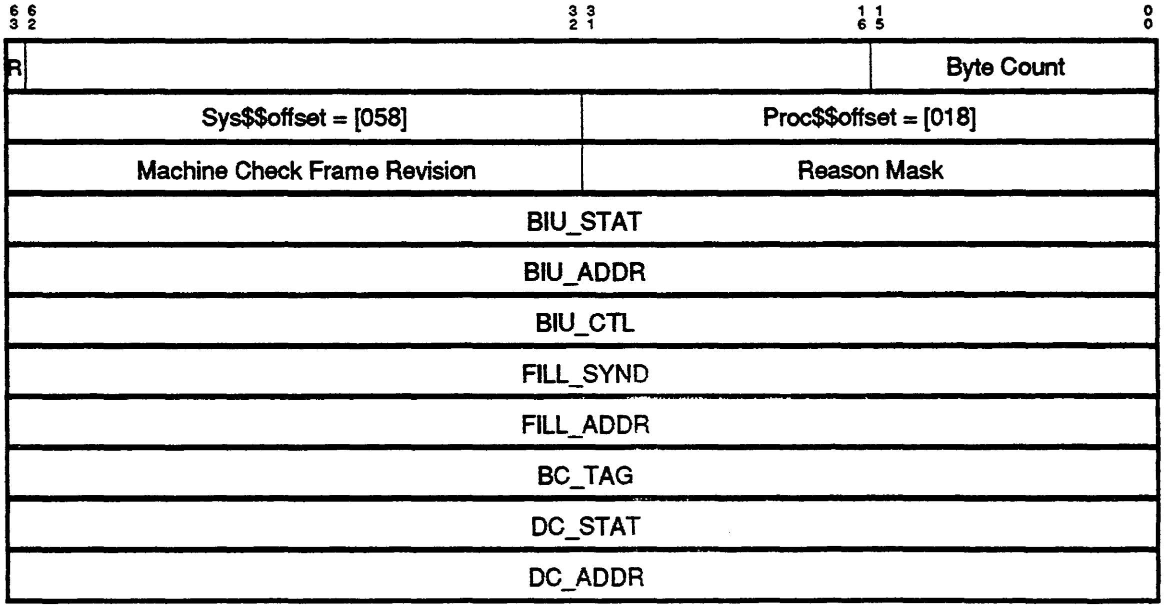
Figure 14-3 shows the format of the 630 machine check error log packet. Figure 14-4 shows the stack frame of the 630 error log packet.

Figure 14-3 630 Error Log Packet Format



BXB-0636-93

Figure 14-4 630 Stack Frame



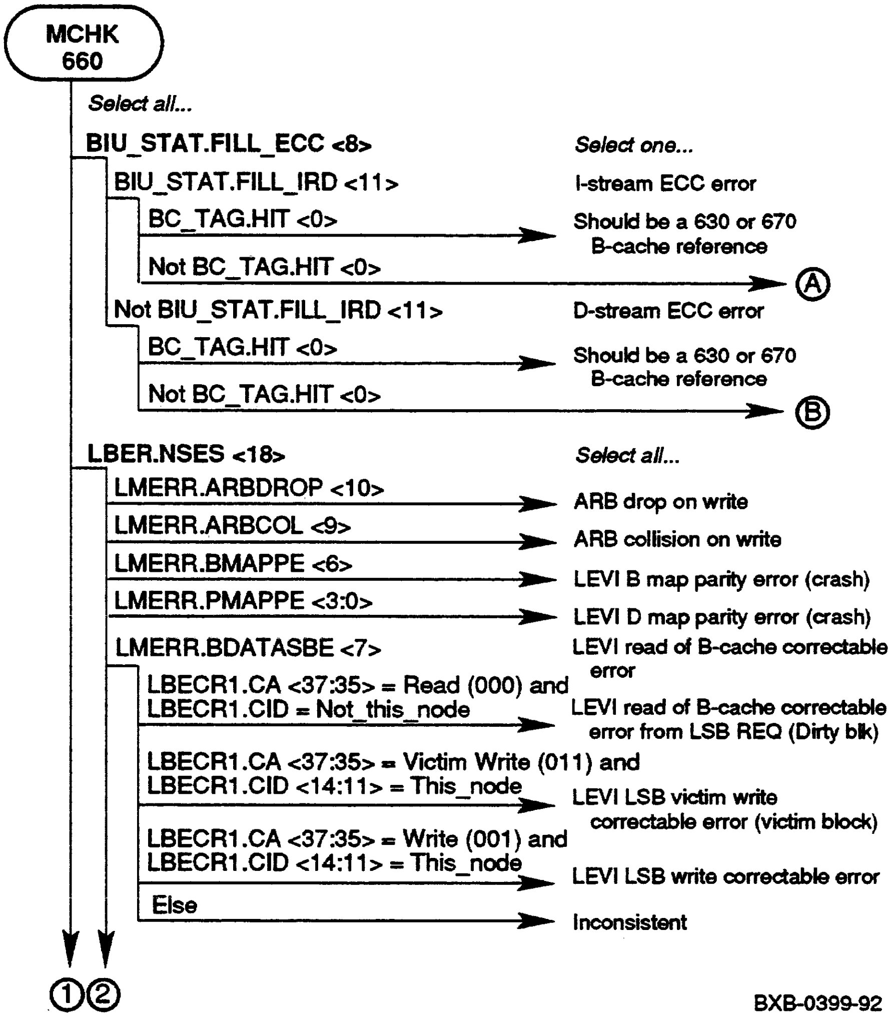
BXB-0637-93

14.3.2 Error Parse Trees

Parse trees (sorting diagrams) are used to represent how an error condition is examined by a deductive method. The parse tree indicates which registers and bits need to be checked to isolate and identify the error.

The technique is illustrated in Example 14-1, which shows a portion of a parse tree and assumes that LBER<NSES> is set (an error is detected).

Example 14-1 Error Isolation Using a Parse Tree



MCHK 660
Select all...
BIU_STAT.FILL_ECC <8>
Select one...
BIU_STAT.FILL_IRD <11>
I-stream ECC error
BC_TAG.HIT <0>
Should be a 630 or 670 B-cache reference
Not BC_TAG.HIT <0>
A
Not BIU_STAT.FILL_IRD <11>
D-stream ECC error
BC_TAG.HIT <0>
Should be a 630 or 670 B-cache reference
Not BC_TAG.HIT <0>
B
LBER.NSES <18>
Select all...
LMERR.ARBDROP <10>
ARB drop on write
LMERR.ARBCOL <9>
ARB collision on write
LMERR.BMAPPE <6>
LEVI B map parity error (crash)
LMERR.PMAPPE <3:0>
LEVI D map parity error (crash)
LMERR.BDATASBE <7>
LEVI read of B-cache correctable error
LBECR1.CA <37:35> = Read (000) and
LBECR1.CID = Not_this_node
LEVI read of B-cache correctable error from LSB REQ (Dirty blk)
LBECR1.CA <37:35> = Victim Write (011) and
LBECR1.CID <14:11> = This_node
LEVI LSB victim write correctable error (victim block)
LBECR1.CA <37:35> = Write (001) and
LBECR1.CID <14:11> = This_node
LEVI LSB write correctable error
Else
Inconsistent
1
2
BXB-0399-92

Many error conditions can cause LBER<NSES> to be set. To determine the exact cause of the error, follow the arrows that branch out of LBER<NSES>. This requires reading the LMERR register. If any one of bits <10>, <9>, <6>, or a single bit in the field <3:0> is set in this register, the source of the error is determined and no further inquiry is needed. However, if LMERR <7> is set, reading of the LBECR1 register is required. In this case, the states of bits LBECR1<CA> and LBCR1<CID> are used to derive the source of error, as shown in the diagram. If no bit-state combination that isolates a particular single-bit error is found, then an inconsistent error is indicated.

Table 14-2 lists the registers that report error conditions.

Table 14-2 Registers That Report Error Conditions

Register	Location	Address
BIU_STAT	DECchip 21064	Abox 10
DC_STAT	DECchip 21064	Abox 12
HIRR	DECchip 21064	Ibox 12
LBER	CPU module	BB[1] 0040
LMERR	CPU module	BB 0C40
LBECR1	CPU module	BB 0740
MERA	Memory module	BB 2140
IOP_LBECR1	IOP module	A00 0740

[1] BB is the node address of the module in hex.

14.3.3 Events Reported Through 670 Machine Checks

This section classifies and describes the errors that cause a 670 machine check. Figure 14-5 is the parse tree associated with the 670 machine check. Following the parse tree is a description of each type of error and, when possible, a suggested recovery method.

Figure 14-5 Processor Machine Check 670 Parse Tree

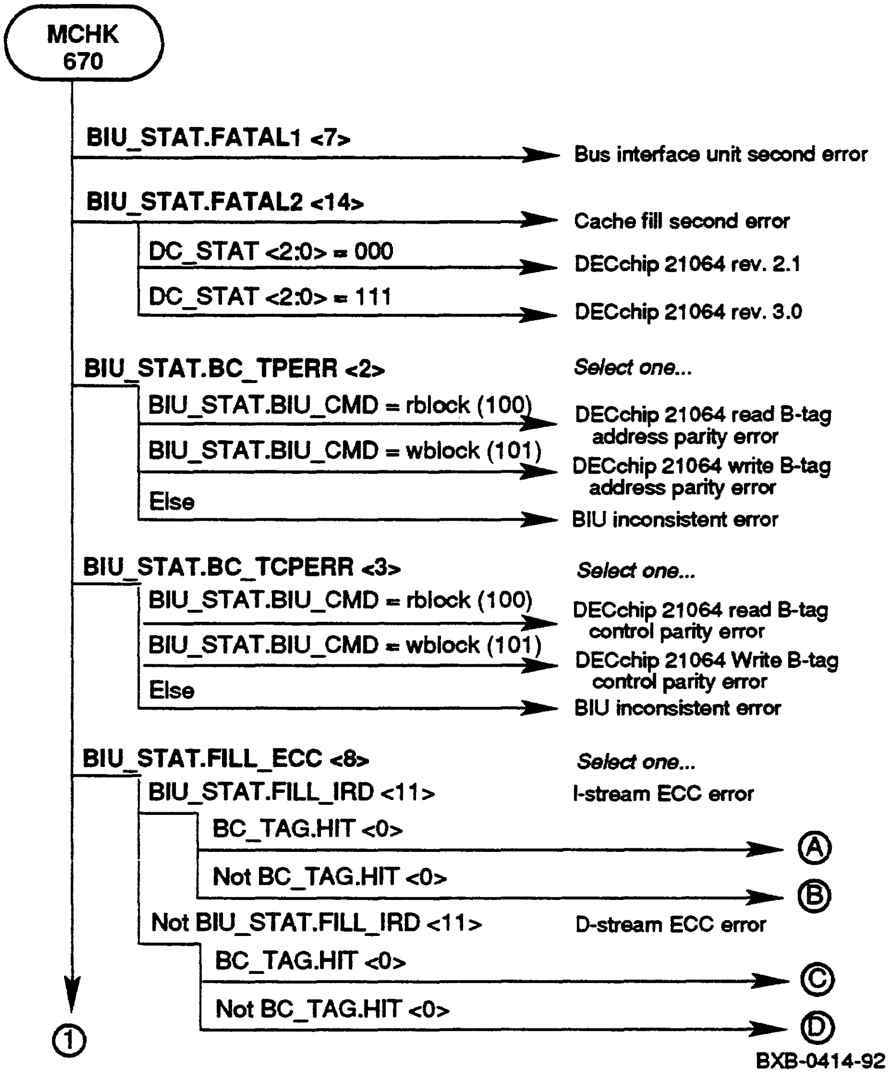


Figure 14-5 Processor Machine Check 670 Parse Tree (Continued)

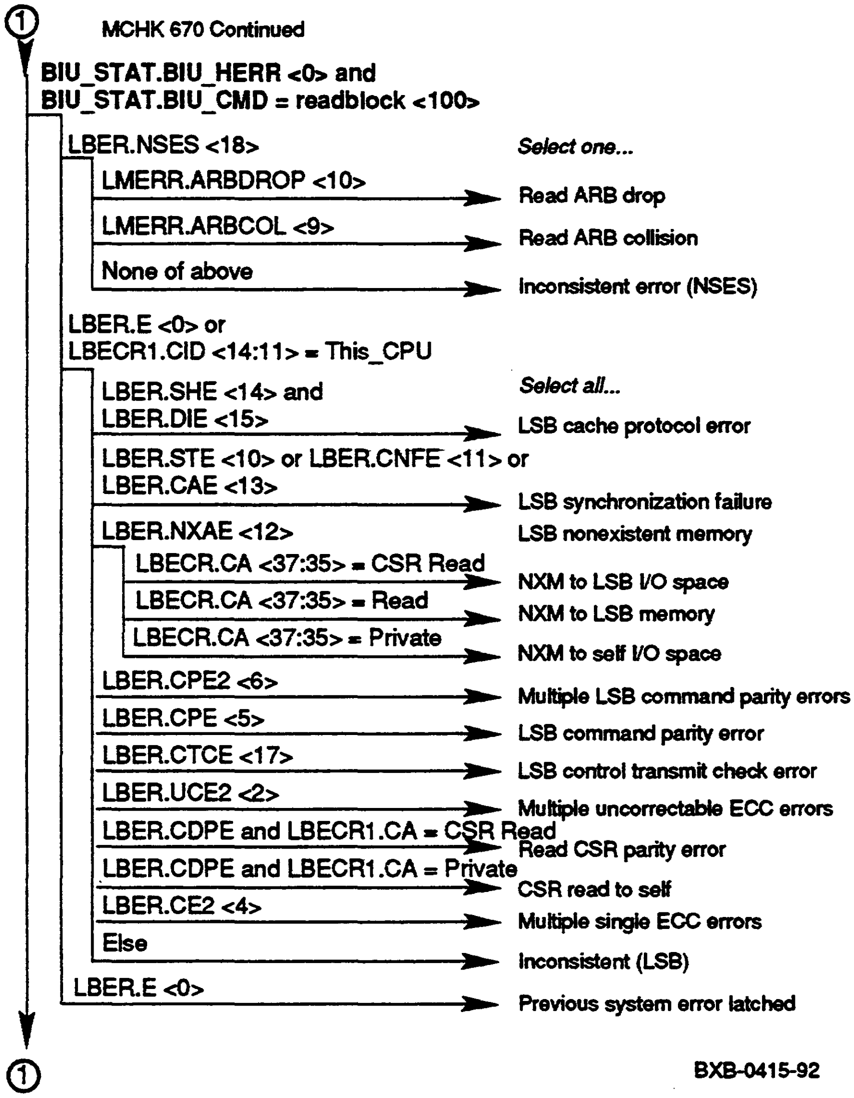


Figure 14-5 Processor Machine Check 670 Parse Tree (Continued)

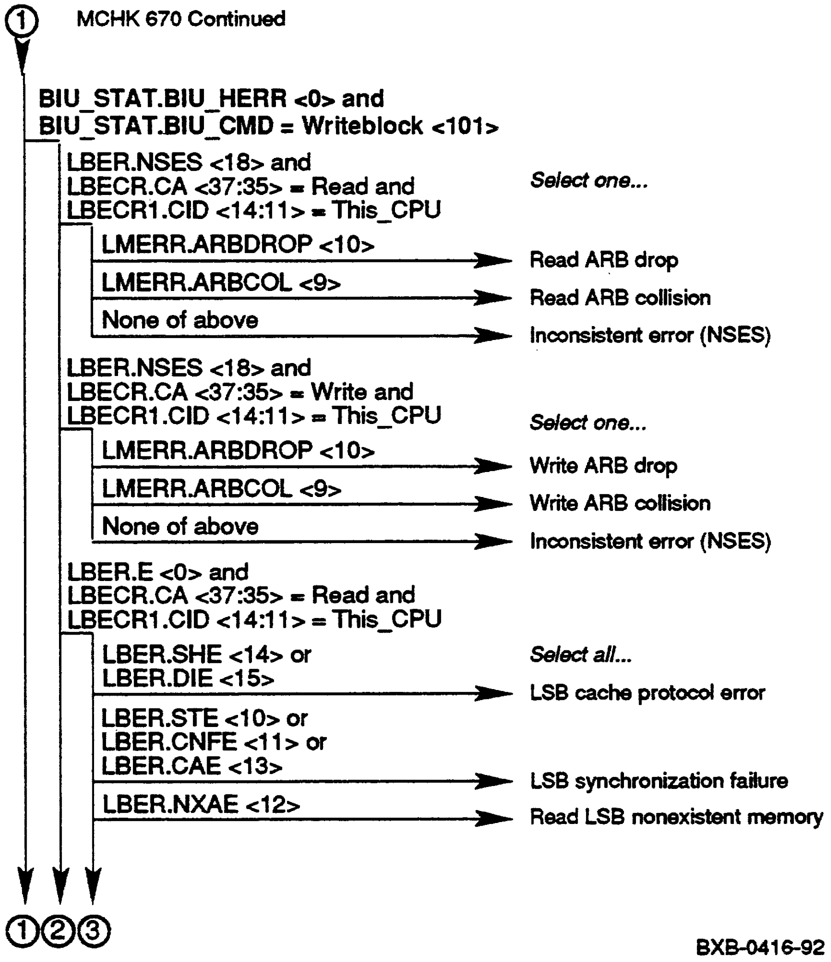
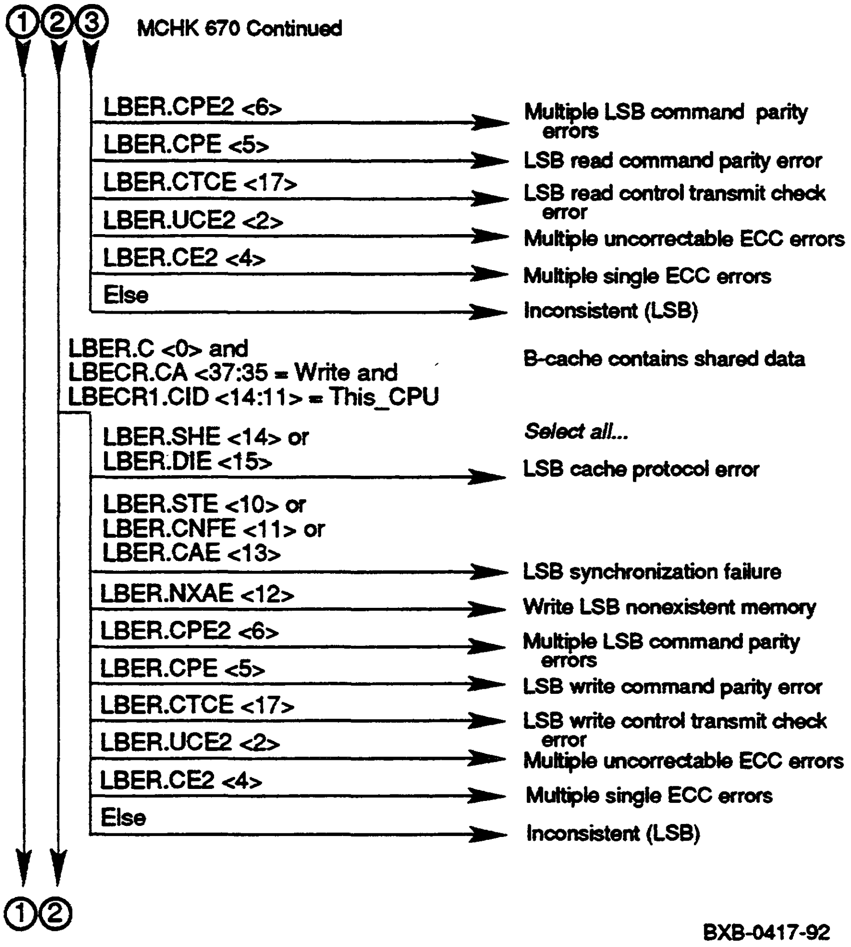


Figure 14-5 Processor Machine Check 670 Parse Tree (Continued)



①②③
MCHK 670 Continued
LBER.CPE2 <6>
Multiple LSB command parity errors
LBER.CPE <5>
LSB read command parity error
LBER.CTCE <17>
LSB read control transmit check error
LBER.UCE2 <2>
Multiple uncorrectable ECC errors
LBER.CE2 <4>
Multiple single ECC errors
Else
Inconsistent (LSB)
LBER.C <0> and
LBECR.CA <37:35 = Write and
LBECR1.CID <14:11> = This_CPU
B-cache contains shared data
Select all...
LBER.SHE <14> or
LBER.DIE <15>
LSB cache protocol error
LBER.STE <10> or
LBER.CNFE <11> or
LBER.CAE <13>
LSB synchronization failure
LBER.NXAE <12>
Write LSB nonexistent memory
LBER.CPE2 <6>
Multiple LSB command parity errors
LBER.CPE <5>
LSB write command parity error
LBER.CTCE <17>
LSB write control transmit check error
LBER.UCE2 <2>
Multiple uncorrectable ECC errors
LBER.CE2 <4>
Multiple single ECC errors
Else
Inconsistent (LSB)
①②
BXB-0417-92

Figure 14-5 Processor Machine Check 670 Parse Tree (Continued)

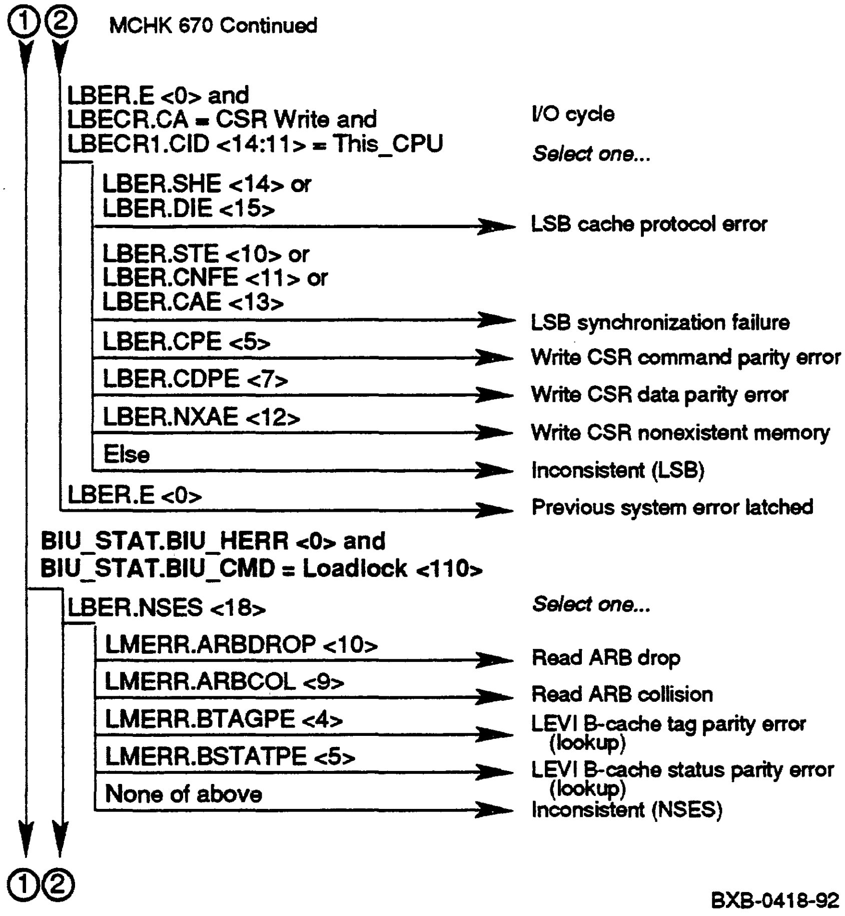


Figure 14-5 Processor Machine Check 670 Parse Tree (Continued)

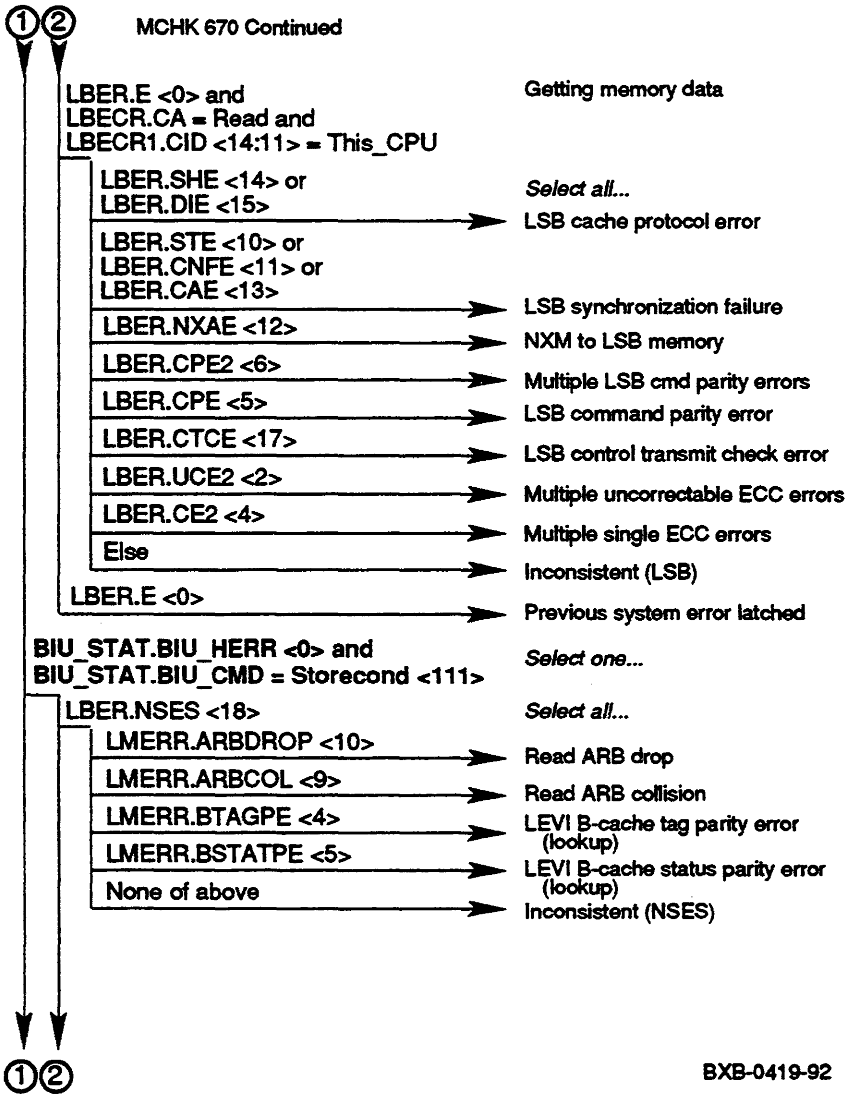


Figure 14-5 Processor Machine Check 670 Parse Tree (Continued)

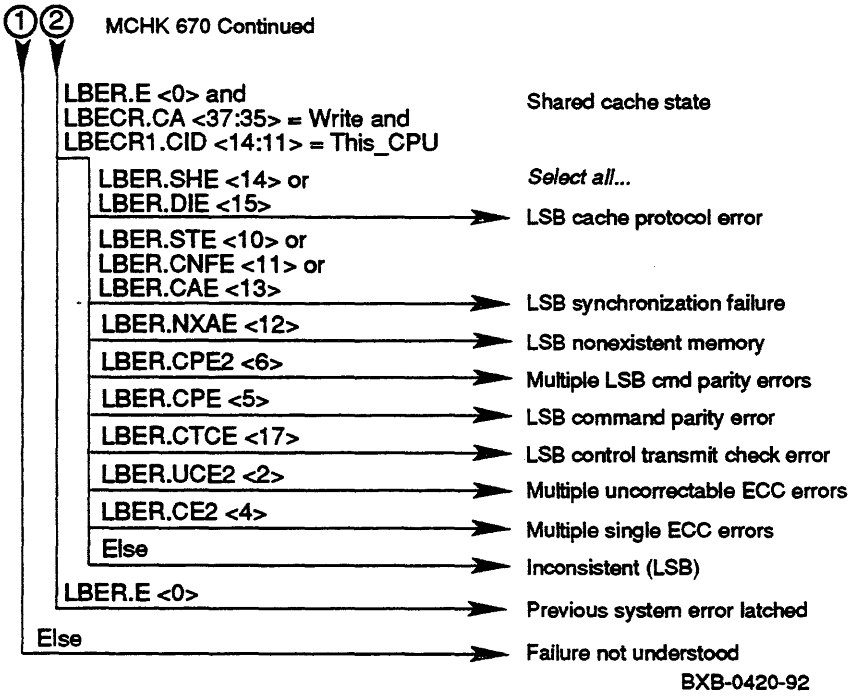
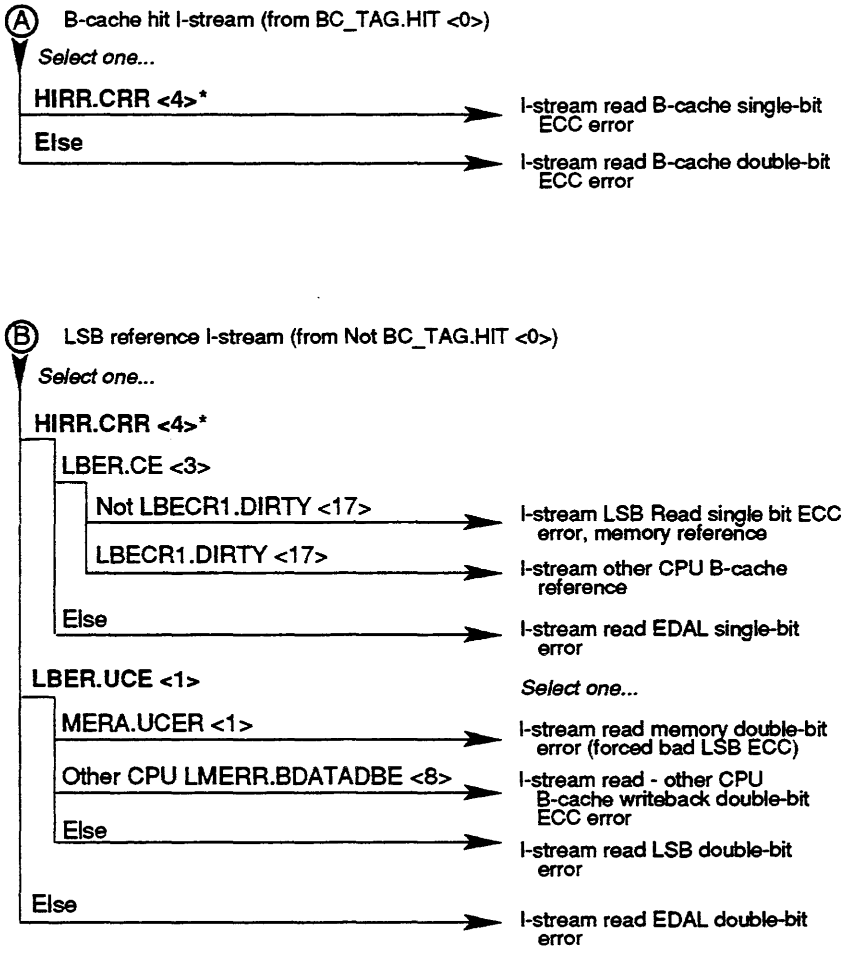


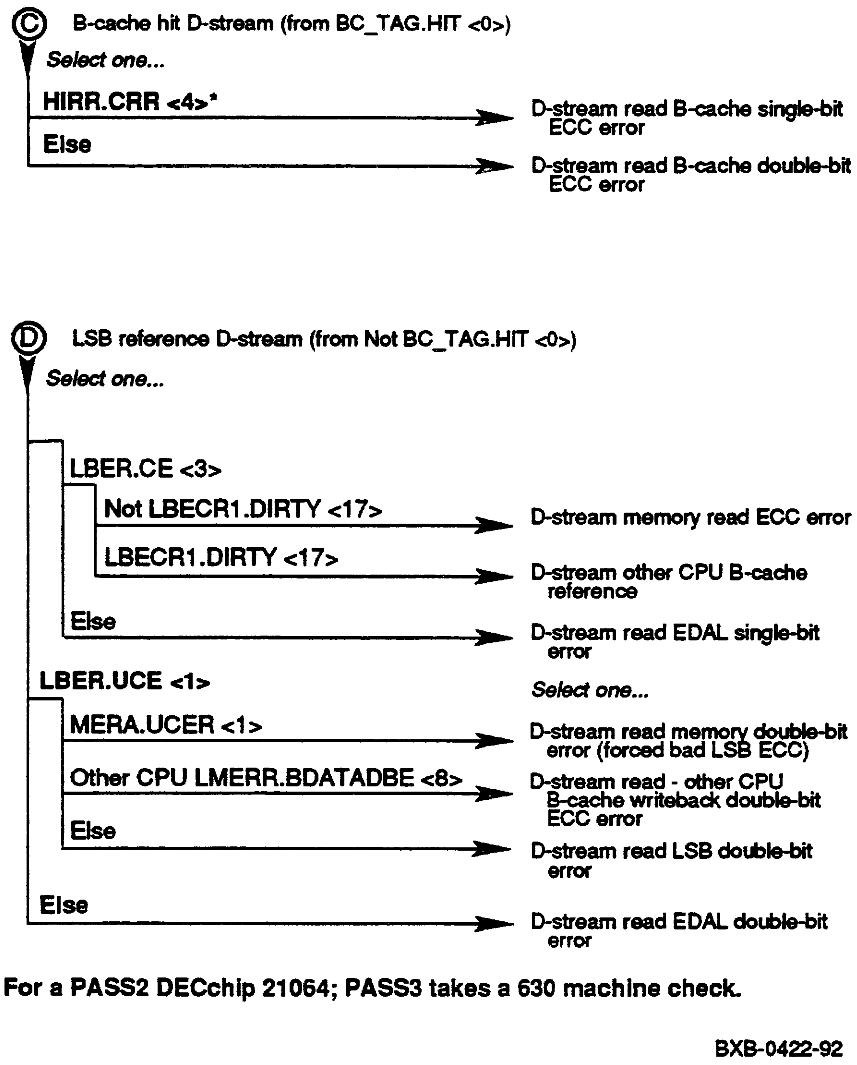
Figure 14-5 Processor Machine Check 670 Parse Tree (Continued)


A
B-cache hit I-stream (from BC_TAG.HIT <0>)
Select one...
HIRR.CRR <4>*
I-stream read B-cache single-bit ECC error
Else
I-stream read B-cache double-bit ECC error
B
LSB reference I-stream (from Not BC_TAG.HIT <0>)
Select one...
HIRR.CRR <4>*
LBER.CE <3>
Not LBECR1.DIRTY <17>
I-stream LSB Read single bit ECC error, memory reference
LBECR1.DIRTY <17>
I-stream other CPU B-cache reference
Else
I-stream read EDAL single-bit error
LBER.UCE <1>
Select one...
MERA.UCER <1>
I-stream read memory double-bit error (forced bad LSB ECC)
Other CPU LMERR.BDATADBE <8>
I-stream read - other CPU B-cache writeback double-bit ECC error
Else
I-stream read LSB double-bit error
Else
I-stream read EDAL double-bit error

* For a PASS2 DECchip 21064; PASS3 takes a 630 machine check.

BXB-0421-92

Figure 14-5 Processor Machine Check 670 Parse Tree (Continued)


C
B-cache hit D-stream (from BC_TAG.HIT <0>)
Select one...
HIRR.CRR <4>*
D-stream read B-cache single-bit ECC error
Else
D-stream read B-cache double-bit ECC error
D
LSB reference D-stream (from Not BC_TAG.HIT <0>)
Select one...
LBER.CE <3>
Not LBECR1.DIRTY <17>
D-stream memory read ECC error
LBECR1.DIRTY <17>
D-stream other CPU B-cache reference
Else
D-stream read EDAL single-bit error
LBER.UCE <1>
Select one...
MERA.UCER <1>
D-stream read memory double-bit error (forced bad LSB ECC)
Other CPU LMERR.BDATADBE <8>
D-stream read - other CPU B-cache writeback double-bit ECC error
Else
D-stream read LSB double-bit error
Else
D-stream read EDAL double-bit error
BXB-0422-92

* For a PASS2 DECchip 21064; PASS3 takes a 630 machine check.

BIU Second Error

Description: This bit sets when an external cycle is terminated with the cAck_h pins indicating hard error or when a B-cache tag probe encounters bad parity in the tag address or control RAM while an error bit which locks the BIU_STAT register was already set. Having this bit set indicates that the system state for the second error was lost.

Recovery procedure: Since system state has been lost for at least one of these fill errors recovery is not possible.

Restart condition: None. Terminate the session.

Error logging: Since this is just an additional status bit, no error logging is suggested here. Do the error logging according to the first error that was latched. Set software flag bit 95.

Cache Fill Second Error

Description: The meaning of this error bit is different, based on the revision of the DECchip 21064. If the DECchip 21064 is rev 2.1, the occurrence of this error indicates that multiple primary cache fill errors have occurred. Any fill error after the first will be lost. Recovery is not possible. If the DECchip 21064 is rev 3.0, the occurrence of this bit means that a multi-bit ECC error has been detected during a CPU chip cache fill.

Recovery procedure: Since system state has been lost for at least one fill, error recovery is not possible.

Restart condition: None. Terminate the session.

Error logging: No error logging is suggested here. Set software flag bit 94. The correct error logging of each case will be handled later on in the parse tree.

DECchip 21064 Read B-Tag Address Parity Error

Description: During a CPU B-cache lookup, the CPU detected an address parity error within the tag. BC_TAG holds the results of the probe. The physical address is latched in the BIU_ADDR register.

Recovery procedure: Software can attempt to recover the correct tag and parity by looking at the same index into the B-map. The correct tag can then be loaded via the LTAGW register. However, this is not advisable and software should terminate the session.

Restart condition: Crash the system.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 1. Also save the BIU_STAT and BIU_ADDR to the EEPROM area reserved for machine check 670 detected tag and status errors.

DECchip 21064 Write B-Tag Address Parity Error

Description: During a CPU B-cache lookup, the CPU detected an address parity error within the tag. BC_Tag holds the results of the probe. The physical address is latched in the BIU_ADDR register.

Recovery procedure: Software can attempt to recover the correct tag and parity by looking at the same index into the B-map. The correct tag can then be loaded via the LTAGW register. However, this is not advisable and software should terminate the session.

Restart condition: Crash the system.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 2. Also save the BIU_STAT and BIU_ADDR to the EEPROM area reserved for machine check 670 detected tag and status errors.

BIU Inconsistent Error

Description: During a CPU B-cache lookup, the CPU detected either an address parity error or control parity error within the tag. However, with this error, the BIU_CMD was not a type that should be doing a probe.

Recovery procedure: None.

Restart condition: None. System state appears corrupt. Terminate the session.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 3.

DECchip 21064 Read B-Tag Control Parity Error

Description: During a CPU B-cache lookup for a DECchip 21064 read, the CPU detected a control parity error within the tag. BC_Tag holds the results of the probe. The physical address is latched in the BIU_ADDR register.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 4. Also save the BIU_STAT and BIU_ADDR to the EEPROM area reserved for machine check 670 detected tag and status errors.

DECchip 21064 Write B-Tag Control Parity Error

Description: During a CPU B-cache lookup for a DECchip 21064 write, the CPU detected a control parity error within the tag. BC_Tag holds the results of the probe. The physical address is latched in the BIU_ADDR register.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 5. Also save the BIU_STAT and BIU_ADDR to the EEPROM area reserved for machine check 670 detected tag and status errors.

I-Stream Read B-Cache Single-Bit ECC Error

Description: During an I-stream reference with a B-cache hit, the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. With a pass 2 DECchip 21064, this error will be fatal. No correction is accomplished. With a pass 3 DECchip 21064, you will have taken the 630 error path and would not be here. These error parse branches are in the machine check parse flow because the DEC 7000 system initially shipped with the rev 2 DECchip 21064.

Recovery procedure: None.

Restart condition: Restart if corrected by PALcode or hardware. Terminate the user or system.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 6.

I-Stream Read B-Cache Double-Bit ECC Error

Description: During an I-stream Reference with a B-cache hit, the DECchip 21064 detected an uncorrectable ECC error. The failing syndrome is latched in the FILL_SYND register. This is a double-bit error, and no correction can be performed.

Recovery procedure: None.

Restart condition: Terminate the session.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 7.

I-Stream LSB Read Single-Bit ECC Error, Memory Reference

Description: During an I-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. With a pass 2 DECchip 21064, this error will be fatal. No correction is accomplished. With a pass 3 DECchip 21064, you will have taken the 660 error path and would not be here. These error parse branches are left in the machine check parse flow because the DEC 7000 system initially shipped with the rev 2 DECchip 21064.

From parsing the error, it was found that the bus cycle associated with this error had a CE error. Also, the dirty bit in LBECR1 was clear which implies that a memory supplied the data. Operating system software should look and figure out which memory controller is associated with the latched address and append a memory controller subpacket from the associated memory.

Recovery procedure: None.

Restart condition: Restart if corrected by PALcode or hardware. Terminate the user or session.

Error logging: For this error, the basic machine check entry with an LSB subpacket and an LMA subpacket will be required. Set software flag bit 8, 96 (LSB), and 97 (LMA) subpacket present bits.

Additional parsing: Memory address correlation.

I-Stream Read Other CPU B-Cache Single-Bit ECC Error

Description: During an I-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. With a pass 2 DECchip 21064, this error will be fatal. No correction is accomplished. With a pass 3 DECchip 21064, you will have taken the 660 error path and would not be here. These error parse branches are left in the machine check parse flow because the DEC 7000 system initially shipped with the pass 2 DECchip 21064. Note that this error was caused by a B-cache single-bit error that was sourced from another CPU node. The other CPU will be attempting to parse this error via its 660 error handler.

Recovery procedure: None.

Restart condition: Restart if corrected by PALcode. Terminate the session.

Error logging: For this error, the basic machine check entry with an LSB subpacket will be required.. Set software flag bit 9 and 96 (LSB).

Additional parsing: BDATASBE flow.

I-Stream Read EDAL Single-Bit ECC Error

Description: During an I-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. With a pass 2 DECchip 21064, this error will be fatal. No correction is accomplished. With a pass 3 DECchip 21064, you will have taken the 630 error path and would not be here. These error parse branches are left in the machine check parse flow because the DEC 7000 system initially shipped with the rev 2 DECchip 21064. Note that this error was caused by the EDAL data path. Note there were no LSB errors on this data transfer.

Recovery procedure: None.

Restart condition: Terminate the session.

Error logging: For this error, the basic machine check entry is fine. Set software flag bit 11.

I-Stream Read Memory Double-Bit ECC Error

Description: During an I-stream reference the DECchip 21064 detected an uncorrectable ECC error. The failing syndrome is latched in the FILL_SYND register. This is fatal to the user or system as determined by the operating system. This error was caused by a memory RAM double-bit error.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry with an LMA subpacket (the one associated with the error) will be required. Set software flag bit 12 and 97, the LMA subpacket present bit.

Additional parsing: Memory address correlation.

I-Stream Read Other CPU B-Cache Double-Bit ECC Error

Description: During an I-stream reference the DECchip 21064 detected an uncorrectable ECC error. The failing syndrome is latched in the

FILL_SYND register. This is fatal to the user or system as determined by the operating system. Note that this error was caused by another CPU's B-cache double-bit error. The other CPU will be attempting to parse this error via its 660 error handler.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry with an LSB subpacket will be required. Set software flag bit 13 and 96 (LSB subpacket).

Additional parsing: BDATADBE flow.

I-Stream Read LSB Double-Bit ECC Error

Description: During an I-stream reference the DECchip 21064 detected an uncorrectable ECC error. The failing syndrome is latched in the FILL_SYND register. This is fatal to the user or system as determined by the operating system. Note that this error was caused by the LSB.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry with an LSB subpacket will be required. Set software flag bit 14 and 96 (LSB subpacket).

Additional parsing: Memory address correlation.

I-Stream Read EDAL Double-Bit ECC Error

Description: During an I-stream reference the DECchip 21064 detected an uncorrectable ECC error. The LSB was the source of the data. However, the data was correct (no errors) off of the LSB. Because of this, it is assumed the EDAL data path broke the data. The failing syndrome is latched in the FILL_SYND register. This is fatal to the user or system as determined by the operating system.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry is fine. Set software flag bit 15.

D-Stream Read B-Cache Single-Bit ECC Error

Description: During a D-stream reference with a B-cache hit, the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. With a pass 2 DECchip 21064, this error will be fatal. No correction is accomplished. With a pass 3 DECchip 21064, you will have taken the 630 error path and would not be here. These error parse branches are left in the machine check parse flow because the DEC 7000 system initially shipped with the rev 2 DECchip 21064.

Recovery procedure: None.

Restart condition: Terminate the session.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 16.

D-Stream Read B-Cache Double-Bit ECC Error

Description: During a D-stream reference with a B-cache hit, the DECchip 21064 detected an uncorrectable ECC error. The failing syndrome is latched in the FILL_SYND register. Note that this is a double-bit error, and no correction can be performed.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 17.

D-Stream LSB Read Single-Bit ECC Error, Memory Reference

Description: During a D-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. With a pass 2 DECchip 21064, this error will be fatal. No correction is accomplished. With a pass 3 DECchip 21064, you will have taken the 660 error path and would not be here. These error parse branches are left in the machine check parse flow because the DEC 7000 system initially shipped with the pass 2 DECchip 21064. Note that this error was caused by an LSB single-bit error in which a memory was the source.

Recovery procedure: None.

Restart condition: Terminate the session.

Error logging: For this error, the basic machine check entry with an LMA subpacket (the one associated with the error) will be required. Set software flag bit 18 and 97, the LMA subpacket present bit.

Additional parsing: Memory address correlation.

D-Stream Read Other CPU B-Cache Single-Bit ECC Error

Description: During a D-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. With a pass 2 DECchip 21064, this error will be fatal. No correction is accomplished. With a pass 3 DECchip 21064, you will have taken the 660 error path and would not be here. These error parse branches are-left in the machine check parse flow because the DEC 7000 system initially shipped with the rev 2 DECchip 21064. Note that this error was caused by another CPU's B-cache. The other CPU will be attempting to parse this error via its 660 error handler.

Recovery procedure: None.

Restart condition: Terminate the session.

Error logging: For this error, the basic machine check entry with an LSB subpacket will be required. Set software flag bit 19 and 96, the LSB subpacket present bit.

Additional parsing: BDATASBE flow.

D-Stream Read EDAL Single-Bit Error

Description: During a D-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. With a pass 2 DECchip 21064, this error will be fatal. No correction is accomplished. With a pass 3 DECchip 21064, you will have taken the 630 error path and would not be here. These error parse branches are left in the machine check parse flow because the DEC 7000 system initially shipped with the pass 2 DECchip 21064. Note that this error was caused by the EDAL data path. There were no LSB errors.

Recovery procedure: None.

Restart condition: Restart if corrected by PALcode or hardware. Terminate the user or system.

Error logging: For this error, the basic machine check entry is fine. Set software flag bit 21.

D-Stream Read Memory Double-Bit ECC Error

Description: During a D-stream reference the DECchip 21064 detected an uncorrectable ECC error. The failing syndrome is latched in the FILL_SYND register. This is fatal to the user or system as determined by the operating system. This error was caused by a memory RAM double-bit error.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry with an LMA subpacket (the one associated with the error) will be required. Set software flag bit 22 and 97, the LMA subpacket present bit.

Additional parsing: Memory address correlation.

D-Stream Read Other CPU B-Cache Double-Bit ECC Error

Description: During a D-stream reference the DECchip 21064 detected an uncorrectable ECC error. The failing syndrome is latched in the FILL_SYND register. This is fatal to the user or system as determined by the operating system. Note that this error was caused by a B-cache double-bit error which was sourced from another CPU node. The other CPU will be attempting to parse this error via its 660 error handler.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry with an LSB subpacket will be required. Set software flag bit 23 and 96,the LSB subpacket present bit.

D-Stream Read LSB Double-Bit ECC Error

Description: During a D-stream reference the DECchip 21064 detected an uncorrectable ECC error. The failing syndrome is latched in the FILL_SYND register. This is fatal to the user or system as determined by the operating system. Note that this error was caused by the LSB.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry with an LSB subpacket will be required. Set software flag bit 24 and 96, the LSB subpacket present bit.

Additional parsing: Memory address correlation.

D-Stream Read EDAL Double-Bit ECC Error

Description: During a D-stream reference the DECchip 21064 detected an uncorrectable ECC error. The failing syndrome is latched in the FILL_SYND register. This is fatal to the user or system as determined by the operating system. Note that this error was caused by the EDAL data path. The source of the data was the LSB but the UCE bit was not set, so the data came off the LSB with no error.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry is fine. Set software flag bit 25.

Read Arbitration Drop

Description: During a CPU readblock command, the LEVI arbitrated for the bus, assumed someone else won based on the requesting nodes, and then saw that the bus did not have a command address cycle asserted. This is a fatal error condition.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check along with an LSB subpacket will be sufficient. Set software flag bits 96 (LSB subpacket), 40 (NSES), and 41 (arbdrop).

Read Arbitration Collision

Description: An LSB arbitration collision was detected by the LEVI while the LEVI was attempting to go get read data to do a B-cache fill in an attempt to satisfy a read request from the CPU. Arbitration collision is considered fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check along with an LSB subpacket will be sufficient. Set software flag bits 96 (LSB subpacket), 40 (NSES), and 42 (arbcol).

Inconsistent Error—NSES

Description: During a DECchip 21064 readblock command, the Node-Specific Error bit in the LBER register was set. However, no supporting error bits were set to further isolate the cause of the error. This is an inconsistent state that is considered fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check along with an LSB subpacket will be sufficient. Set software flag bits 96 (LSB subpacket), 40 (NSES), and 45 (inconsistent).

LSB Cache Protocol Error

Description: During an outstanding CPU readblock request, the LEVI detected an ilbranchal assertion of either Shared or Dirty by another CPU node. This is considered fatal to the system. Cache state is probably corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check along with an LSB subpacket will be sufficient. Set software flag bits 96 (LSB subpacket), 48 (LSB ERR Read), 31 (DECchip 21064 readblock), and 51/52 (Shared/Dirty) if appropriate.

LSB Synchronization Failure

Description: During an outstanding CPU readblock request, the LEVI detected either a stall error, confirmation error, or command/address error. These errors imply that LSB synchronization was lost. These errors probably signify that some other internal node errors have occurred elsewhere in the system. These are considered fatal errors.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check with an LSB snapshot will be fine. Set software flag bits 96 (LSB subpacket), 31 (readblock), 48 (LSB ERR read), and 53/54/55 (stall, confirmation, command/address) if appropriate.

NXM to LSB I/O Space

Description: During an outstanding DECchip 21064 readblock command, the LEVI detected that the LSB request did not get a confirmation. This results in an NXM. The command on the LSB was a CSR read so this read was actually going to an LSB I/O address. Use the latched address in the LBECSR register to determine which I/O address the request was sent to.

Recovery procedure: Clear error bits.

Restart condition: Restart the read if the address was to an existent piece of hardware.

Error logging: For this error, a basic machine check with an LSB subpacket will do. If the I/O address was to the IOP, then provide a log adapter subpacket also. Set software flag bits 96 (LSB), 31 (DECchip 21064 readblock), 47 (LSB ERR Read CSR), 56 (NXM CSR Read). If the IOP is the target address, then set software flag bit 98 (log adapter present) bit also.

Additional parsing: I/O address parse.

NXM to LSB Memory

Description: During an outstanding DECchip 21064 readblock command, the LEVI detected that the LSB request did not get a confirmation. This results in an NXM. The command on the LSB was a read, so this was going to an LSB MEM address. Use the latched address in the LBECSR register to determine if this is a valid address. If the address is valid, it would appear that a memory is at fault. If the address is not valid, this would lean toward either a software problem or a hardware address generation problem.

Recovery procedure: None

Restart condition: If expected, continue, else crash the system.

Error logging: The basic machine check, LSB subpacket and LMA subpacket will be provided. Set software flag bits 96 (LSB subpacket), 97 (LMA subpacket), 31 (DECchip 21064 readblock), 48 (LSB ERR read), 57 (NXM-mem read).

Additional parsing: Memory address correlation.

NXM to Self I/O Space

Description: This CPU executed an I/O space read to its own I/O space. These transactions use the LSB. In this case the read did not get confirmation on the LSB.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error a basic machine check entry will do. Set software flag bits 31 (DECchip 21064 readblock), 46 (LSB ERR, private), 58 (NXM, private space).

Multiple LSB Command Parity Errors

Description: This branch of the parse tree is for information purposes only. It just shows that multiple command parity errors have occurred.

Recovery procedure: None.

Restart condition: None.

Error logging: For this branch, just set software flag bit 59 which indicates that this error has occurred.

LSB Command Parity Errors

Description: An LSB command parity error was detected on a bus cycle during which this CPU was the commander. This results in a cAck_h hard error being sent to the DECchip 21064, which causes a machine check through 670.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and an LSB subpacket. Set software flags 96 (LSB), 60 (CPE), 48 (LSB ERR, read), 31 (DECchip 21064 readblock).

LSB Control Transmit Check Error

Description: This module detected that an LSB control line(s) it was driving did not match with what it had seen on the LSB bus. This is a fatal condition. Cache coherence could be lost.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check and an LSB snapshot is required. Set software flag bits 96 (LSB subpacket), 48 (LSB ERR, read), 31 (DECchip 21064 readblock), and 61 (CTCE).

Multiple Uncorrectable ECC Errors

Description: Multiple double-bit ECC errors have been detected on the LSB bus. The error state for all errors occurring after the first is lost. This is fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: Set software bit 62. Other branches of the parse tree will describe what logging to do and also what other appropriate software bits to set.

CSR Data Parity Error

Description: During a read to an LSB-based I/O register, the data cycle contained a parity error. This error could have occurred when referencing another LSB node or to the node doing the read. In the first case, the LSB command would be a CSR read. In the latter case, the LSB command would be private.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, log the basic machine check, an LSB snapshot and either an LMA subpacket or log adapter subpacket depending on where the I/O address pointed to. If it was to another CPU, the LSB snapshot will be sufficient. Set software flags 96 (LSB), 63 (CDPE), 31 (DECchip 21064 read), and 47 (LSB Read CSR) or 46 (LSB private) depending on the LSB cycle type. If the I/O address was to the IOP (node 8) set software flag 98 (log adapter subpacket). If the I/O address is a memory, log an LMA subpacket and set software flag 97 (LMA subpacket).

Additional parsing: I/O address correlation.

Multiple Single-Bit ECC Errors

Description: Multiple single-bit LSB ECC errors have been detected on the LSB bus. These could be caused by a variety of error conditions. The error state for all errors occurring after the first is lost. For systems with a rev 2 DECchip 21064, this is fatal because the source and destination of the error cannot be determined for the subsequent errors. For systems with rev 3 DECchip 21064s, CE2 is NOT fatal. All recipients of the data perform correction, thus all data with single-bit errors will have been corrected.

The system still loses the information for error state determination but can resume operation because all data is corrected by hardware.

Recovery procedure: None.

Restart condition: None.

Error logging: Error logging will be done by another parse tree branch. Just set software flag bit 64 if this branch is true.

Inconsistent Error—LSB

Description: Attempting to parse the possible error conditions for a machine check while the DECchip 21064 has an outstanding readblock. It was found that no error cases were present that should have caused the system to machine check.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check. Set software flag bit 70 (inconsistent error), 31 (readblock), and 48 (LSB ERR, read).

Previous System Error Latched

Description: When parsing the reasons for a machine check, it was found that the latched LSB bus state did not correspond to the CPU detecting the machine check. It is assumed that a previous LSB error has latched all the bus registers. Multiple errors must have occurred.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and an LSB snapshot. Set software flag bits 96 (LSB), 31 (DECchip 21064 readblock), and 69 (previous system error).

Read Arbitration Drop

Description: An LSB arbitration drop was detected by the LEVI while the LEVI was attempting to get read data to do a B-cache fill to satisfy a write request from the CPU. Arbitration drop is considered fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 32 (DECchip 21064 writeblock), 48 (LSB ERR, read), 40 (NSES), and 41 (arbdrop).

Read Arbitration Collision

Description: An LSB arbitration collision was detected by the LEVI while the LEVI was attempting to get read data to do a B-cache fill to satisfy a write request from the CPU. Arbitration collision is considered fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 32 (DECchip 21064 writeblock), 48 (LSB ERR, read), 40 (NSES), and 42 (arbcol).

Inconsistent Error—NSES

Description: During the LEVI sourcing data from the LSB to satisfy a DECchip 21064 writeblock, the NSES bit was set indicating that an internal KN7AA error was detected. However, no supporting error bits were set to indicate this condition. This is an inconsistent error condition and is fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 32 (DECchip 21064 writeblock), 48 (LSB ERR, read), 40 (NSES), and 45 (inconsistent).

Write Arbitration Drop

Description: An LSB arbitration drop was detected by the LEVI while the LEVI was attempting to write data to the LSB to satisfy a write request from the CPU. Arbitration drop is considered fatal. Note that this write was going to the LSB because the B-cache contained shared data.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 32 (DECchip 21064 writeblock), 49 (LSB ERR, write), 40 (NSES), and 41 (arbdrop).

Write Arbitration Collision

Description: An LSB arbitration collision was detected by the LEVI while the LEVI was attempting to write data to the LSB to satisfy a write request from the CPU. Arbitration collision is considered fatal. Note that this write was going to the LSB because the B-cache contained shared data.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 32 (DECchip 21064 writeblock), 49 (LSB ERR, write), 40 (NSES). and 42 (arbcol).

Inconsistent Error—NSES

Description: During the LEVI attempting to write data to the LSB to satisfy a DECchip 21064 writeblock, the NSES bit was set indicating that an internal KN7AA error was detected. However, no supporting error bits were set to indicate this condition. This is an inconsistent error condition and is fatal. Note that this write was going onto the LSB because the B-cache contained shared data.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 32 (DECchip 21064 writeblock), 49 (LSB ERR, write), 40 (NSES), and 45 (inconsistent).

LSB Cache Protocol Error

Description: During an outstanding CPU writeblock request, the LEVI, when attempting to read LSB data, detected an assertion of either Shared or Dirty by another node. This is considered fatal to the system. Cache state is probably corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check along with an LSB subpacket will be sufficient. Set software flag bits 96 (LSB subpacket), 48 (LSB ERR, read), 32 (writeblock), and 51/52 (Shared/Dirty error) if appropriate.

LSB Synchronization Failure

Description: During an outstanding CPU writeblock request, the LEVI detected either a stall error, confirmation error, or command/address error while the LEVI was attempting to do an LSB read. These errors imply loss of LSB synchronization. They probably signify that some other internal node errors have occurred elsewhere in the system. These are considered fatal errors.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check with an LSB snapshot will be fine. Set software flag bits 96 (LSB subpacket), 32 (DECchip 21064 writeblock), 48 (LSB ERR, read), and 53/54/55 (stall, confirmation, command/address) if appropriate.

Read LSB Nonexistent Memory

Description: During an outstanding DECchip 21064 writeblock command, the LEVI detected that the LSB read request did not get a confirmation. This results in an NXM. The command on the LSB was a read so this read was actually going to an LSB MEM address. Use the latched address in the LBECSR register to determine if this is a valid address. If the address is valid, it would appear that a memory is at fault. If the address is not valid, this would lean toward either a software problem or a hardware address generation problem. If the address is within memory range, include the memory registers for the associated memory controller.

Recovery procedure: None.

Restart condition: If expected, continue, else crash the system.

Error logging: The basic machine check, LSB subpacket and LMA subpacket will be provided. Set software flag bits 96 (LSB subpacket), 97

(LMA subpacket), 32 (DECchip 21064 writeblock), 48 (LSB ERR, read), and 57 (NXM-mem read).

Additional parsing: Memory address correlation.

LSB Command Parity Errors

Description: An LSB command parity error was detected on a bus cycle in which this CPU was the commander. This condition results in CACK HERR being sent to the DECchip 21064, which causes a machine check through 670.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and an LSB subpacket. Set software flags 96 (LSB), 60 (CPE), 48 (LSB ERR, read), and 32 (DECchip 21064 writeblock).

LSB Control Transmit Check Error

Description: This module detected that an LSB control line(s) it was driving did not match with what it had seen on the LSB bus. This is a fatal condition. Cache coherence could be lost.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check and an LSB snapshot is required. Set software flag bits 96 (LSB), 48 (LSB ERR, read), 32 (DECchip 21064 writeblock), and 61 (CTCE).

Inconsistent—LSB

Description: Attempting to parse the possible error conditions for a machine check while the DECchip 21064 has an outstanding writeblock, it was found that no error cases were present which would have caused the system to machine check.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check. Set software flag bit 32 (DECchip 21064 writeblock), 48 (LSB ERR, read), and 70 (inconsistent error).

LSB Cache Protocol Error

Description: During an outstanding CPU writeblock request, the LEVI when attempting to write LSB data detected an assertion of either Shared or Dirty by another node. This is considered fatal to the system. Cache state is probably corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check along with an LSB subpacket will be sufficient. Set software flag bits 96 (LSB subpacket), 49

(LSB ERR, write), 32 (writeblock), and 51/52 (Shared/Dirty error) based on the bit(s) set.

LSB Synchronization Failure

Description: During an outstanding CPU writeblock request, the LEVI detected either a stall error, confirmation error, or command/address error while the LEVI was attempting to do an LSB write. These errors imply loss of LSB synchronization. They probably signify that some other internal node errors have occurred elsewhere in the system. These are considered fatal errors.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check with an LSB snapshot will be fine. Set software flag bits 96 (LSB subpacket), 32 (writeblock), 49 (LSB ERR, write), and 53/54/55 (stall, confirmation, command/address) if appropriate.

Write LSB Nonexistent Memory

Description: During an outstanding DECchip 21064 writeblock command, the LEVI detected that the LSB write request did not get a confirmation. This condition results in an NXM. The command on the LSB was a write, so this was actually going to an LSB MEM address. Use the latched address in the LBECSR register to determine if this is a valid address. If the address is valid, it would appear that a memory is at fault. If the address is not valid, this would lean toward either a software problem or a hardware problem. If the address is within memory range, include the memory registers for the associated memory controller.

Recovery procedure: None.

Restart condition: None.

Error logging: The basic machine check, LSB subpacket and LMA subpacket will be provided. Set software flag bits 96 (LSB subpacket), 97 (LMA subpacket), 32 (DECchip 21064 writeblock), 49 (LSB ERR, write), and 66 (NXM-mem write).

Additional Parsing: Memory address correlation.

LSB Command Parity Errors

Description: An LSB command parity error was detected on an LSB bus write cycle in which this CPU was the commander. This results in CACK HERR being sent to the DECchip 21064, which causes a machine check through 670.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and an LSB subpacket. Set software flags 96 (LSB), 60 (CPE), 49 (LSB ERR, write), and 32 (DECchip 21064 writeblock).

LSB Control Transmit Check Error

Description: This module detected that an LSB control line(s) it was driving did not match with what it had seen on the LSB bus. This is a fatal condition. Cache coherence could be lost.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check and an LSB snapshot is required. Set software flag bits 96 (LSB), 49 (LSB ERR, write), 32 (DECchip 21064 writeblock), and 61 (CTCE).

Inconsistent—LSB

Description: Attempting to parse the possible error conditions for a machine check while the DECchip 21064 has an outstanding writeblock, it was found that no error cases were present which would have caused the system to machine check.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check. Set software flag bits 32 (DECchip 21064 writeblock), 49 (LSB ERR, write), and 70 (inconsistent error).

LSB Cache Protocol Error

Description: During an outstanding CPU writeblock request, the LEVI, when attempting to do a write CSR, detected an ilbranchal assertion of either Shared or Dirty by another node. This is considered fatal to the system. Cache state is probably corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check along with an LSB subpacket will be sufficient. Set software flag bits 96 (LSB subpacket), 50 (LSB ERR, write CSR), 32 (writeblock), and 51/52 (Shared/Dirty error) based on the bit(s) set.

LSB Synchronization Failure

Description: During an outstanding CPU writeblock request, the LEVI detected either a stall error, confirmation error, or command/address error while the LEVI was attempting to do an LSB write CSR. These errors imply loss of LSB synchronization. They probably signify that some other internal node errors have occurred elsewhere in the system. These are considered fatal errors.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check with an LSB snapshot will be fine. Set software flag bits 96 (LSB subpacket), 32 (writeblock), 50 (LSB ERR, write CSR), and 53/54/55 (stall, confirmation, command/address) if appropriate.

LSB Command Parity Errors

Description: An LSB command parity error was detected on an LSB bus write CSR cycle in which this CPU was the commander. This results in a CACK HERR being sent to the DECchip 21064, which causes a machine check through 670.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and an LSB subpacket. Set software flags 96 (LSB), 60 (CPE), 50 (LSB ERR, write CSR), and 32 (DECchip 21064 writeblock).

Write CSR Data Parity Error

Description: During a CSR write to an LSB based I/O register, the data cycle contained a parity error. This error can only occur when writing CSR data to another LSB node. If the CPU was writing CSR data to itself, it would have used a backdoor method which would not use the LSB bus.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, log the basic machine check, an LSB snapshot and either an LMA subpacket or log adapter subpacket depending on where the I/O address pointed to. If it was to another CPU, the LSB snapshot will be sufficient. Set software flags 96 (LSB), 63 (CDPE), 32 (DECchip 21064 writeblock), and 50 (LSB write CSR).

CSR Write LSB Nonexistent Address

Description: During an outstanding DECchip 21064 writeblock command, the LEVI detected that the LSB CSR write request did not get a confirmation. This condition results in an NXM. The command on the LSB was a CSR write so this was actually going to an LSB I/O address. Use the latched address in the LBECSR register to determine if this is a valid I/O address. If the address is valid, it would appear that the associated node at that I/O address failed to respond. If the address is not valid, this would lean toward either a software problem or a hardware address generation problem. If the address belongs to a known node, log the appropriate additional error log subpacket (that is, if a memory, log an LMA, if IOP, log a log adapter, for a CPU, the LSB subpacket will do just fine).

Recovery procedure: None.

Restart condition: None.

Error logging: The basic machine check, LSB subpacket and either a log adapter or an LMA subpacket will be provided based on what type of node the latched I/O address pointed to. Set software flag bits 96 (LSB subpacket), 32 (DECchip 21064 writeblock), 50 (LSB ERR, write CSR), and 97 (LMA subpacket) or 98 (log adapter) based on type of node.

Inconsistent—LSB

Description: In an attempt to parse the possible error conditions for a machine check while the DECchip 21064 has an outstanding writeblock to an

LSB I/O address, it was found that no error cases were present which would have caused the system to machine check.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check. Set software flag bit 32 (DECchip 21064 writeblock), 50 (LSB ERR, write CSR), and 70 (inconsistent error).

Previous System Error Latched

Description: When parsing the reasons for a machine check while the CPU had an outstanding writeblock request, it was found that the latched LSB bus state did not correspond to the CPU detecting the machine check. It is assumed that a previous LSB error has latched all the bus registers. Multiple errors must have occurred.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and an LSB snapshot. Set software flag bits 96 (LSB), 32 (DECchip 21064 writeblock), and 69 (previous system error).

Read Arbitration Drop

Description: An LSB arbitration drop was detected by the LEVI while the LEVI was attempting to get read data to do a B-cache fill to satisfy a loadlock request from the CPU. Arbitration drop is considered fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 33 (DECchip 21064 loadlock), 40 (NSES), and 41 (arbdrop).

Read Arbitration Collision

Description: An LSB arbitration collision was detected by the LEVI while the LEVI was attempting to get read data to do a B-cache fill to satisfy a loadlock request from the CPU. Arbitration collision is considered fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 33 (DECchip 21064 loadlock), 40 (NSES) and 42 (arbcol).

LEVI B-Cache Tag Parity Error on Lookup

Description: During a CPU-requested loadlock command, the LEVI attempted to do a B-cache lookup to determine if the requested data resided in the B-cache. The LEVI detected a parity error while checking the TAG address information in the RAMs. This is fatal. B-cache state could be corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check will do. All error information required to do diagnosis resides on this CPU. Set software flag bits 33 (DECchip 21064 loadlock) and 43 (BTAGPE).

LEVI B-Cache Status Parity Error on Lookup

Description: During a CPU-requested loadlock command, the LEVI attempted to do a B-cache lookup to determine if the requested data resided in the B-cache. The LEVI detected a parity error while checking the status information in the RAMs. This error is fatal. B-cache state could be corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check will do. All error information required to do diagnosis resides on this CPU. Set software flag bits 33 (DECchip 21064 loadlock) and 44 (BSTATPE).

Inconsistent Error—NSES

Description: During the LEVI sourcing data from the LSB to satisfy a DECchip 21064 loadlock command, the NSES bit was set indicating detection of an internal KN7AA error. However, there were no supporting error bits set to indicate this condition. This is an inconsistent error condition.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 32 (DECchip 21064 loadlock), 40 (NSES), and 45 (inconsistent).

LSB Cache Protocol Error

Description: During an outstanding CPU loadlock request, the LEVI, when attempting to read LSB data, detected an assertion of either Shared or Dirty by another node. This is considered fatal to the system. Cache state is probably corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check along with an LSB subpacket will be sufficient. Set software flag bits 96 (LSB subpacket), 48 (LSB ERR, read), 33 (loadlock), and 51/52 (Shared/Dirty error) if appropriate.

LSB Synchronization Failure

Description: During an outstanding CPU loadlock request, the LEVI detected either a stall error, confirmation error, or command/address error while the LEVI was attempting to do an LSB read. These errors imply loss of LSB synchronization. They probably signify that some other internal node errors have occurred elsewhere in the system. These are considered fatal errors.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check with an LSB snapshot will be fine. Set software flag bits 96 (LSB subpacket), 33 (DECchip 21064 loadlock), 48 (LSB ERR, read), and 53/54/55 (stall, confirmation, command/address) if appropriate.

Read LSB Nonexistent Memory

Description: During an outstanding DECchip 21064 loadlock command, the LEVI detected that the LSB read request did not get a confirmation. This condition results in an NXM. The command on the LSB was a read, so this was going to an LSB MEM address. Use the latched address in the LBECSR register to determine if this is a valid address. If the address is valid, it would appear that a memory is at fault. If the address is not valid, this would lean toward either a software problem or a hardware address generation problem. If the address is within memory range, include the memory registers for the associated memory controller.

Recovery procedure: None.

Restart condition: If expected, continue, else crash the system.

Error logging: The basic machine check, LSB subpacket and LMA subpacket will be provided. Set software flag bits 96 (LSB subpacket), 97 (LMA subpacket), 33 (DECchip 21064 loadlock), 48 (LSB ERR, read), and 57 (NXM-mem read).

Additional parsing: Memory address correlation.

LSB Command Parity Errors

Description: An LSB command parity error was detected on a bus cycle in which this CPU was the commander. This results in CACK HERR being sent to the DECchip 21064, which causes a machine check through 670.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and an LSB subpacket. Set software flags 96 (LSB), 60 (CPE), 48 (LSB ERR, read), and 33 (DECchip 21064 loadlock).

LSB Control Transmit Check Error

Description: This module detected that an LSB control line(s) it was driving did not match with what it had seen on the LSB bus. This is a fatal condition. Cache coherence could be lost.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check and an LSB snapshot is required. Set software flag bits 96 (LSB), 48 (LSB ERR, read), 33 (DECchip 21064 loadlock), and 61 (CTCE).

Inconsistent—LSB

Description: In an attempt to parse the possible error conditions for a machine check while the DECchip 21064 has an outstanding loadlock, it was found that no error cases were present which would have caused the system to machine check.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check. Set software flag bits 33 (DECchip 21064 loadlock), 48 (LSB ERR, read), and 70 (inconsistent error).

Previous System Error Latched

Description: When parsing the reasons for a machine check while the CPU had an outstanding loadlock request, it was found that the latched LSB bus state did not correspond to the CPU detecting the machine check. It is assumed that a previous LSB error has latched all the bus registers. Multiple errors must have occurred.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and an LSB snapshot. Set software flag bits 96 (LSB), 33 (DECchip 21064 loadlock), and 69 (previous system error).

Read Arbitration Drop

Description: An LSB arbitration drop was detected by the LEVI while the LEVI was attempting to arbitrate for the LSB to do an LSB write to satisfy a store conditional request from the CPU. Arbitration drop is considered fatal. Note that this write was going to the LSB bus because the B-cache state for the requested data was set at Shared.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 34 (DECchip 21064 StoreCond), 40 (NSES), and 41 (arbdrop).

Read Arbitration Collision

Description: An LSB arbitration collision was detected by the LEVI while the LEVI was attempting to satisfy a loadlock request from the CPU. Arbitration collision is considered fatal. Note that the LEVI was attempting to do an LSB write because the B-cache state of the requested data was Shared.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 34 (DECchip 21064 StoreCond), 40 (NSES), and 42 (arbcol).

LEVI B-Cache Tag Parity Error on Lookup

Description: During a CPU-requested StoreCond command, the LEVI attempted to do a B-cache lookup to determine if the requested data resided in the B-cache and the cache state it was in. The LEVI detected a parity error while checking the tag address information in the RAMs. This is fatal. B-cache state could be corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check will do. All error information required to do diagnosis resides on this CPU. Set software flag bits 34 (DECchip 21064 StoreCond), and 43 (BTAGPE).

LEVI B-Cache Status Parity Error on Lookup

Description: During a CPU-requested StoreCond command, the LEVI attempted to do a B-cache lookup to determine if the requested data resided in the B-cache and also the cache state it was in. The LEVI detected a parity error while checking the status information in the RAMs. This is fatal. B-cache state could be corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check will do. All error information required to do diagnosis resides on this CPU. Set software flag bits 34 (DECchip 21064 StoreCond) and 44 (BSTATPE).

Inconsistent Error—NSES

Description: During the LEVI sourcing data to satisfy a DECchip 21064 StoreCond command, the NSES bit was set indicating that an internal KN7AA error was detected. However, there were no supporting error bits set to indicate this condition. This is an inconsistent error condition.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check along with an LSB snapshot will do. Set software flag bits 96 (LSB subpacket), 34 (DECchip 21064 StoreCond), 40 (NSES), and 45 (inconsistent).

LSB Cache Protocol Error

Description: During an outstanding CPU StoreCond request, the LEVI, when attempting to write LSB data, detected an assertion of either Shared or Dirty by another node. This is considered fatal to the system. Cache state is probably corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check along with an LSB subpacket will be sufficient. Set software flag bits 96 (LSB subpacket), 49

(LSB ERR write), 34 (StoreCond), and 51/52 (Shared/Dirty error) based on the bit(s) set.

LSB Synchronization Failure

Description: During an outstanding DECchip 21064 CPU StoreCond request, the LEVI detected either a stall error, confirmation error, or command/address error while the LEVI was attempting to do an LSB write. These errors imply loss of LSB synchronization. They probably signify that some other internal node errors have occurred elsewhere in the system. These are considered fatal errors.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error, a basic machine check with an LSB snapshot will be fine. Set software flag bits 96 (LSB subpacket), 34 (StoreCond), 49 (LSB ERR, write), and 53/54/55 (stall, confirmation, command/address) if appropriate.

Write LSB Nonexistent Memory

Description: During an outstanding DECchip 21064 StoreCond command, the LEVI detected that the LSB write request did not get a confirmation. This results in an NXM. The command on the LSB was a write, so this was going to an LSB MEM address. Use the latched address in the LBECSR register to determine if this is a valid address. If the address is valid, it would appear that a memory is at fault. If the address is not valid, this would lean toward either a software problem or a hardware address generation problem. If the address is within memory range, include the memory registers for the associated memory controller. Note that this DECchip 21064 store conditional request produced an LSB write because the cache state was Shared.

Recovery procedure: None.

Restart condition: None.

Error logging: The basic machine check, LSB subpacket and LMA subpacket will be provided. Set software flag bits 96 (LSB subpacket), 97 (LMA subpacket), 34 (DECchip 21064 StoreCond), 49 (LSB ERR, write), and 66 (NXM-mem write).

Additional parsing: Memory address correlation.

LSB Command Parity Errors

Description: An LSB command parity error was detected on an LSB bus write cycle in which this CPU was the commander. This results in CACK HERR being sent to the DECchip 21064, which causes a machine check through 670. This LSB request resulted from a DECchip 21064 store conditional command in which the LEVI determined the B-cache state was set to Shared. This is fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and an LSB subpacket. Set software flags 96 (LSB), 60 (CPE), 49 (LSB ERR, write), and 34 (StoreCond).

LSB Control Transmit Check Error

Description: This module detected that an LSB control line(s) it was driving did not match with what it had seen on the LSB bus. This is a fatal condition. Cache coherence could be lost.

Recovery procedure: None.

Restart condition: None.

Error logging: A basic machine check and an LSB snapshot is required. Set software flag bits 96 (LSB), 49 (LSB ERR, write), 34 (DECchip 21064 StoreCond), and 61 (CTCE).

Inconsistent—LSB

Description: In an attempt to parse the possible error conditions for a machine check while the DECchip 21064 has an outstanding StoreCond, it was found that no error cases were present which would have caused the system to machine check.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check. Set software flag bit 34 (DECchip 21064 StoreCond), 49 (LSB ERR, write), and 70 (inconsistent error).

Previous System Error Latched

Description: When parsing the reasons for a machine check while the CPU had an outstanding StoreCond request, it was found that the latched LSB bus state did not correspond to the CPU detecting the machine check. It is assumed that a previous LSB error has latched all the bus registers. Multiple errors must have occurred.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and an LSB snapshot. Set software flag bits 96 (LSB), 34 (DECchip 21064 StoreCond), and 69 (previous system error).

Failure Not Understoud

Description: After parsing all possible known reasons for a machine check, it was found that there were no matches. This could be due to unexpected system error behavior or error cases that were not understood when the parse trees were developed. In any case, this is considered inconsistent and will be a fatal condition.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic machine check and also an LSB subpacket. Set software flag 96 (LSB) and 70 (inconsistent).

14.3.4 Events Reported Through 660 Machine Checks

The KN7AA module detects error conditions related to the operation of the B-cache (LSB side), the Gbus, and the LSB. These errors, when occurring asynchronous to the operation of the DECchip 21064 processor, are reported through the irq_4 signal line on the KN7AA module. This, in turn, becomes an IPL31 interrupt and is vectored through offset 660. Single-bit ECC errors that are caused by either another CPU's B-cache or the LSB also cause a 660 to occur.

The KN7AA also vectors through 660 if other nodes pull the LSB ERR line. This is the case for the IOP. Error handling in the 660 domain includes monitoring and error checking for IOP-detected LSB errors. See the *I/O System Technical Manual* for discussion IOP error parse trees.

Figure 14-6 is the parse tree associated with the 660 machine check. Following the parse tree is a description of each type of error and, when possible, a suggested recovery method.

Figure 14-6 System Machine Check 660 Parse Tree

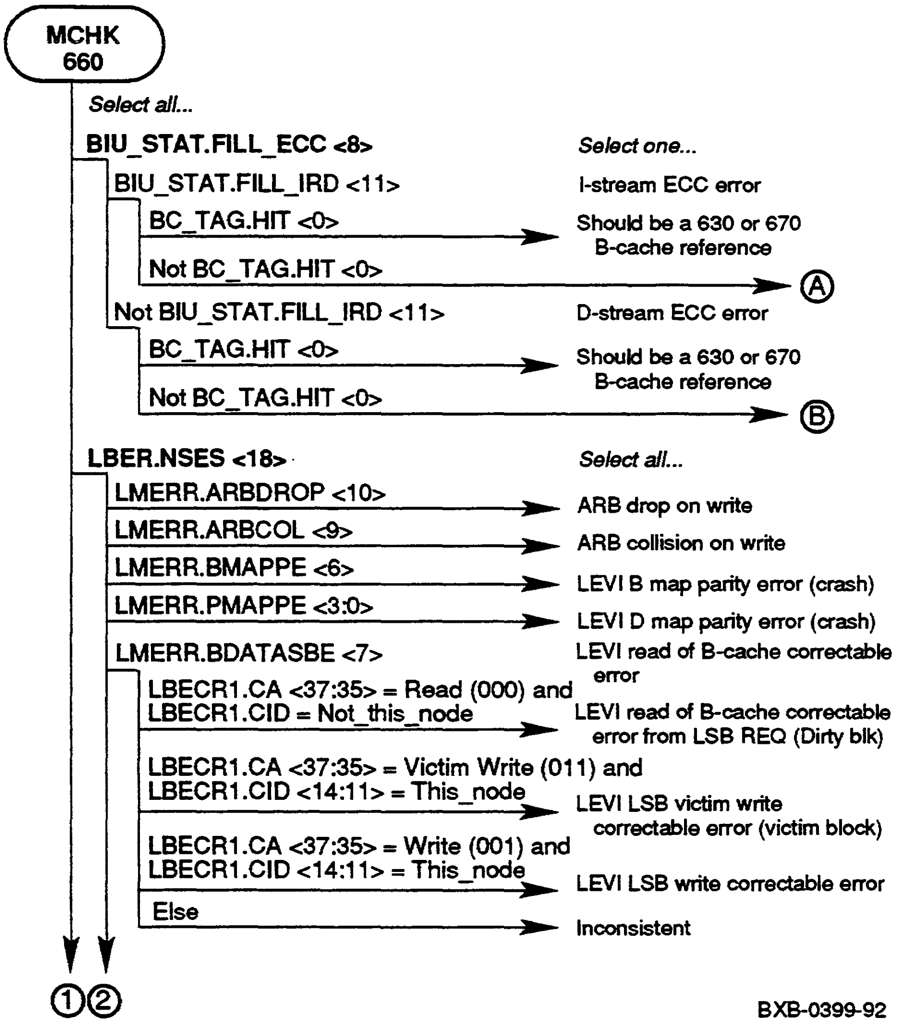

MCHK 660
Select all...
BIU_STAT.FILL_ECC <8>
Select one...
BIU_STAT.FILL_IRD <11>
I-stream ECC error
BC_TAG.HIT <0>
Should be a 630 or 670 B-cache reference
Not BC_TAG.HIT <0>
A
Not BIU_STAT.FILL_IRD <11>
D-stream ECC error
BC_TAG.HIT <0>
Should be a 630 or 670 B-cache reference
Not BC_TAG.HIT <0>
B
LBER.NSES <18>
Select all...
LMERR.ARBDROP <10>
ARB drop on write
LMERR.ARBCOL <9>
ARB collision on write
LMERR.BMAPPE <6>
LEVI B map parity error (crash)
LMERR.PMAPPE <3:0>
LEVI D map parity error (crash)
LMERR.BDATASBE <7>
LEVI read of B-cache correctable error
LBECR1.CA <37:35> = Read (000) and LBECR1.CID = Not_this_node
LEVI read of B-cache correctable error from LSB REQ (Dirty blk)
LBECR1.CA <37:35> = Victim Write (011) and LBECR1.CID <14:11> = This_node
LEVI LSB victim write correctable error (victim block)
LBECR1.CA <37:35> = Write (001) and LBECR1.CID <14:11> = This_node
LEVI LSB write correctable error
Else
Inconsistent
1
2
BXB-0399-92

Figure 14-6 System Machine Check 660 Parse Tree (Continued)

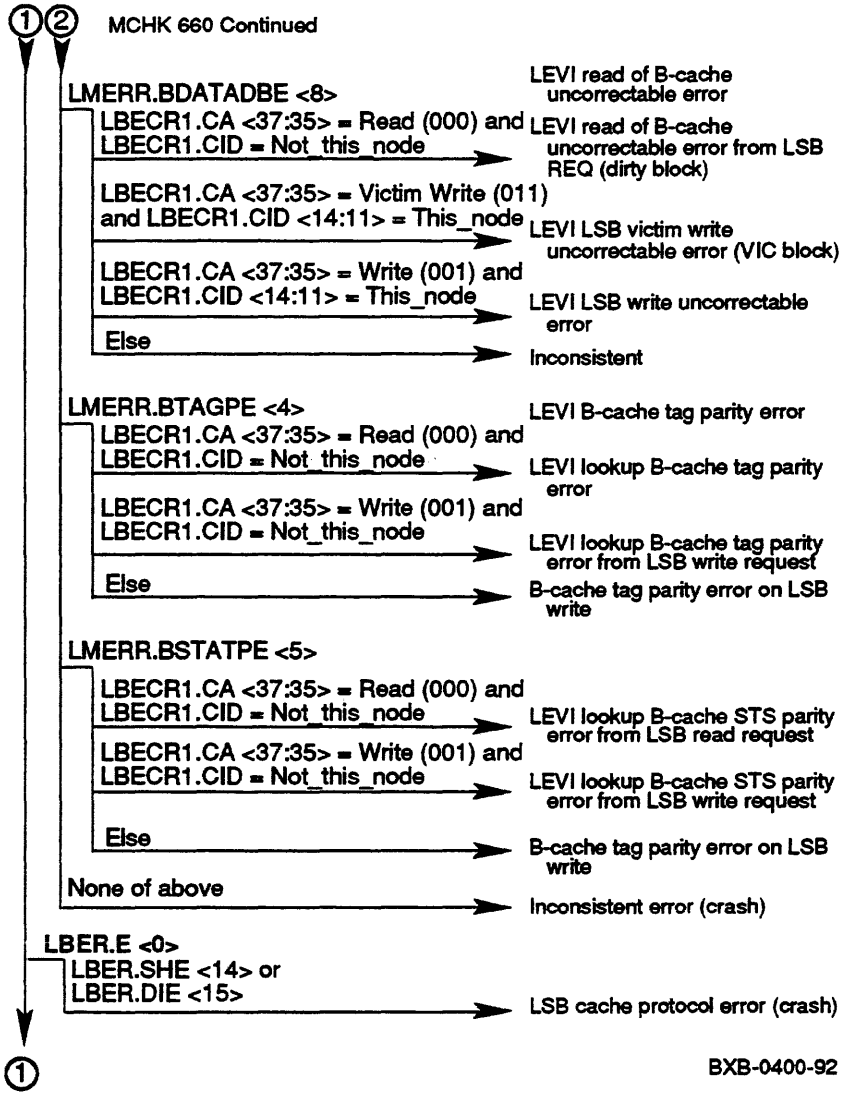

①②
MCHK 660 Continued
LMERR.BDATADBE <8>
LEVI read of B-cache uncorrectable error
LBECR1.CA <37:35> = Read (000) and LBECR1.CID = Not_this_node
LEVI read of B-cache uncorrectable error from LSB REQ (dirty block)
LBECR1.CA <37:35> = Victim Write (011) and LBECR1.CID <14:11> = This_node
LEVI LSB victim write uncorrectable error (VIC block)
LBECR1.CA <37:35> = Write (001) and LBECR1.CID <14:11> = This_node
LEVI LSB write uncorrectable error
Else
Inconsistent
LMERR.BTAGPE <4>
LEVI B-cache tag parity error
LBECR1.CA <37:35> = Read (000) and LBECR1.CID = Not_this_node
LEVI lookup B-cache tag parity error
LBECR1.CA <37:35> = Write (001) and LBECR1.CID = Not_this_node
LEVI lookup B-cache tag parity error from LSB write request
Else
B-cache tag parity error on LSB write
LMERR.BSTATPE <5>
LBECR1.CA <37:35> = Read (000) and LBECR1.CID = Not_this_node
LEVI lookup B-cache STS parity error from LSB read request
LBECR1.CA <37:35> = Write (001) and LBECR1.CID = Not_this_node
LEVI lookup B-cache STS parity error from LSB write request
Else
B-cache tag parity error on LSB write
None of above
Inconsistent error (crash)
LBER.E <0>
LBER.SHE <14> or LBER.DIE <15>
LSB cache protocol error (crash)
①
BXB-0400-92

Figure 14-6 System Machine Check 660 Parse Tree (Continued)

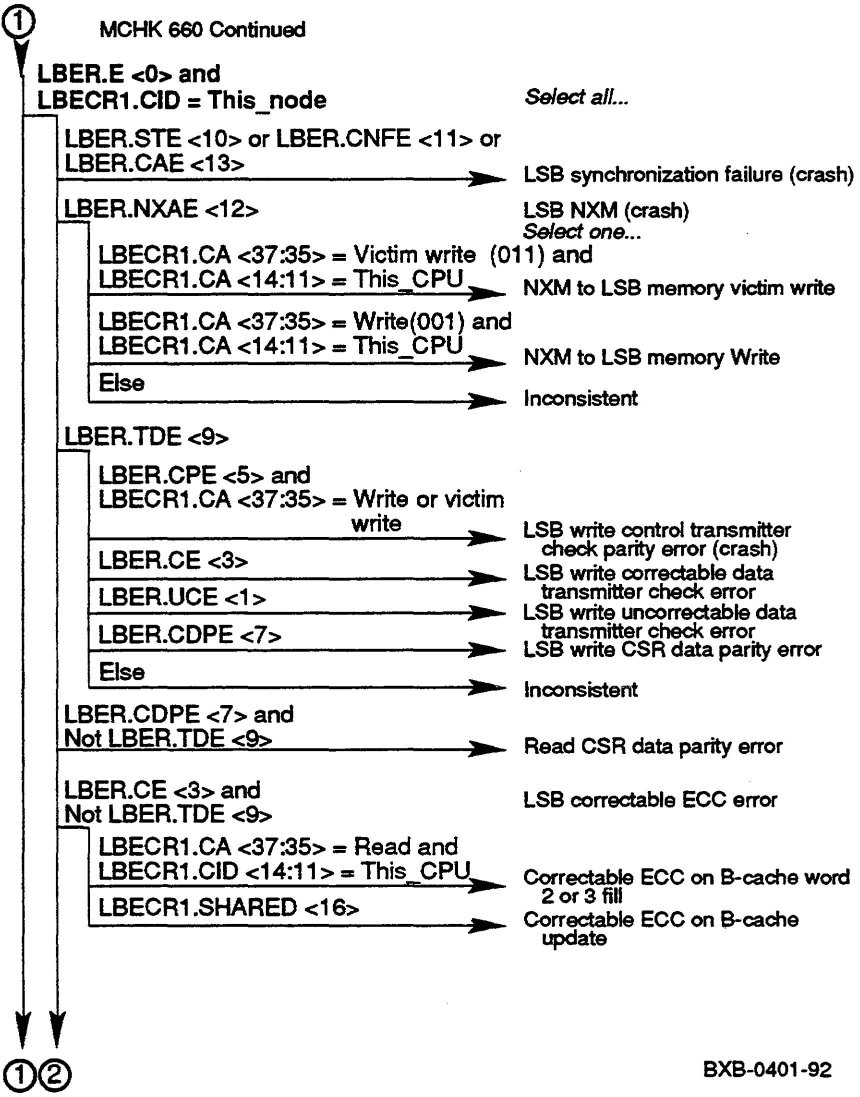

1
MCHK 660 Continued
LBER.E <0> and
LBECR1.CID = This_node
Select all...
LBER.STE <10> or LBER.CNFE <11> or
LBER.CAE <13>
LSB synchronization failure (crash)
LBER.NXAE <12>
LSB NXM (crash)
Select one...
LBECR1.CA <37:35> = Victim write (011) and
LBECR1.CA <14:11> = This_CPU
NXM to LSB memory victim write
LBECR1.CA <37:35> = Write(001) and
LBECR1.CA <14:11> = This_CPU
NXM to LSB memory Write
Else
Inconsistent
LBER.TDE <9>
LBER.CPE <5> and
LBECR1.CA <37:35> = Write or victim
write
LSB write control transmitter
check parity error (crash)
LBER.CE <3>
LSB write correctable data
transmitter check error
LBER.UCE <1>
LSB write uncorrectable data
transmitter check error
LBER.CDPE <7>
LSB write CSR data parity error
Else
Inconsistent
LBER.CDPE <7> and
Not LBER.TDE <9>
Read CSR data parity error
LBER.CE <3> and
Not LBER.TDE <9>
LSB correctable ECC error
LBECR1.CA <37:35> = Read and
LBECR1.CID <14:11> = This_CPU
Correctable ECC on B-cache word
2 or 3 fill
LBECR1.SHARED <16>
Correctable ECC on B-cache
update
1
2
BXB-0401-92

Figure 14-6 System Machine Check 660 Parse Tree (Continued)

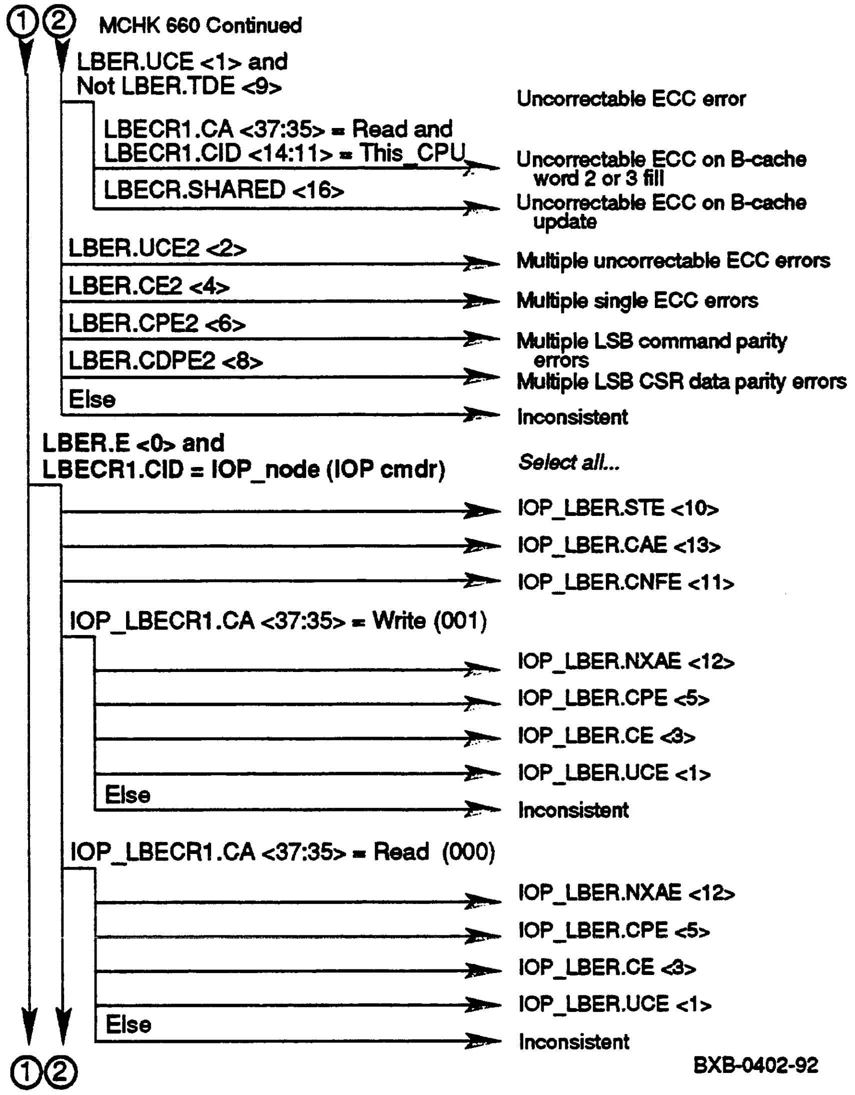
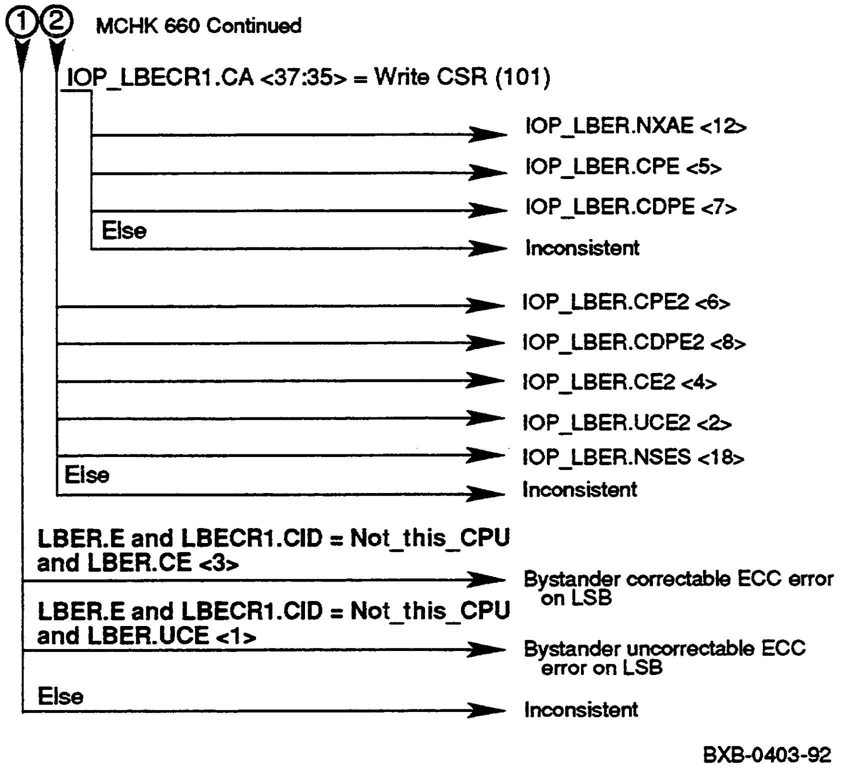
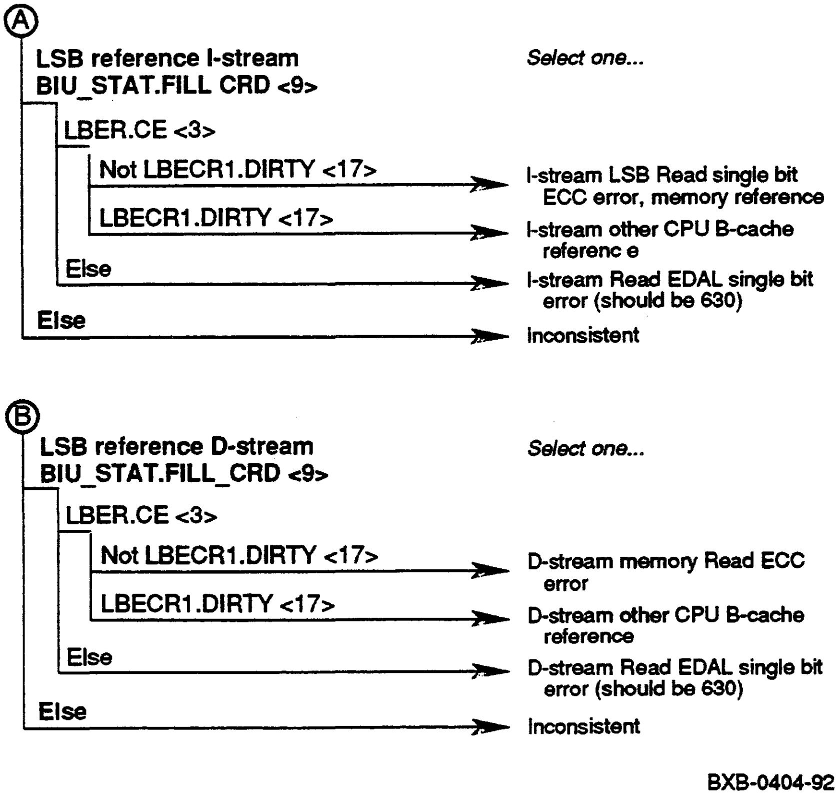


Figure 14-6 System Machine Check 660 Parse Tree (Continued)



1
2
MCHK 660 Continued
IOP_LBECR1.CA <37:35> = Write CSR (101)
IOP_LBER.NXAE <12>
IOP_LBER.CPE <5>
IOP_LBER.CDPE <7>
Else
Inconsistent
IOP_LBER.CPE2 <6>
IOP_LBER.CDPE2 <8>
IOP_LBER.CE2 <4>
IOP_LBER.UCE2 <2>
IOP_LBER.NSES <18>
Else
Inconsistent
LBER.E and LBECR1.CID = Not_this_CPU and LBER.CE <3>
Bystander correctable ECC error on LSB
LBER.E and LBECR1.CID = Not_this_CPU and LBER.UCE <1>
Bystander uncorrectable ECC error on LSB
Else
Inconsistent
BXB-0403-92

Figure 14-6 System Machine Check 660 Parse Tree (Continued)



I-Stream Inconsistent Error

Description: During an I-stream reference the error bit FILL_CRD was not set, indicating that this error was not a single-bit error. This is an inconsistent state for the 660 parse flow.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 67.

LSB Single-Bit ECC Error, Memory Reference

Description: During an I-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. From parsing the error, it was found that the bus cycle associated with this error had a CE error. Also, the dirty bit in LBECR1 was clear, which implies that a memory supplied the data. Operating system software should determine which memory controller is associated with the latched LSB address and append a memory controller subpacket from the associated memory.

Recovery procedure: Hardware and PALcode do recovery.

Restart condition: Restart if corrected by PALcode or hardware.

Error logging: For this error, the basic machine check entry with an LSB subpacket and an LMA subpacket will be required. Set software flag bits 68, 96 (LSB subpacket), and 97 (LMA subpacket).

Additional parsing: Memory address correlation.

I-Stream Read Other CPU B-Cache Single-Bit ECC Error

Description: During an I-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. From parsing the error, it was found that it was caused by a B-cache single-bit error which was sourced from another CPU node. The other CPU(s) will also be attempting to parse this error through its 660 error handler.

Recovery procedure: Hardware and PALcode do recovery.

Restart condition: Restart if corrected by PALcode or hardware.

Error logging: For this error the basic machine check entry with an LSB subpacket will be required. Set software flag bits 69 and 96 (LSB subpacket).

Additional parsing: BDATASBE parse.

I-Stream Read EDAL Single-Bit ECC Error

Description: During an I-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. This error was caused by the EDAL data path. Note that there were no LSB errors on this data transfer. This error should be a 630!

Recovery procedure: Hardware and PALcode do recovery.

Restart condition: Restart if corrected by PALcode or hardware.

Error logging: None. Should be a 630.

D-Stream Inconsistent Error

Description: During a D-stream reference the error bit FILL_CRD was not set, indicating that this was not a single-bit error. This is an inconsistent state for the 660 parse flow.

Recovery procedure: None.

Restart condition: Terminate the user or session.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 77.

D-Stream LSB Read Single-Bit ECC Error, Memory Reference

Description: During a D-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. From parsing the error, it was found that the bus cycle associated with this error had a CE error. Also, the dirty bit in LBECR1 was clear, which implies that a memory supplied the data. Operating system software should figure out which memory controller is associated with the latched LSB address and append a memory controller subpacket from the associated memory. This error was caused by an LSB single-bit error in which a memory was the source.

Recovery procedure: Hardware and PALcode do recovery.

Restart condition: Restart if corrected by PALcode or hardware.

Error logging: For this error, the basic machine check entry with an LMA subpacket (the one associated with the error) will be required. Set software flag bits 78 and 97 (LMA subpacket).

Additional parsing: Memory address correlation.

D-Stream Read Other CPU B-Cache Single-Bit ECC Error

Description: During a D-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. Note that this error was caused by another CPU's B-cache. The other CPU will also be attempting to parse this error through its 660 error handler.

Recovery procedure: Hardware and PALcode do recovery.

Restart condition: Restart if corrected by PALcode or hardware.

Error logging: For this error, the basic machine check entry with an LSB subpacket will be required. Set software flag bits 79 and 96 (LSB subpacket).

Additional parsing: BDATASBE parse.

D-Stream Read EDAL Single-Bit Error

Description: During a D-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. Note that this error was caused by the EDAL data path. Note that this should be a 630!

Recovery procedure: Hardware and PALcode do recovery.

Restart condition: Restart if corrected by PALcode or hardware.

Error logging: None.

Arbitration Drop or Arbitration Collision

Description: A serious LSB failure has occurred in which one or several nodes have asserted acknowledgment of status signals out of sequence. This implies a possible loss of cache coherence.

Recovery procedure: No specific recovery action is called for.

Restart condition: This error cannot be recovered from. Cache status is potentially corrupt. The current operating system session should be terminated.

Error logging: Log the basic 660 error log and an LSB subpacket. Set software flag 96 (LSB subpacket) and 1 (arbdrop) or 2 (arbcol), depending on the error type.

B-Map Parity Error

Description: The LEVI detected a parity error when accessing the B-map RAMs. This is considered fatal. Cache state could be corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660 error log. All error data required is resident on the CPU reporting the error. Set software flag 3.

P-Map Parity Error

Description: The LEVI detected a P-map parity error when looking up the CPU cache status. This is considered fatal. Because the lookup was not successful, a required invalidate might not have occurred, making the CPU P-cache retain stale data. Cache coherence is at risk.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660 error log. All error data required is resident on the CPU reporting the error. Set software flag 4.

LEVI Read of B-Cache Correctable Error

Description: During an LSB transaction, it was determined by this LEVI that this B-cache contained the latest copy of the data (dirty). This CPU's LEVI read the cache and supplied the data to the LSB. This data contained a single-bit error.

Recovery procedure: For this error, log the fact that this board detected that its cache supplied correctable data. Clear the error state.

Restart condition: After recording the error and clearing error state, return to the calling routine. This error is corrected by the consumer of the data. It is not fatal and is recoverable.

Error logging: Log a basic 660. Set software flag bit 5. Note that if the consumer of the data was another CPU, that CPU will be going through its machine check code thread. In that case, don't log the additional 660. Just flag the error in the software flag bits so EEPROM logging will eventually take place. If the consumer of the data was the IOP, log the 660 and an LSB snapshot with a log adapter subpacket. In the latter case, make sure software flag bits 96 (LSB subpacket), 98 (log adapter), and 5 (BDATASBE) are set.

LEVI Read of B-Cache Uncorrectable Error

Description: During an LSB transaction, it was determined by this LEVI that this B-cache contained the latest copy of the data (dirty). This CPU's LEVI read the cache and supplied the data to the LSB. This data contained a double-bit error. Note that there are three system states associated with this error condition. This text applies to all of those.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660. Set software flag bit 6. Note that this is fatal to the system. The consumer of the data, if it was a CPU, will have gone through the machine check code flow. In that case, don't log this 660, because machine check will have already logged the error state. Just set the software flag bit so EEPROM logging can take place. If the consumer was the IOP, log the 660 and an LSB snapshot with a log adapter subpacket. In the latter case, make sure software flag bits 96 (LSB subpacket), 98 (log adapter), and 6 (BDATADBE) are set.

LEVI B-Cache Tag Parity Error

Description: When attempting to do a B-cache lookup, the LEVI detected a parity error in the B-cache tag RAMs. Because the lookup failed, the cache state could potentially be corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660. All required error state is on this CPU. Set software flag bit 7 (BTAGPE).

LEVI B-Cache Status Parity Error

Description: When attempting to do a B-cache lookup, the LEVI detected a parity error in the B-cache status RAMs. Because the lookup failed, the cache state could be corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660. All required error state is on this CPU. Set software flag bit 8 (BSTATPE).

Inconsistent Error

Description: During analysis of the error, the NSES bit was set but no supporting error state could be found. This is inconsistent.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660 and an LSB snapshot. Set software flag bits 96 (LSB) and 9 (inconsistent NSES).

LSB Cache Protocol Error

Description: The LEVI on this CPU detected that the cache state line(s) on the LSB were NOT set on the appropriate cycle on the LSB. This error implies that cache state is corrupt.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a 660 error log with an LSB snapshot. Set software flag bits 96 (LSB), 16 (LSB ERR), and 17 (SHE) or 18 (DIE), depending on the error detected.

LSB Synchronization Failure

Description: The LEVI detected the assertion of STALL, or CNF during the wrong cycle type. Also, the KN7AA could have detected a command/address error. All these errors are considered fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660 with an LSB subpacket. Set software flag bits 96 (LSB), 16 (LSB ERR), and one of 19 (STE), 20 (CNFE), or 21 (CAE), depending on the error type.

Additional parsing: Memory address correlation.

LSB Nonexistent Memory

Description: The LEVI, upon writing data out to memory, detected that the command/address cycle did not get a confirmation. This is an NXM error and sets NXAE. The write data is lost. This is fatal.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a 660 and an LSB subpacket. Also, if any memory errors exist, log the appropriate LMA subpacket. Set software flag bits 96 (LSB), 22 (NXAE), and 97 (LMA), if required.

Control Parity Error

Description: The LEVI, when outputting onto the LSB, detected that what it sent out for command information had a parity error. Command parity error bit was set indicating this error. If CTCE is also set, this implies that the data driven from the module to the bus changed. Something directly on the bus broke the data. If CTCE is not set, it can be assumed that this module has caused the failure.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a 660 and an LSB subpacket. Set software flag bits 96 (LSB), 16 (LSB ERR), and 23 (CPE).

LSB Correctable Error

Description: The LEVI chips detected a single-bit error in the data that this node was transmitting on the bus. The consumer of the data does correct this data error. Also, if DTCE is set, this implies that something on the bus broke the data. If DTCE is not set, this implies that this module drove the bad data onto the bus.

Recovery procedure: Log the event and clear the error bits.

Restart condition: Restart. The error is recoverable from this node's point of view.

Error logging: Log the 660 and the LSB subpacket. If another CPU was the consumer, it will have taken a machine check. If this is the case, do not log this 660, just record the error into the software flags. If a CPU was not the consumer, make sure you log this and set software flags 96 (LSB), 16 (LSB ERR), and 24 (CE). Also, if the IOP was the consumer, a log adapter packet should be provided. Set software flag bit 98 (log adapter).

Additional parsing: Memory address correlation.

LSB Uncorrectable Error

Description: The LEVI chips detected that when this node was transmitting on the bus the data had a double-bit error. The consumer of the data cannot correct this error. Also, if DTCE is set, this implies that something on the bus broke the data. If DTCE is not set, this implies that this module drove the bad data onto the bus.

Recovery procedure: None.

Restart condition: None.

Error logging: Log the 660 and an LSB subpacket. If another CPU was the consumer, it will have taken a machine check. If this is the case, do not log this 660, just record the error into the software flags. If a CPU was not the consumer, make sure you log this and set software flags 96 (LSB), 16 (LSB ERR), and 25 (UCE). Also, if the IOP was the consumer, a log adapter packet should be provided within this 660 error log.

Additional parsing: Memory address correlation.

Write CSR Data Parity Error

Description: During a write to an LSB I/O space address, a CSR data parity error was detected. This is considered fatal. Also, if DTCE is set, this implies that something on the bus broke the data. If DTCE is not set, this implies that this module drove the bad data onto the bus.

Recovery procedure: None.

Restart condition: None.

Error logging: Log the 660 and also an LSB subpacket. Set software flag bits 96 (LSB), 16 (LSB ERR), and 26 (CDPE).

Additional parsing: Memory address correlation.

Read CSR Data Parity Error

Description: During a read from an LSB I/O space address, a CSR data parity error was detected. This is considered fatal. Note that if TDE is also set, the read was probably to our own CSR I/O address space.

Recovery procedure: None.

Restart condition: None.

Error logging: Log the 660 and also an LSB subpacket. Set software flag bits 96 (LSB), 16 (LSB ERR), and 26 (CDPE).

Additional parsing: I/O address correlation.

B-Cache Fill 2/3 Correctable Error

Description: During a CPU read command, the first two (target) octawords were received without error and proceeded into the B-cache and CPU as fill data. The remaining two octawords off the bus had a single-bit ECC error. This error is triggered as a 660 because the first words were received by the CPU correctly without error. The LEVI signals this error via the IRQ4 line. In this case the B-cache has a single-bit error resident.

Recovery procedure: You can choose to do nothing and then expect to see a machine check if the CPU references this single-bit location (not the preferable course) or the B-cache block can be invalidated or flushed back to memory if it has become dirty. In either case, clear the error bits.

Restart condition: Continue, this error is recoverable.

Error logging: Log a basic 660 with an LSB snapshot. Set software flag bits 96 (LSB), 16 (LSB ERR), and 27 (B-cache fill 2-3 CE).

Additional parsing: Memory address correlation.

Bystander Correctable ECC

Description: For this error, this LEVI has detected the correctable ECC error on the LSB. This node was not a participant in this transaction; it was a bystander.

Recovery procedure: Dismiss this error; clear the error bits.

Restart condition: Return. This node was not involved.

Error logging: No error logging is required. An error log entry occurs from either a CPU that took a machine check or from the IOP flows within this

660 parse tree. Just set software flag 28 (bystander CE) for EEPROM logging information.

B-Cache Fill 2/3 Uncorrectable Error

Description: During a CPU read command, the first two (target) octawords were received without error and proceeded into the B-cache and CPU as fill data. The remaining two octawords off the bus had a double-bit ECC error. This error is triggered as a 660 because the first words were received by the CPU correctly without error. The LEVI signals this error via the IRQ4 line. In this case our B-cache has a double-bit error now resident.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a 660 and an LSB subpacket. If the data source was memory, log an LMA subpacket from the associated memory controller. Set software flags 96 (LSB), 97 (LMA), 16 (LSB ERR), and 29 (B-cache fill 2-3 UCE).

Additional parsing: Memory address correlation.

Bystander Uncorrectable ECC

Description: For this error, this LEVI has detected the uncorrectable ECC error on the LSB. This node was not a participant in this transaction; it was a bystander.

Recovery procedure: None. But dismiss this, because this node was not involved. Clear the error bits.

Restart condition: None.

Error logging: No logging needed. The hardware that initiated this transaction will log the error. If it was a CPU, we'll get a machine check. If it was the IOP, then the error will be found by another branch farther down the 660 parse tree. Just set software flag 30 (bystander UCE) for EEPROM logging reasons.

LSB Second Errors

Description: Second error occurrences indicate serious system errors. Except for CE2, all are considered fatal. For CE2, if the system has a rev 2 DECchip 21064, this is fatal. If the system has rev 3 DECchip 21064, all occurrences of CE are recovered by hardware. Note that when these occur, the error state from the subsequent occurrence(s) are lost. The source and destination of the CE errors cannot be determined.

Recovery procedure: For CE2, rev 2 DECchip 21064, none; rev 3 DECchip 21064, continue; all others, none.

Restart condition: For CE2, rev 2 DECchip 21064, none; rev 3 DECchip 21064, continue; all others, none.

Error logging: There is no specific error logging required. The parse tree branch associated with the first occurrence of the error will indicate the proper error logging for the type of error encountered. Just set software flag 32 (UCE2), 33 (CE2), 34 (CPE2), or 35 (CDPE2), as appropriate, to indicate the occurrence of the second error.

Inconsistent State

Description: When parsing for a 660 reason, no LSB reason was found. This is an inconsistent state.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a 660 with an LSB snapshot. Set software flag bits 96 (LSB), 16 (LSB ERR), and 31 (LBER_Inconsistent).

IOP Commander

Description: The LSB error indicates that the commander node is the IOP. The rest of the errors parsed after this point deal with the error state from the IOP's viewpoint. Up to this point in the parse tree, any branch that was true should have been associated with a CPU being a responder or a bystander to the error detected by the IOP. In the case of a CPU bystander, the IOP error should have had the memory as the responder. The IOP flows will look for memory errors and report accordingly.

Recovery procedure: None.

Restart condition: None.

Error logging: None. Error logging associated with the IOP LSB detected errors will be specified farther down the parse tree. Just set software flag bit 36 (IOP CMDR) to highlight the IOP as the commander.

IOP-Detected LSB Synchronization Errors

Description: During an LSB command in which the IOP was the commander, one of the LSB synchronization errors occurred. Just as with the CPU, these errors are considered fatal. Stall error, command/address error, and confirmation error are the candidates for this class of error.

Recovery procedure: None.

Restart condition: None.

Error logging: For this error log a basic 660 with an LSB subpacket and a log adapter subpacket. Set software flag bits 96 (LSB subpacket), 98 (log adapter), 36 (IOP CMDR), and one of 40 (IOP Stall Err), 41 (IOP CAE), or 42 (IOP CNFE), depending on the type of LSB synchronization error.

IOP LSB Nonexistent Memory

Description: During an LSB command in which the IOP was the commander, the command/address cycle sent out did not get a confirmation. This results in an LSB NXM. When this occurs, one of following three cycles could be present on the LSB bus. First, a write cycle. For this the IOP was attempting to write to LSB memory. The latched LBECR0 will identify what address the write was trying to access. Second, the IOP could be reading from memory. In this case the latched LBECR0 identifies which address was being read from. Remember that the memory is the node that always does the confirmation to the command/address cycle, even if the data will be returned by some dirty B-cache. Third, the cycle could be a CSR write. This would be true if the IOP is writing to the Interrupt register. All cases are considered fatal. Finally, if the error shows up with some other command type, this error state is considered inconsistent. Note that this error could also be caused by some I/O bus adapter supplying an out of range memory address. If this is the case, suspect the XMI controller first, the DWLMA second. If the address is within memory space, then suspect the IOP and/or corresponding LSB memory. Also, if the bus type is an LSB read, the XMI I/O controller that initiated this transaction will be returned an RER error from the DWLMA. The operating system must do an XMI snapshot so the controller that initiated this transfer can be identified. For LSB writes, the information regarding the initiating adapter is lost. No XMI subpacket is required if the LSB bus transaction is a write. For systems with multiple hoses (XMIs), software will need to find the node on which the bus has the XBER<RER> bit set and supply an XMI snapshot from that XMI bus.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a 660 error log with an LSB subpacket. Set software flag bits 96 (LSB), 36 (IOP CMDR), 48 (IOP NXAE), and one of 45 (IOP LSB Write), 46 (IOP LSB Read), or 47 (IOP LSB Write CSR), depending on the type of command. If the cycle type is inconsistent, set software flag 49 (IOP NXAE Inconsistent). If the LSB cycle type is a read, log an XMI subpacket and set software flag 112 (XMI subpacket).

Additional parsing: Memory address correlation; XMI RER parsing to find the XMI RER only when cycle type is a read.

IOP LSB Command Parity Error

Description: During an LSB command in which the IOP was the commander, the command/address cycle that was sent out was determined to have bad parity. This results in LSB CPE being set. When this occurs, with the IOP being the commander, one of following three cycles could be present on the LSB bus. First, a write cycle. For this the IOP was attempting to write to LSB memory. Second, the IOP could be reading from memory. And third, the cycle could be a CSR write. This would be true if the IOP is writing to the Interrupt register. All cases are considered fatal. Finally, if the error shows up with some other command type, this error state is considered inconsistent.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a 660 error log with an LSB subpacket. Set software flag bits 96 (LSB), 36 (IOP CMDR), 50 (IOP CPE), and one of 45 (IOP LSB Write), 46 (IOP LSB Read), or 47 (IOP LSB Write CSR), depending on the type of command. If the LSB command is not one of those three, set 51 (IOP CPE inconsistent).

IOP LSB CSR Data Parity Error

Description: During a CSR write to the Interrupt register, the IOP detected that the data it placed on the data lines had a parity error. This is a fatal condition.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a 660 and an LSB snapshot. Set software flag bits 96 (LSB), 36 (IOP CMDR), 47 (IOP LSB Write CSR) and 52 (IOP CDPE).

IOP LSB Corrected ECC Error

Description: During an LSB transaction in which the IOP was the commander, the data cycle that was sent out was determined to have a single-bit ECC error. This results in LSB CE being set. When this occurs with the IOP being the commander, one of the following two cycles could be present on the LSB bus: a write cycle if the IOP was attempting to write to LSB memory; or a read cycle if the IOP was attempting a read from memory. In all cases, the error has been recovered by the hardware. In the write case, the memory corrects the error when converting the data from 32-bit LSB ECC to 64-bit memory ECC. In the read case, the IOP corrects the data before it continues onto the Vortex bus (see the *I/O System Technical Manual*) and down the hose. In the read case, a CPU B-cache could have been the supplier of the data. Note that there could be many, if not all CPUs trying to parse this error. One will have BDATASBE set in its LMERR register if this was B-cache supplied data.

Recovery procedure: No specific software recovery. The hardware corrects these errors.

Restart condition: Log the error and continue.

Error logging: Log a basic 660 with an LSB subpacket. Set software flag bits 96 (LSB), 36 (IOP CMDR), 53 (IOP CE), and 45 (IOP LSB Write) or 46 (IOP LSB Read), depending on the type of command.

IOP LSB Corrected ECC Error (Dirty)

Description: During an LSB transaction in which the IOP was the commander, the data cycle sent out was determined to have a single-bit ECC error. This results in LSB CE being set. When this occurs with the IOP being the commander, one of the following two cycles could be present on the LSB bus: a write cycle if the IOP was attempting to write to LSB memory; or a read cycle if the IOP was attempting a read from memory. In all cases, the error has been recovered by the hardware. In the write case, the memory corrects the error when converting the data from 32-bit LSB ECC to 64-bit memory ECC. In the read case, the IOP corrects the data before it continues onto the Vortex bus and down the hose. In the read case, a CPU B-cache could have been the supplier of the data. Note that there could be many, if not all CPUs trying to parse this error. One will have BDATASBE set in its LMERR register if this was B-cache supplied data.

Recovery procedure: No specific software recovery. The hardware corrects these errors.

Restart condition: Log the error and and continue.

Error logging: Log a basic 660 with an LSB subpacket. Set software flag bits 96 (LSB), 36 (IOP CMDR), 53 (IOP CE), and 45 (IOP LSB Write) or 46 (IOP LSB Read), depending on the type of command.

Additional parsing: BDATASBE.

IOP LSB Uncorrectable ECC Error

Description: During an LSB transaction in which the IOP was the commander, the data cycle sent out was determined to have a double-bit ECC error. This results in LSB UCE being set. When this occurs with the IOP being the commander, one of following two cycles could be present on the LSB bus. First, a write cycle. For this, the IOP was attempting to write to LSB memory. Second, the IOP could be reading from memory. In all cases, the error cannot be recovered from. In the write case, the memory inverts ECC when converting the data from 32-bit LSB ECC to 64-bit memory ECC. It will show up bad on a later read of that location. In the read case, the IOP detects the error and sets the UCE bit. The data does not go down the hose. In the read case, a CPU B-cache could have been the supplier of the data. Note that there could be many, if not all CPUs trying to parse this error. One will have BDATADBE set in its LMERR register if this was B-cache supplied data.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660 with an LSB subpacket. Set software flag bits 96 (LSB subpacket), 36 (IOP CMDR), 54 (IOP UCE), and 45 (IOP LSB Write) or 46 (IOP LSB Read), depending on the type of command.

Additional parsing: Memory address correlation.

LSB Uncorrectable ECC Error (Dirty)

Description: During an LSB transaction in which the IOP was the commander, the data cycle sent out was determined to have a double-bit ECC error. This results in LSB UCE being set. When this occurs with the IOP being the commander, one of following two cycles could be present on the LSB bus. First, a write cycle. For this, the IOP was attempting to write to LSB memory. Second, the IOP could be reading from memory. In all cases, the error cannot be recovered from. In the write case, the memory inverts ECC when converting the data from 32-bit LSB ECC to 64-bit memory ECC. It will show up bad on a later read of that location. In the read case, the IOP detects the error and sets the UCE bit. The data does not go down the hose. In the read case, a CPU B-cache could have been the supplier of the data. Note that there could be many, if not all CPUs trying to parse this error. One will have BDATADBE set in its LMERR register if this was B-cache supplied data.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660 with an LSB subpacket. Set software flag bits 96 (LSB subpacket), 36 (IOP CMDR), 54 (IOP UCE), and 45 (IOP LSB Write) or 46 (IOP LSB Read), depending on the type of command.

Additional parsing: BDATADBE.

IOP LSB Inconsistent Error

Description: During parsing the IOP LSB errors, it was determined that no supporting error bits were set to indicate why this error occurred. This is determined to be an inconsistent state.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660 with an LSB subpacket. Set software flag bits 96 (LSB subpacket), 98 (log adapter), 36 (IOP CMDR), 55 (IOP LSB inconsistent state), and 45 (IOP LSB Write) or 46 (IOP LSB Read), depending on the type of command.

IOP Multiple Data Errors

Description: When parsing the LSB errors, note the occurrence of these LSB multiple error bits. Except for CE2, in all cases, if these are set, the error conditions are not recoverable. Error state of the subsequent occurrence(s) has been lost. For CE2 errors, if the system has a rev 2 DECchip 21064, it is NOT recoverable. If the system has a rev 3 DECchip 21064, it is recoverable.

Recovery procedure: For CE2, rev 2 DECchip 21064, none; rev 3 DECchip 21064, continue; all others, none.

Restart condition: For CE2, rev 2 DECchip 21064, none; rev 3 DECchip 21064, continue; all others, none.

Error logging: No specific error logging required. Log per the parse tree branch that was true above. However, set software flag 56 (IOP CPE2), 57 (IOP CDPE2), 58 (IOP CE2), or 59 (IOP UCE2) as appropriate, to show the occurrence of this error.

IOP Node-Specific Error Summary

Description: If this condition is met, the implication is that there is an internal IOP error that needs to be serviced. If this is the case, there will be an interrupt IPL 17 pending. It is suggested that this error case be flagged in the error log but be handled by the interrupt IPL17 IOP service routine.

Recovery procedure: Clean up the LSB errors, but let the NSES errors be parsed by the IPL 17 routine when that is called.

Restart condition: None.

Error logging: Log a 660 and an LSB subpacket along with a log adapter subpacket. Set software flags 96 (LSB subpacket), 98 (log adapter), 36 (IOP CMDR), and 60 (IOP NSES).

Additional parsing: Eventually parse the subpackets.

Interrupt 660 Inconsistent Error

Description: This is a catchall for the case when all the error parsing fails to find a failure. Should never get here.

Recovery procedure: None.

Restart condition: None.

Error logging: Log a basic 660 with an LSB subpacket. Set software flag bits 96 (LSB subpacket) and 61 (660 Inconsistent).

14.3.5 Events Reported Through Entry 630

The KN7AA module detects error conditions related to the operation of the B-cache (LSB side), the Gbus, and the LSB. These errors, when occurring asynchronous to the operation of the DECchip 21064 processor, are reported through the irq_4 signal line on the KN7AA module. This, in turn, becomes an IPL31 interrupt and is vectored through offset 660. Single-bit ECC errors that are caused by either another CPU's B-cache or the LSB also cause a 660 to occur.

The KN7AA also vectors through 660 if other nodes assert the LSB ERR signal. This is the case for the IOP. Error handling in the 660 domain includes monitoring and error checking for IOP-detected LSB errors. See the *I/O System Technical Manual* for discussion IOP error parse trees.

Figure 14-6 is the parse tree associated with the 660 machine check. Following the parse tree is a description of each type of error and, when possible, a suggested recovery method.

The KN7AA PALcode entry 630 pertains to all DECchip 21064 processor detected single-bit ECC errors that occur when a P-cache fill is in progress and the error occurs on this module. Two situations cause this to occur: first, when the B-cache is the source; second, when the EDAL causes the error. Upon detection of this error, the system jumps to PALcode, which has the routines required to coordinate recovery of this error based on the two DECchip 21064 revisions, 2.1 or 3.0. Note that if the source of the errors is an LSB request (that is, LBER<E> is set), then the single-bit error is handled in the 660 parse trees. The LSB error causes the 660 error interrupt and it takes precedence over the 630. For these cases, a 660 error log will be provided. The 630 will be dismissed and will not produce a 630 error log.

14.3.5.1 DECchip 21064 Revision 2.1

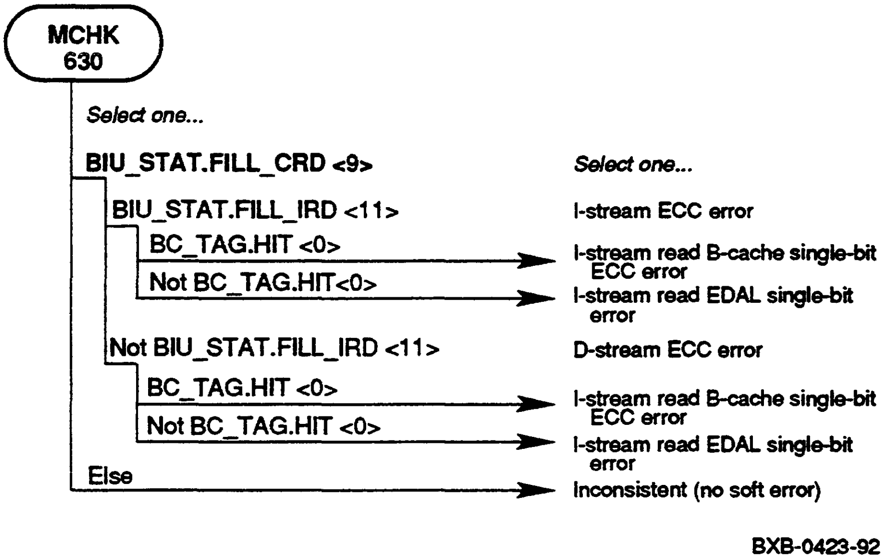
If the DECchip 21064 processor is at revision 2.1, the recovery is tried at the PALcode level. For rev 2.1 DECchip 21064 processors, recovery will only occur for single-bit ECC errors detected during I-stream reads. All other single-bit ECC errors detected by the CPU will result in no correction being performed. These uncorrected errors will produce a 670 machine check. Those errors must be parsed with the 670 parse tree.

14.3.5.2 DECchip 21064-C Revision 3.0

For revision 3.0 DECchip 21064s, single-bit ECC recovery is performed by the hardware. For single-bit errors on I-cache fills which corrupt more than a single quadword of the cache fill, the DECchip 21064 traps to PALcode and attempts to recover by flushing the I-cache. All errors that are not recoverable will produce a 670 machine check.

Figure 14-7 shows the PALcode entry 630 parse tree.

Figure 14-7 PALcode 630 Parse Tree


MCHK
630
Select one...
BIU_STAT.FILL_CRD <9>
Select one...
BIU_STAT.FILL_IRD <11>
I-stream ECC error
BC_TAG.HIT <0>
I-stream read B-cache single-bit ECC error
Not BC_TAG.HIT<0>
I-stream read EDAL single-bit error
Not BIU_STAT.FILL_IRD <11>
D-stream ECC error
BC_TAG.HIT <0>
I-stream read B-cache single-bit ECC error
Not BC_TAG.HIT <0>
I-stream read EDAL single-bit error
Else
Inconsistent (no soft error)
BXB-0423-92

I-Stream Read B-Cache Single-Bit ECC Error

Description: During an I-stream reference with a B-cache hit, the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register.

Recovery procedure: Hardware and PALcode do recovery.

Restart condition: Restart if corrected by PALcode or hardware.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 6. Also, log the 630 machine check stack frame to the EEPROM logging area.

I-Stream Read EDAL Single-Bit ECC Error

Description: During an I-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. Note that this error was caused by the EDAL data path and no LSB errors occurred on this data transfer.

Recovery procedure: hardware and PALcode do recovery.

Restart condition: Restart if corrected by PALcode or hardware.

Error logging: For this error, the basic machine check entry is fine. Set software flag bit 11. Log the 630 stack to the EEPROM area.

D-Stream Read B-Cache Single-Bit ECC Error

Description: During a D-stream reference with a B-cache hit, the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register.

Recovery procedure: Hardware and PALcode do recovery.

Restart condition: Restart if corrected by PALcode or hardware.

Error logging: For this error, the basic machine check entry will do. All error state associated with this error resides on this CPU module. Set software flag bit 16. Log the 630 stack frame to the EEPROM area.

D-Stream Read EDAL Single-Bit Error

Description: During a D-stream reference the DECchip 21064 detected a correctable ECC error. The failing syndrome is latched in the FILL_SYND register. Note that this error was caused by the EDAL data path.

Recovery procedure: Hardware and PALcode do recovery.

Restart condition: Restart if corrected by PALcode or hardware.

Error logging: For this error, the basic machine check entry is fine. Set software flag bit 21. Log the 630 stack to the EEPROM area.

Index