21464 Internal Design Specification

Available Internally from: HTTP://segsrv.hlo.dec.com/arana

This document specifies the internal design for the Alpha microprocessor that is also known as EV8 and Araña.

Revision/Update Information: Revision 1.1k, January, 2001

COMPAQ

Compaq Computer Corporation
Shrewsbury, Massachusetts

January 2001

Contents

3 Instruction Fetch Unit — the Ibox

4 Dependency Mapper Unit — the Pbox

5 Instruction Issue and Retire Unit — the Qbox

6 Integer Execution Unit — the Ebox

7 Register File

8 Floating-Point Execution Units — the Fbox

9 Memory Instruction Execution Unit — the Mbox

10 Internal Ring Bus

11 Second-Level Cache and Controller (Cbox)

12 Cache Coherence Protocol Processing

13 Router Interface — the Rbox

14 Rambus Interface — the Zbox

15 Miscellaneous Interfaces

16 Internal Processor Registers

17 Privileged Architecture Library Code

18 Initialization and Configuration

19 Performance Monitoring

20 Hardware Debug Features

21 Testability and Diagnostics

22 Error Detection and Error Handling

23 Hardware Interface

C Proposed Memory Management IPR Design

Figures

Tables

Preface

Audience

This specification is for system designers and programmers who are involved in the Alpha 21464 microprocessor engineering project.

Organization

This specification contains the following chapters. A top-level presentation of the main topics in these chapters is presented in Chapter 2.

Chapter 1, Introduction, which describes the terminology and conventions that are used in this specification.

Chapter 2, Architecture Overview, which summarizes the 21464 new features and design organization.

Chapter 3, Instruction Fetch Unit — the Ibox, which describes the first part of the instruction unit microarchitecture.

Chapter 4, Dependency Mapper Unit — the Pbox, which describes the second part of the instruction unit microarchitecture.

Chapter 5, Instruction Issue and Retire Unit — the Qbox, which describes the third part of the instruction unit microarchitecture.

Chapter 6, Integer Execution Unit — the Ebox, which describes how integer instructions are executed.

Chapter 7, Register File, which describes the creation and management of the virtual and physical registers in that file.

Chapter 8, Floating-Point Execution Units — the Fbox, which describes how floating-point instructions are executed.

Chapter 9, Memory Instruction Execution Unit — the Mbox, which describes how memory-reference instructions are executed.

Chapter 10, Internal Ring Bus, which describes the bus that connects the Cbox, Rbox, and Zbox.

Chapter 11, Second-Level Cache and Controller (Cbox), which describes how the second-level cache is controlled.

Chapter 12, Cache Coherence Protocol Processing, which describes how caches in a multprocessor system maintain their coherency.

Chapter 13, Router Interface — the Rbox , which describes the interprocessor switch.

Chapter 14, Rambus Interface — the Zbox, which describes that interface.

Chapter 15, Miscellaneous Interfaces, which describes the GIO Port.

Chapter 16, Internal Processor Registers, which describes those registers.

Chapter 17, Privileged Architecture Library Code, which describes the interface between the microarchitecture and the PALcode environment.

Chapter 18, Initialization and Configuration, which describes the sequences that are used in the initialization and configuration of the microprocessor, along with their characteristics.

Chapter 19, Performance Monitoring, which describes the means available for monitoring the performance of the 21464.

Chapter 20, Hardware Debug Features, which describes the physical capabilities that have been placed in the 21464 to aid debugging.

Chapter 21, Testability and Diagnostics, which describes the capabilities that have been placed in the 21464 to aid in testing and performing diagnostics.

Chapter 22, Error Detection and Error Handling, which describes the various error detection mechanisms that have been placed in the 21464 and the corresponding recovery procedures.

Chapter 23, Hardware Interface, which describes the 21464 at the level of its interface pins.

Chapter 24, New Instructions, which describes instructions that are new for the 21464.

Chapter 25, System Configurations, which describes considerations for configuring systems.

Chapter 26, Physical Addressing and Input/Output, which describes physical addressing and Input/output considerations.

Chapter 27, Requirements to Support "Tandem", which describes those parts of the design that are significant to Tandem machines that will use the 21464.

A Glossary, which provides the definition of the terms used in the specification for which the definitions can be specific to this specification.

An Index, which provides the appropriate references into the specification.

Related Documentation

The following documents are included by reference in this specification:

- The Alpha System Reference Manual (the SRM), Version 7
- The ALPHA_SRM notesfile, which includes an on-going discussion of topics related to this design

To obtain an SRM and access to the ALPHA_SRM notesfile, send mail to Audrey.Reith@Compaq.com.

The following documents are referenced in this specification. These documents can provide historical context, supporting information, additional information, or be of general interest to those using this specification. These documents are available in the same general directory as the specification and can be viewed in your browser.

1 Introduction

1.1 Terminology and Conventions

This section defines the abbreviations, terminology, and other conventions used throughout this document.

Abbreviations

- Binary Multiples

  The abbreviations K, M, and G (kilo, mega, and giga) represent binary multiples and have the following values.

  K = 2^{10} (1024)
  M = 2^{20} (1,048,576)
  G = 2^{30} (1,073,741,824)

  For example:

2KB	=	2 kilobytes	=	2×2^{10} bytes
4MB	=	4 megabytes	=	4×2^{20} bytes
8GB	=	8 gigabytes	=	8×2^{30} bytes
2K pixels	=	2 kilopixels	=	2×2^{10} pixels
4M pixels	=	4 megapixels	=	4×2^{20} pixels

- Register Access

  The abbreviations used to indicate the type of access to register fields and bits have the following definitions:

Abbreviation	Meaning
IGN	Ignore Bits and fields specified are ignored on writes.
MBZ	Must Be Zero Software must never place a nonzero value in bits and fields specified as MBZ. A nonzero read produces an Illegal Operand exception. Also, MBZ fields are reserved for future use.
RAZ	Read As Zero Bits and fields return a zero when read.

Abbreviation	Meaning
RC	Read Clears Bits and fields are cleared when read. Unless otherwise specified, such bits cannot be written.
RES	Reserved Bits and fields are reserved by Compaq and should not be used; however, zeros can be written to reserved fields that cannot be masked.
RO	Read Only The value may be read by software. It is written by hardware. Software write operations are ignored.
RO,*n*	Read Only, and takes the value *n* at power-on reset. The value may be read by software. It is written by hardware. Software write operations are ignored.
RW	Read/Write Bits and fields can be read and written.
RW,*n*	Read/Write, and takes the value *n* at power-on reset. Bits and fields can be read and written.
W1C	Write One to Clear If read operations are allowed to the register, then the value may be read by software. If it is a write-only register, then a read operation by software returns an UNPREDICTABLE result. Software write operations of a 1 cause the bit to be cleared by hardware. Software write operations of a 0 do not modify the state of the bit.
W1S	Write One to Set If read operations are allowed to the register, then the value may be read by software. If it is a write-only register, then a read operation by software returns an UNPREDICTABLE result. Software write operations of a 1 cause the bit to be set by hardware. Software write operations of a 0 do not modify the state of the bit.
WO	Write Only Bits and fields can be written but not read.
WO,*n*	Write Only, and takes the value *n* at power-on reset. Bits and fields can be written but not read.

- Sign extension

  SEXT(x) means *x* is sign-extended to the required size.

Addresses

Unless otherwise noted, all addresses and offsets are hexadecimal.

Aligned and Unaligned

The terms *aligned* and *naturally aligned* are interchangeable and refer to data objects that are powers of two in size. An aligned datum of size $2n$ is stored in memory at a byte address that is a multiple of $2n$; that is, one that has n low-order zeros. For example, an aligned 64-byte stack frame has a memory address that is a multiple of 64.

A datum of size $2n$ is *unaligned* if it is stored in a byte address that is not a multiple of $2n$.

Bit Notation

Multiple-bit fields can include contiguous and noncontiguous bits contained in square brackets ([]). Multiple contiguous bits are indicated by a pair of numbers separated by a colon [:]. For example, [9:7,5,2:0] specifies bits 9,8,7,5,2,1, and 0. Similarly, single bits are frequently indicated with square brackets. For example, [27] specifies bit 27. See also Field Notation.

Caution

Cautions indicate potential damage to equipment or loss of data.

Data Units

The following data unit terminology is used throughout this manual.

Term	Words	Bytes	Bits	Other
Byte	½	1	8	—
Word	1	2	16	—
Longword	2	4	32	Dword
Quadword	4	8	64	2 longword

Do Not Care (X)

A capital X represents any valid value.

External

Unless otherwise stated, external means not contained in the chip.

Field Notation

The names of single-bit and multiple-bit fields can be used rather than the actual bit numbers (see Bit Notation). When the field name is used, it is contained in square brackets ([]). For example, **RegisterName[LowByte]** specifies **RegisterName[7:0]**.

Note

Notes emphasize particularly important information.

Numbering

All numbers are decimal or hexadecimal unless otherwise indicated. The prefix 0x indicates a hexadecimal number. For example, 19 is decimal, but 0x19 and 0x19A are hexadecimal (also see Addresses). Otherwise, the base is indicated by a subscript; for example, 100_2 is a binary number.

Ranges and Extents

Ranges are specified by a pair of numbers separated by two periods (..) and are inclusive. For example, a range of integers 0..4 includes the integers 0, 1, 2, 3, and 4.

Extents are specified by a pair of numbers in square brackets ([]) separated by a colon (:) and are inclusive. Bit fields are often specified as extents. For example, bits [7:3] specifies bits 7, 6, 5, 4, and 3.

Register Figures

The gray areas in register figures indicate reserved or unused bits and fields.

Bit ranges that are coupled with the field name specify the bits of the named field that are included in the register. The bit range may, but need not necessarily, correspond to the bit *Extent* in the register.

Signal Names

The following examples describe signal-name conventions used in this document.

AlphaSignal[n:n]	Boldface, mixed-case type denotes signal names that are assigned internal and external to the 21464 (that is, the signal traverses a chip interface pin).
AlphaSignal_*x*[n:n]	When a signal has high and low assertion states, a lowercase italic *x* represents the assertion states. For example, **SignalName_*x*[3:0]** represents **SignalName_H[3:0]** and **SignalName_L[3:0]**.

UNDEFINED

Operations specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing to stopping system operation.

UNDEFINED operations may halt the processor or cause it to lose information. However, UNDEFINED operations must not cause the processor to hang, that is, reach an unhalted state from which there is no transition to a normal state in which the machine executes instructions.

UNPREDICTABLE

UNPREDICTABLE results or occurrences do not disrupt the basic operation of the processor; it continues to execute instructions in its normal manner. Further:

- Results or occurrences specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE.
- An UNPREDICTABLE result may acquire an arbitrary value subject to a few constraints. Such a result may be an arbitrary function of the input operands or of any state information that is accessible to the process in its current access mode. UNPREDICTABLE results may be unchanged from their previous values.

  Operations that produce UNPREDICTABLE results may also produce exceptions.
- An occurrence specified as UNPREDICTABLE may happen or not based on an arbitrary choice function. The choice function is subject to the same constraints as are UNPREDICTABLE results and, in particular, must not constitute a security hole.

  Specifically, UNPREDICTABLE results must not depend upon, or be a function of, the contents of memory locations or registers that are inaccessible to the current process in the current access mode.

Also, operations that may produce UNPREDICTABLE results must not:

- Write or modify the contents of memory locations or registers to which the current process in the current access mode does not have access, or
- Halt or hang the system or any of its components.

For example, a security hole would exist if some UNPREDICTABLE result depended on the value of a register in another process, on the contents of processor temporary registers left behind by some previously running process, or on a sequence of actions of different processes.

X

Do not care. A capital X represents any valid value.

2
Architecture Overview

This chapter presents an overview of the major parts of the 21464 microarchitecture.

- The new features of the 21464
- Microarchitecture diagram, a high-level view of the overall architecture
- Simultaneous multithreading (SMT), the essential new performance element of the 21464 design
- Instruction unit, composed of the Ibox, Pbox, and Qbox
- Execution unit, composed of the Register File, Ebox, and Fbox
- Memory controller unit, composed of the Mbox
- External interface, composed of the Cbox, Rbox, and Zbox
- Pipeline organization
- Instruction Execution Pipelines and Latency
- Instruction issue and retire rules
- New Instructions
- Implementation-specific execution of the CMOV and FCMOV instructions
- Interrupt handling
- AMASK and IMPLVER instruction values
- Performance monitoring features

2.1 New Features

The 21464 can be summarized as follows.

2.1.1 Processor Features

The processor has the following characteristics:

- Instruction issue and execute out of order

  Dynamic four-way simultaneous multi-threading (SMT)

  Up to eight instructions mapped, issued, executed, and retired per cycle, from the following menu:

  – Up to eight integer operations, including branches

  - Up to four floating-point operations
  - Up to four memory references
  - Up to four multimedia operations
- Latency is one cycle for most integer operations and three cycles for loads and most floating-point operations
- Store-sets memory dependence predictor – for predicting store-load dependencies
- Fetches up to 16 instructions for each cycle
- Collapsing instruction buffer for merging basic blocks
- Up to 256 instructions in flight
- 128 entry instruction queue
- 1.4 GHz clock rate, resulting in a 700 psec cycle
- Peak instruction rate exceeds 11 billion instructions per second (gigaops)
- New SIMD instructions for video, graphics, and signal-processing applications
- Unified register file (integer and floating point)
  - 512 quadword capacity
  - 16 read ports
  - 8 write ports
- Instruction L1 cache (Icache)
  - 64 KB capacity, 2-way pseudo-set associative
  - 64 byte (16-instruction) block size
  - 8 instructions per cycle from each of two addresses
  - Bandwidth is 90 GB/sec
  - Parity protected
- Data L1 cache (Dcache)
  - 64 KB capacity
  - Two-way set associative
  - 64 byte (8-quadword) block size
  - 8 bytes per cycle read from each of three addresses
  - Write-through concurrent with reads, subject to bank conflict
  - Function unit bandwidth: 32 bytes per cycle, resulting in 45 GB/sec
  - Hit latency is three cycles
  - Fill/write bandwidth is 64 bytes per cycle, resulting in 90 GB/sec
  - Parity protected
- Onchip L2 cache (Scache)
  - 3 MB capacity

  - Six-way set-associative
  - 64 byte block size
  - Write-back
  - DC/IC fill bandwidth: 64 bytes per cycle, resulting in 90 GB/sec
  - Best-case hit latency is 10 cycles
  - DC write-through bandwidth is 16 bytes per cycle, resulting in 22 GB/sec
  - Peak Scache fill rate is 32 bytes per cycle, resulting in 45 GB/sec
  - ECC protected by quadwords
- 52-bit virtual address, 48-bit physical address, 8-bit ASN
- 8K and 64K page sizes, granularity hint for bigger contiguous regions
- Independent 128-entry fully-associative ITB and DTB, with superpages for kernel maps

2.1.2 Memory Features

The memory has the following characteristics:

- Glueless interface to Rambus main memory
- Two independent interleavable RDRAM ports per processor, with optional four-way processor striping
- Each port consists of four channels
- All transactions in units of 64-byte (512-bit) blocks
- Optional redundant fifth channel protects against full-chip failure
- Each channel supports up to 32 RDRAM chips
- Each processor can support up to 256 RDRAM chips (plus redundancy)
- With 1 GB parts, 32 GB per processor (35 address bits)
- Peak processor memory bandwidth is 200M blocks per second, or 12.8 GB per second.
- With redundant channel deployed, system tolerates total failure of a memory chip plus single-bit errors in another chip.
- Without redundant channel, system corrects single-bit errors in memory and detects double-bit errors.

2.1.3 Multiprocessor Features

The 21464 provides the following support for multiprocessor configurations:

- Up to 512 processors with main memory and coherent caches
- Fully-distributed, non-blocking, directory-based CC-NUMA coherence protocol
- Optional I/O node per processor, may have cache and/or I/O memory but not cacheable main memory
- Glueless torus configuration — others possible with switch ASIC's

- Maximum total physical memory is 2^{44} bytes = 16 Terabytes.
- Peak instruction rate exceeds 5.7 trillion instructions per second (teraops).
- Buffered crossbar switch fabric with virtual circuits
- In 21364 mode, each network port supports 3.2 GB per second in and out
  - Four-port network throughput is 12.8 GB per second
- In 21464 mode, each port supports 4.8 GB per second in and out.
  - Five-port network throughput is 24 GB per second
- Bisection bandwidth of a 16x32 torus, cut across the narrow axis, is more than 300 GB per second

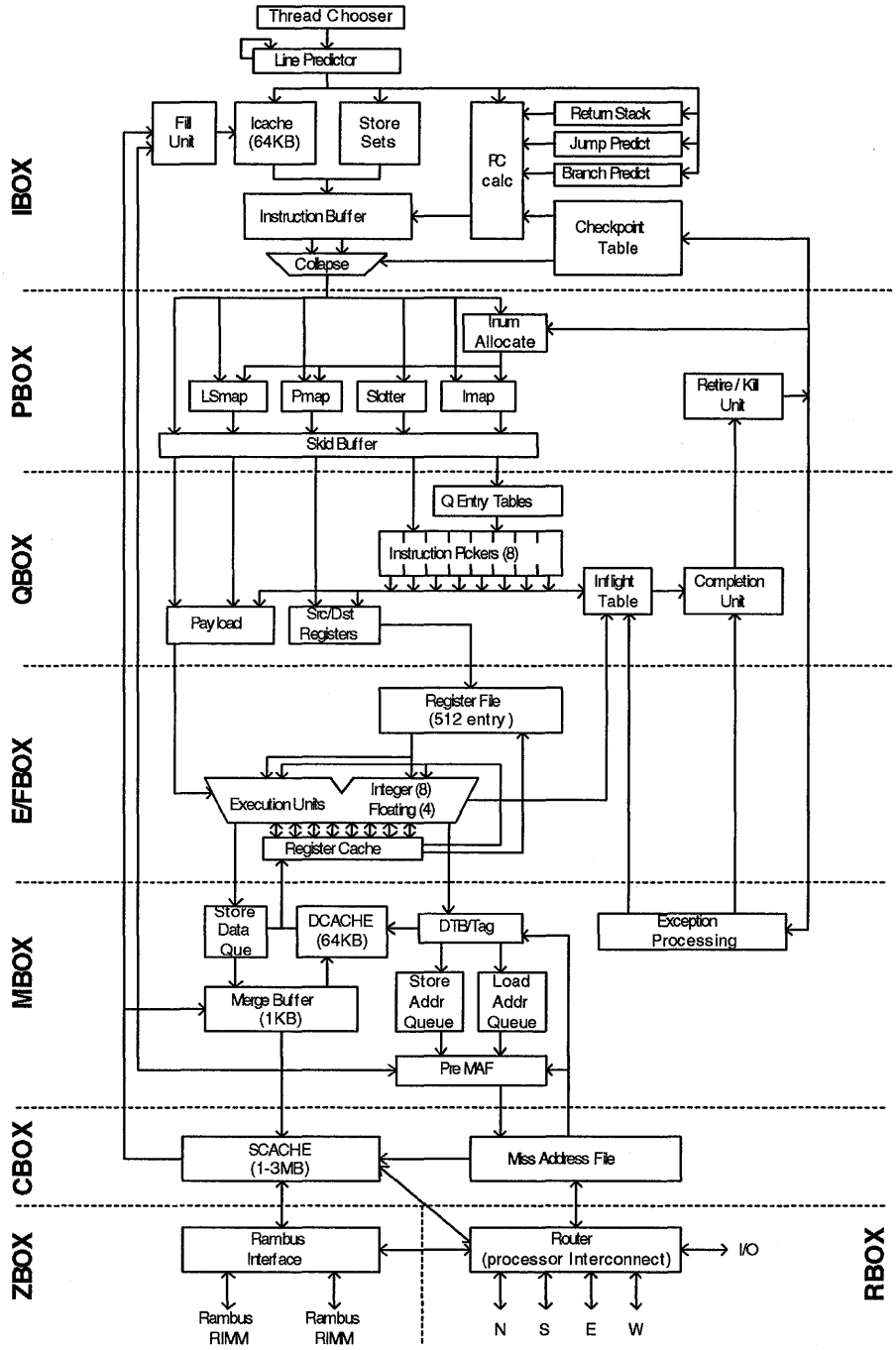
2.2 Microarchitecture Diagram

Figure 2–1 shows a simplified block diagram of the 21464 microarchitecture. As listed in Table 2–1, the microarchitecture of the 21464 is separated into four major sections or units, each of which contain one or more functional subsections, called boxes.

Table 2–1 Microarchitecture Major Sections Summary

Major Section	Subsection	Description
Instruction Unit	Ibox	Instruction fetch unit
	Pbox	Instruction processing (dependency resolution) unit
	Qbox	Instruction issue and retire unit
Execution Unit	Register file	
	Ebox	Integer instruction execution unit
	Fbox	Floating-point instruction execution unit
Memory Controller Unit	Mbox	Memory-reference instruction execution unit
External Interface Unit	Cbox	Second-level cache (Scache) controller
	Rbox	Router controller
	Zbox	Rambus memory controller

Figure 2–1 21464 Block Diagram



2.3 Simultaneous Multithreading (SMT)

SMT differs from the more traditional forms of hardware multithreading in that every thread can compete for issue slots at every cycle. Traditional multithreading designs tend to invoke alternative threads only on second-level cache misses or to schedule the threads in a rigid, round-robin fashion. The result is less resource utilization and less performance improvement.

The 21464 can execute up to four programs simultaneously, each program running in one of the four thread processing units (TPUs). While each TPU has some dedicated hardware, most resources are shared between the four TPUs. Maximum single-stream performance occurs when a single program is the only active thread in the CPU. In that case, most chip resources are available to the active TPU (the single program) and the design makes no compromises in single-stream performance. On the other hand, because many programs cannot always use all the chip resources, it is often possible to at least double overall throughput by running four programs simultaneously.

SMT adds very little cost to a single processor and can be used either to increase throughput while executing independent programs or to speed up a single task that has been decomposed into separate threads. The 21464 adds two instructions, LDx_ARM and QUIESCE, to provide easy synchronization between cooperating threads. LDx_ARM sets up a memory address register that monitors memory traffic. QUIESCE suspends a thread until the memory location is written or a time-out counter expires; the thread does not consume any resources while waiting for the signal to continue. See Appendix B for information on the LDx_L and QUIESCE instructions.

Each TPU has its own dedicated program state that consists of 32 integer registers, 32 floating-point registers, a PC, and internal processor registers (IPRs). Also, some microarchitectural structures, such as the Return Stack and the Instruction Buffer, are statically divided into four parts, with each part dedicated to a TPU. However, most microarchitecture structures are dynamically shared among the TPUs on an as-needed basis. Dynamically shared structures include the caches, the translation buffers, the branch predictor, functional execution units, and the instruction queue.

Overview of SMT Operation

Most thread-specific operations take place in the Ibox, the front end of the CPU pipeline. The Ibox is time-multiplexed on a cycle-by-cycle basis between four active threads, each being given equal priority. The Ibox thread fetch chooser normally fetches instructions from that thread with the fewest instructions in the instruction queue. This policy helps programs with high ILP go as fast as they can, yet provides instructions to programs with low ILP as those instructions are needed.

The Ibox accesses the line predictor after the thread fetch chooser selects a thread from which to fetch an instruction. At the next cycle, the resulting indexes are used to access the Icache and the branch predictor. The two fetch chunks are stored in the instruction buffer. The thread map chooser selects a thread and reads its two oldest fetch chunks from the instruction buffer and collapses them into a single map chunk, which is sent to the mapper in the Pbox. The mapper maps the registers, assigns INums, and slots the instructions to the various pickers as the instructions are entered into the instruction queue. The INum space is divided into a four segments for the four threads (the four TPUs) and the INums are thread-specific because they keep track of program order.

The instruction queue then contains a mix of instructions from the four threads. Once instructions are in the instruction queue, they are eligible to issue when their source operands are available, regardless of which thread they belong to. The oldest, issue-ready instruction is chosen by each picker and sent to the appropriate Ebox or Fbox execution unit. Because the threads have no register dependencies on each other, it is much more likely that eight instructions are continuously ready to issue.

2.4 Instruction Unit

The instruction unit consists of the Ibox, Pbox, and Qbox. The Ibox is the instruction fetch engine. It provides high instruction-stream bandwidth to the remainder of the chip. Specifically, the Ibox delivers instructions directly to the Pbox, which is responsible for instruction number (INum) resource management, dependence analysis and register renaming. From there, instructions proceed to the Qbox, where they await the resolution of their source register dependencies. Once an instruction's register dependencies have been resolved, the instruction is issued, provided that it wins arbitration for an appropriate functional unit in the Ebox (arithmetic and logic integer operations), Fbox (arithmetic floating point operations) or Mbox (memory operations). Once an instruction has completed execution, it retires when it is the oldest non-retired instruction in the machine for the appropriate thread processing unit (TPU) context.

2.4.1 Instruction Fetch Unit — the Ibox

Instruction stream bandwidth is one of the major factors in overall chip performance. A program cannot execute faster than the rate of its instructions entering the machine. Achieving sufficient instruction bandwidth for a machine that can execute up to eight instructions per cycle poses several challenges. In order to meet those challanges, the Ibox contains many new features that were not designed into prior Alpha implementations.

Features

The Ibox delivers up to eight instructions per cycle to the remainder of the machine. The Ibox maintains the correct program counter (PC) while the CPU executes programs and receives interrupts and exceptions to properly redirect the machine.

The Ibox contains the following new features to support high-bandwidth instruction-stream fetching, advanced control flow prediction, simultaneous multithreading (SMT), and memory dependence prediction:

- An Icache size of 64 KB or 16K instructions
- Up to two potentially noncontiguous cache blocks are fetched per cycle
- A fetch TPU chooser that creates a resource-balanced SMT fetch engine
- Advanced branch prediction that predicts up to 16 branches per cycle
- History-based jump target prediction
- A collapsing buffer that facilitates over-fetching and merging fetch blocks
- Memory dependence prediction that uses store sets
- Advanced hardware Istream prefetching
- A simultaneous multithreaded fill unit
- An anti-thrashing Icache fill policy

The Ibox components can be grouped into the following major sections:

Table 2–2 Ibox Major Component Summary

Name	Description
Checkpoint Unit	The Checkpoint Unit maintains state for restarting the CPU in the event of an exception, and trains the control flow predictors and the memory dependence predictor. In the event of an exception, the Checkpoint Unit resets the PC, branch predictor, jump target predictor, and return stack to the state that existed just before the fetch of the instruction that caused an exception. Training information for the branch and jump target predictors is also kept and used to train the predictors at the retirement time of branch or jump instructions.
Control Flow Prediction Unit	The Control Flow Prediction Unit predicts PC changes at fetch-time for instructions that can change control flow when executed: conditional branches, computed jumps, and subroutine returns. There is a corresponding dedicated predictor for each: the conditional branch predictor, the jump target predictor, and the return address stack.
Fill Unit	The Fill Unit fetches instructions from lower-level memory and can fetch instruction blocks for multiple TPUs simultaneously. The Fill Unit maintains a dynamic hardware prefetcher that attempts to fill the Icache with blocks that would have missed in the future. The Fill Unit also contains the Icache Translation Buffer (ITB) that translates virtual PC miss addresses to physical addresses before making memory requests.
Index Unit	The Index Unit produces up to two indexes per cycle. The indexes are usually predictions from the Line Predictor that are used to access the Icache, Branch Predictor, and Store Sets Array. The Index Unit also contains the Fetch TPU Chooser that arbitrates among multiple TPUs that are ready to fetch instructions. The indexes that are produced will have an associated TPU that is sent along with the indexes down the Ibox pipeline. The Line Predictor itself consists of a sequential and non-sequential component, to address the sequential and non-sequential code sequences of the running programs.
Instruction Processing Unit	The Instruction Processing Unit stores and retrieves instructions and associated tags and data into its 64KB Icache and associated tag array. Instruction pre-decode bits are also stored in the Icache data and tag arrays to speed instruction processing in the Ibox and instruction format decoding in the Pbox. The Instruction Processing Unit also contains the Store Sets Array, which produces memory synchronization identifiers called store sets for potentially every load and store operation. The store sets instruct the Pbox to create explicit dependencies between certain loads and stores. The Instruction Processing Unit also contains the Collapsing Buffer, which stores instruction blocks that are driven by the Icache and collapses up to two instruction blocks per cycle to deliver up to 8 instructions per cycle to the Pbox.
PC Unit	The PC Unit maintaines the program counters for each TPU. Typically, the PC Unit calculates PCs based on the exiting instructions of the fetch blocks (such as branches, jumps, returns, fall-through, and so forth), but it also can be reset by interrupts and exceptions. The PC Unit is also responsible for determining Icache misses, index mispredicts, and way mispredicts in the Ibox pipeline.

2.4.2 Dependency Mapper Unit — the Pbox

The Pbox processes instructions that are fetched by the Ibox. The Pbox assigns INums (instruction numbers) to the instructions, analyzes the data dependencies between instructions, and maps their architectural source and destination values into physical registers. The Pbox also maintains data structures that allow recovery of all relevant processor state that corresponds to the architectural state of the machine prior to any un-retired instruction. This allows the processor to perform rapid trap recovery in the presence of branch mispredicts or other exception conditions. The Pbox passes the renamed instructions to the Qbox for scheduling and dispatch.

The Pbox consists of the following components:

Table 2–3 Pbox Major Component Summary

Name	Description
Bid/Grant Exception Logic	Chooses which of the pending kills from all TPUs should be broadcast to the rest of the chip.
Instruction Decoder	Decodes each of the eight instructions that arrive in a cycle. The decoder is placed early in the pipe to aid slotting decisions and to provide inputs to the load/store flow control mechanisms and to the IPR interlock mechanisms
INum Allocator	Allocates INums to new map blocks sent down by the Ibox. Also contains the Map Thread Chooser, which picks the next thread that will map instruction blocks and subsequently informs the Ibox
INum Mapper	Maps source operand registers (VReg) into the INum of the last writer for the source operand
Load/Store Serial Number Allocator	Associates a sequential identifier with each load instruction (LUum) and a second identifier with each store instruction (SNum). These LNums and SNums prevent deadlock and manage flow control into the Mbox load and store queues
Mapper Exception Logic	When notified of an exception by the Bid/Grant Exception Logic, rolls the Inum Mapper, Physical Register May, Load/Store Serial Number, and RC/RS Interupt Flag Widget state back to the trap point
Physical Register Map	Allocates physical destination registers to each dispatched instruction. This table is also used to map virtual register operands into the corresponding physical registers
Post-Map Skid Buffer	Holds a silo of the last few map blocks that have passed through the Pbox forward path
RC/RS Interrupt Flag Widget	Maintains state necessary to implement the RC/RS instructions
Retire/Kill Unit	Communicates the identity of retired and/or killed instructions to all concerned boxes by way of the Retire/Kill bus
Memory Queue Allocation Unit	Governs the allocation and deallocation of load queue (LQ) and store queue (SQ) chunks to memory instructions. Also controls the High-Water Mark (HWM) that is sent to the Qbox to regulate the issuing of loads and stores.

2.4.3 Instruction Issue and Retire Unit — the Qbox

The Qbox processes instructions that are renamed by the Pbox, and determines an appropriate schedule for those instructions. Instructions cannot be executed until they are "data ready", until their dependencies have been resolved. The Qbox can identify a data-ready instruction by checking to see that both of its parent entries have asserted their result-ready signals. This method is called a "decoded-space" dependence array.

The Qbox attempts to choose the "best" 8 instructions to execute for each tic of the clock from a "window" of 128 candidates that are received from the Pbox. Each of the eight scheduling pipelines can handle a subset of the 128 candidate instructions. Because the subset can contain (in some cases) up to half of the instructions in the window, the Qbox includes "pickers" that choose the best instruction out of a set of 64 candidates.

Scheduling is a four step process:

1. Identify all data-ready instructions.

2. For each pipe, select the ``oldest" data-ready instruction enabled for execution in that pipe.
3. Assert the result-ready signal that corresponds to each selected instruction, so that all instructions that are stored in the instruction queue can see that the chosen instructions have been issued.
4. For each instruction in the instruction queue, test the result-ready signal for each operand for each instruction in that queue.

The Qbox selects the eight best "data ready" instructions for execution in eight integer pipeline units and four floating-point pipeline units. In addition, the Qbox selects up to four data-ready branch instructions for resolution in each cycle. It also retires all eligible instructions, committing them to architectural state.

The Qbox consists of the following components:

Table 2–4 Qbox Major Component Summary

Name	Description
Bid Enable Logic	Prevents otherwise-ready instructions from bidding in pipes that cannot service them, either because of a slotting decision or because of non-data-related resource conflict.
Completion Unit	Tracks which instructions have issued, which have passed their trap points, which are I/O instructions, and which have retired.
Dependency Arrays	Contains an identifier for the producer of each operand for each instruction in the instruction queue.
Destination Register Number Array	Contains the destination register specifiers for each instruction. This array are separately located from the SRN because it is not on any performance-critical paths.
Exception Kill Logic	Removes from the Instruction Queue any instructions that have been killed due to an exception.
FPCR Control	Controls the update of the FPCR in the Fbox. The FCR, along with the native mode FPCR trap and PALmode fetch barrier, guarantees the correct architecture (in-order) behavior of writing and reading the FPCR register.
InFlight Table	Tracks instructions that have issued and feeds INums that have passed their trap points to the Completion unit.
Instruction Queue	The queue from which instructions are picked for execution.
Load/Poison Re-arm Widget	Handles notification of load/miss events from the Mbox and ensures that all instructions that depend on a missed load will replay at some later time. The LPR also determines when individual instructions are eligible to be deallocated.
Load/Store Number High-Water Marker	Disables load and store instructions whose LSNums indicate that there may not be space available for them in the Mbox load/store queues. Also contains the logic for preserving the consistency of the DTB on misses.
Oldest CBR Selector	Identifies the oldest conditional branch issuing in the current cycle (that is, the one most likely to cause a misprediction).
Payload Array	Contains all the instructions and the register file addresses of all operands.
Picker Arrays	On each cycle, chooses the oldest data-ready instruction for each execution pipeline.
Post-Issue Logic	Gathers bubble requests and routes them to the appropriate pipelines. The Post-Issue Logic is also responsible for sequencing completion signals for the floating-point pipelines.

Table 2–4 Qbox Major Component Summary (Continued)

Name	Description
Profile-Me Collection	Collects the following instruction-time-oriented performance data for the two in-flight profile-me instructions: data ready, bid, issue, deallocation, and queue chunk deallocation.
Queue Chunk Allocator/ Deallocator	Manages the 32 chunks for instruction queue allocation. Picks the two chunks to be allocated to the next group of eight instructions.
Queue Entry Table	Translates INum dependencies delivered from the Pbox INum Mapper stage into queue entry number dependencies. The queue entry table also sets the No Live Dependency bits, when, for example, an instruction is data-ready upon entry into the queue.
Source Registers Number Arrays	Contain the indexes of the physical registers assigned to each source operand of each instruction. These arrays (there are two) are kept close to the dependence/bid/grant logic as the launch of the input physical register specifiers may be a critical path.

2.5 Execution Unit

The execution unit receives instruction information from the Qbox Payload Array and the Qbox source and destination register number arrays (SRNs and DRN). The former is received directly by the Ebox or Fbox execution units; the latter by the Register File.

2.5.1 Register File

Although the Alpha architecture only defines 64 registers, the 21464 is a multi-threaded, out-of-order machine that requires many more than 64 registers to keep its pipelines full. The four independent threads each require 64 registers, and an additional 256 temporary registers are used to rename registers of inflight instructions to eliminate write-after-read and write-after-write conflicts. At 65 bits per entry, 512-entries result in a 4KB register file.

Eight parallel execution units can consume up to 16 source operands and can produce up to eight results per cycle. The 21464 implements each of 32K 'not-so-little' RAM cells with 16 read ports and 8 write ports. Although such an implementation is not trivial, defining a register file with fewer ports would have forced the Qbox to either issue instructions based on the number of operands needed from the register file, or trap whenever the set of issued instructions needed more than the available number of ports.

2.5.2 Integer Instruction Execution Unit — the Ebox

The Ebox executes those Alpha instructions that do not reference memory and are not floating point. The Ebox contains multiple copies of its various processing elements, allowing the Qbox to schedule as many as eight instructions per cycle.

Table 2–5 lists the Ebox major components.

Table 2–5 Ebox Major Component Summary

Component	Description
Integer Units (8)	The integer functional units execute the traditional integer arithmetic and logical instructions as well as performing the address generation and data formatting of memory instructions.
Multimedia Units (4)	The multimedia units execute the newer integer instructions targeted at accelerating multimedia operations and also perform integer multiplication.
Register Caches (4)	The register caches store recently written register values allowing dependent instructions to issue before the register file is updated.

Structurally, the Ebox processing elements are organized into eight functional units, each of which executes a predefined subset of the instruction set, as listed in Table 2–6. Each integer functional unit is a logical collection of processing elements that collectively execute a specific set of Alpha instructions, and each functional unit is organized as four clusters of two units each.

Table 2–6 Ebox Cluster Section Summary

Section Name	In Units	Description
Adder	0-7	A full 64-bit signed integer adder that produces a complete result each cycle. Services the following instructions: **Type** — **Instructions** Add — ADDL, ADDL/V, ADDQ, ADDQ/V, S4ADDL, S8ADDL, S4ADDQ, S8ADDQ Sub — SUBL, SUBL/V, SUBQ, SUBQ/V, S4SUBL, S8SUBL, S4SUBQ, S8SUBQ Compare — CMPBGE, CMPULT, CMPEQ, CMPULE, CMPLT, CMPLE Other — LDAH, LDA, RS, RC
Cross Cluster Result Interface	0-7	Receives one-cycle results from the other functional units, bypasses the data onto the operand busses if immediately needed, and latches the data for writing into the local register cache.
Global Control	0-7	Decodes the instruction information sent by the Qbox and coordinates the various processing elements within a functional unit.
Load Data Interface	4-7	Interfaces the data returned from the Mbox to the functional units and register caches. Services the following instructions: **Type** — **Instructions** Load — LDL, LDQ, LDQ_U, LDL_L, LDQ_L, LDBU, LDWU, LDG, LDS, LDT, LDF Special — HW_LD, STx_C

Table 2–6 Ebox Cluster Section Summary (Continued)

Section Name	In Units	Description
Logic Box	0-7	Performs logical and arithmetic operations. Services the following instructions: **Type** — **Instructions** Cmove — CMOVLBS, CMOVLBC, CMOVNE, CMOVLT, CMOVGE, CMOVLE, CMOVGT Branch — BLBC, BEQ, BLT, BLE, BLBS, BNE, BGE, BGT Logical — AND, BIC, BIS, ORNOT, XOR, EQV Special — AMASK, IMPLVER, SEXTB, SEXTW
Multimedia Operand Interface	4-7	Forwards the instruction operands from the corresponding integer functional unit to the multimedia clusters. Each multimedia cluster is associated with the lower integer functional unit in a cluster and derives its operands from that functional unit. Services the following instructions: **Type** — **Instructions** Multiply — MULL, MULL/V, MULQ, MULQ/V, UMULH Multimedia — Opcode 1C.XX, except SEXTB, SEXTW Store — STL, STQ, STQ_U, STL_C, STQ_C, STB, STW, STG, STS, STT, STF Special — ITOFF, ITOFS, ITOFT, HW_ST
Register File Operand Interface	0-7	Interfaces the operands from the register file to the Ebox opbusses. Also bypasses literals onto the opbusses.
Register File Result Pipe	0-3	Handles staging of different result latencies, floating-point load format conversion and forwarding of results to the register file.

Table 2–6 Ebox Cluster Section Summary (Continued)

Section Name	In Units	Description
Shifter	0-3	A full 64-bit shifter that produces a complete result each cycle. Services the following instructions:
		Type — **Instructions**
		Shift — SRL, SLL, SRA
		Mask — MSKBL, MSKWL, MSKLL, MSKQL, MSKWH, MSKLH, MSKQH
		Extract — EXTBL, EXTWL, EXTLL, EXTQL, EXTWH, EXTLH, EXTQH
		Insert — INSBL, INSWL, INSLL, INSQL, INSWH, INSLH, INSQH
		Zap — ZAP, ZAPNOT
Store Data Interface	—	Interfaces to the store data buses (to the Mbox). This unit is not actually part of the integer clusters but resides in a separate partition to the right of the integer clusters. Services the following instructions:
		Type — **Instructions**
		Store — STL, STQ, STQ_U, STL_C, STQ_C, STB, STW, STG, STS, STT, STF
		Special — ITOFS, ITOFF, ITOFT, FTOIS, FTOIT
Virtual Address Generator	4-7	Computes the 16-bit displacement add and factors the big/little endian control to form a correct virtual memory address. Services the following instructions:
		Type — **Instructions**
		Load — LDL, LDQ, LDQ_U, LDL_L, LDQ_L, LDBU, LDWU, LDG, LDS, LDT, LDF
		Store — STL, STQ, STQ_U, STL_C, STQ_C, STB, STW, STG, STS, STT, STF
		Jump — JMP, JSR, RET, JSR_COROUTINE
		Special — TRAPB, EXCB, MB, WMB, ECB, FETCH, FETCH_M, WH64, HW_LD, HW_ST, HW_MTPR, LDx_ARM, QUIESCE

2.5.3 Floating-Point Instruction Execution Unit — the Fbox

The Fbox executes all current Alpha floating-point instructions and the new paired single-precision instructions. The Fbox receives instructions from the Qbox, by way of the Ebox, and receives operands from the register file, the load data buses (up to three), or its own register caches. The Fbox returns floating-point results to the Register File and floating-point store data to the Mbox, again by way of the Ebox. The Fbox returns exception information to the Qbox.

Table 2–7 lists the Fbox major components.

Table 2–7 Fbox Major Component Summary

Component	Description
Floating-point control register (FPCR)	Contains rounding information and trap disable bits used by the floating-point operate instructions, and exception status information from floating-point operate instructions. The FPCR is read from and written to the floating-point registers by the MF_FPCR and MT_FPCR instructions. In addition, all operate instructions use the dynamic rounding mode bits to round the results and the trap disable bits to signal traps when an exception is detected.
Interface control (F_INT)	Performs a partial decode of the opcode, function code, and thread processor unit (TPU) to determine if a valid floating-point instruction has been issued. The F_INT also contains logic that allows direct access to internal operand buses from Register File operand buses, and logic to dispatch floating-point store data to the Ebox from either the result data of pipelines F_P0 and F_P1, or from the register cache.
Operand steering unit (F_OSU)	Performs comparisons against incoming physical register (Preg) numbers to determine the source of input operands to the Fbox pipelines.
Pipeline Clusters (F_P*n*)	The Fbox is organized as four identical clusters, each cluster consisting of one execution pipeline. The four pipelines, F_P0 through F_P3, allow up to four floating-point operate instructions to be issued at each cycle. Two copies of a register cache, one for each set of two pipelines, are included to allow the results of recently completed instructions to be used with minimal delay. Each pipeline contains the functional units needed to execute the various floating-point instructions.
Register cache (F_RGC)	Contains staging logic and static RAM that latch and hold recently generated result data of the Fbox pipelines as well as copies of incoming floating-point loads. The result data is eventually dispatched to the Register File. However, this result and load data can be used in subsequent floating-point operations without incurring the transit time delay in returning data from the Register File

2.5.3.1 Functional Units

Table 2–8 lists the instructions that are executed by each functional unit in the

Fbox.

Table 2–8 Fbox Functional Unit Summary

Functional Unit	Instructions
Add pipe 1 : F_AP1	ADD,SUB,CMP
Add pipe 2 : F_AP2	ADD/SUB (align>1), CVTff, CVTfq, CVTqf, CVTql, CVTlq
Divider : F_DIV	DIV[1]
Graphics ADD : F_GAD	Paired single-precision except PMUL, PARCPL, and PARSQRT
Graphics MUL : F_GML	Paired single-precision MUL type instructions: PMUL, PARCPL, PARSQRT
Mull Unit : F_MUL	MUL
Short pipe : F_SHP	CPYSx, FCMOV, FBxx
	Special operands (Zeros, Denormal OPD, NANs, INF,RES.OPD),INPUT EXCEPTIONS, Mx_FPCR
Square root : F_SQR	SQRT[1]

[1] See Section 2.4.3 for instruction issue rules regarding the DIV and SQRT instructions.

2.6 Memory Controller Unit — the Mbox

The Mbox executes Alpha memory access instructions, including integer and floating-point load and store, memory barrier, prefetch, write-hint, load-locked, and store-conditional.

The Mbox can process up to four instructions per cycle, out of order. At each cycle, the Mbox can accept as many as three load instructions and as many as two store instructions, for a maximum of four operations. The Mbox is solely responsible for tracking memory reference instructions that have issued but not retired, and for ensuring that the final effect of memory reference instructions is equivalent to sequential execution of the thread, within the Alpha SRM definition of equivalence. The Mbox also receives fill data from the Cbox and, to maintain cache coherence, processes probes that the Cbox receives from the rest of the system.

There are two data input busses, each of which is associated with a store port.

The Mbox has four instruction ports to handle loads, stores, and prefetches. The Mbox can return data on three of those ports, so the Mbox can accept a maximum of three loads issued per cycle.

Of the four ports:

- Two can perform loads and prefetches
- One can perform loads, stores and prefetches
- One can perform only stores

Table 2–9 lists the Mbox major components.

Table 2–9 Mbox Major Component Summary

Component	Description
Dcache	64KB of data storage, with a write-allocate, write-through write-policy
Dtags	1K entries of tag storage, arranged as 2-way set-associative with 4 read ports and 1 write port
Load Queue	64-entry queue that holds issued, but not-retired load addresses. Handles load ordering traps and re-issuing of loads
Merge Buffer	16-entry buffer that accumulates Store data before writing it into the Dcache and Cbox
Pre-MAF	16-entry buffer that holds the addresses of loads that have missed in the Dcache and need further activity in the Cbox.
Store Queue	64-entry queue that holds store addresses & data before stores have retired. Used to satisfy load requests to addresses with uncompleted stores
Translation Buffers	128-entry, fully-associative with 4 read ports to perform the virtual-to-physical address transactions

2.7 External Interface

The responsibilites of the external interface unit include:

- Resolve misses in the Icache and Dcache, either in the Scache, local memory, or remote memory.
- Ensure that data written by the processor is made visible coherently to other processors and I/O nodes.
- Communicate with other nodes in a multiprocessor configuration so that the total memory space can be shared.
- Control Rambus memories to provide physical memory to the multiprocessor.
- Implement a coherence protocol that ensures that all processors have a consistent image of memory.
- Accept and prioritize interrupt requests, delivering thread-specific requests to the Qbox.

The external interface unit consists of three major subsections that work together, in conjunction with the cache coherency protocol, to present a distributed, shared, coherent, cached, multiprocessor memory (CC-NUMA) to the 21464 core.

2.7.1 Scache Controller — the Cbox

The Cbox controls the second-level cache (Scache). In particular, the Cbox controls:

- Requests for cache blocks from the Ibox and Mbox
- Write-through from the Mbox
- Fills and displaced victims
- Probes from the system

The Cbox contains the following major components:

Table 2–10 Cbox Major Component Summary

Component	Description
Miss address file (MAF)	Holds requests from the processor whilst being processed.
Victim address file (VAF) Victim data buffer (VDB)	Hold blocks being sent back to the system either as displacement victims or in response to system probes.
Probe address file (PAF)	Holds probes waiting to be processed.

2.7.2 Router — the Rbox

The Rbox provides the interprocessor switch — the communication fabric by which 21464 processors are interconnected to form glueless multiprocessor systems. The Rbox interfaces the local processor and memory to I/O controllers, all other processors, and their associated memories, through five bidirectional ports.

The Rbox includes the following physical components:

- Port input queues — packets received from interface but not yet transferred to an output queue
- Port output queues — packets waiting to be transferred to a connected processor
- Routing tables — translate destination node number or mask into output port selection and virtual channel
- Arbitration — selects among port input queues for transfer to output queues

2.7.3 Rambus Interface — the Zbox

The Zbox provides a glueless interface to two independent interleaved arrays of Rambus memories for processor's main memory, including cache-coherence directory. Each array consists of four busses, each accessing up to 32 DRAM chips.

The Zbox includes the following physical components:

- Rambus queues and sequencer — controls attached Rambus memories for read and write operations. Includes scheduling table and page status.
- Directory management and coherence protocol state machine.
- Directory in flight table— DIFT records requests to the local memory that cannot complete immediately because required data is "in-flight" somewhere in the system.

2.7.4 Cache Coherency Protocol

The 21464 adopts the 21364 cache coherence protocol with small enhancements. The protocol is a directory based CC-NUMA and tolerates out-of-order channels except for the I/O channel, thereby supporting an adaptive packet routing.

2.7.4.1 Introduction to the Protocol

The coherence protocol is the mechanism that lets numerous processors maintain a consistent image of the contents of memory, as required by the Alpha SRM.

The 21464 increases reliability and load-distribution by using multiple resources for enforcing cache coherence. Further, the 21464 uses nondeterministic routing, which makes the best use of available network resources. Such routing allows two messages to take different paths and get out of order, even if they start and end at the same nodes.

The protocol is designed to ensure that all processors that cause and/or observe changes in memory, see those changes occur in the same apparent order, even though the messages between processors and memories may get out of order. The order observed by all processors is the order in which requests are serviced in their home memory and, in particular, in the Mbox directory in-flight table (the DIFT). Caches communicate with the DIFT as they manipulate memory data, and the DIFT delays multiple requests for any individual block until it has coordinated previous requests with any caches affected by those requests.

The protocol, as managed by the DIFT, is concerned with the transitions between states, and with performing the transitions in such a way that as much of the communication latency as possible is kept out of the critical paths.

The memory system is designed with the expectation that a disproportionate fraction of the memory traffic produced by any processor is addressed to its own local memory; this is true for most multiprocessor applications, though precisely how much is highly application-dependent. The protocol uses this fact, and the onchip communication between a cache and its local controller, to optimize references to the local memory. The Dcache optimizes the directory accesses for requests from local and remote processors. The onchip Dcache stores the directory information of most frequently used cache blocks to minimize memory accesses for directory information. The Dcache is updated by requests from the local Cbox and remote processors, thereby eliminating the need for the LPR.

2.7.4.2 Structures that Maintain the Cache Coherence

Cache coherence is maintained by using the following structures:

- Miss address file (MAF)
- System request pending queue (SRQ)
- Victim buffer
  - Victim address file (VAF)
  - Victim data buffer (VDB)
- Probe queue (PRQ): probe queue
- Directory in-flight table (DIFT)

2.8 Pipeline Organization

The pipeline is organized as follows.

2.8.1 Pipeline Diagram

Figure 2–2 shows the 21464 pipeline stages. Note the following symbol meanings in Figure 2–2:

Symbol	Meaning
V	Exception funnel timing
V8	The cycle at which an exception kill is driven onto the Retire/Kill Bus. Its position in this diagram is relative to the first good path instruction block after the exception kill is posted on the Retire/ Kill bus.
CMP1	Completion of instructions issued from the 4 main computation pipes and caused no exceptions.
RET1	The earliest retire cycle (Retire Bus cycle) of instructions completed in CMP1.
CMP2	Completion of instructions issued from the 4 main computation pipes which may cause an exception (including all the floating-point instructions).
RET2	The earliest retire cycle (Retire Bus cycle) of instructions completed in CMP2. This is also the V3 timing of a Retire Time Exception.
CMP3	Completion of instructions issued from the 4 memory pipes.
RET3	The earliest retire cycle (Retire Bus cycle) of instructions completed in CMP3. This is also the V3 timing of a Retire Time Exception.

In Figure 2–2, alphabetic characters that follow the box letter (such as the W in the second row's PW) signify negative integers and are defined in Table 2–11.

Figure 2–2 21464 Pipeline Stage Diagram

		FETCH		MAP				ISSUE											CMP1		RFW	CMP2	RET1		CMP3	RET2		RET3
I0	**I1**	**I2**	**I3**	**I4**	I5	I6	I7	I8	I9	I10	I11	I12	I13	I14	I15	I16	I17	I18	I19	I20	I21	I22	I23	I24	I25	I26	I27
PW	PX	PY	PZ	**P0**	**P1**	**P2**	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	P21	P22	P23
QU	QV	QW	QX	QY	QZ	**Q0**	**Q1**	**Q2**	**Q3**	**Q4**	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	**Q14**	**Q15**	Q16	**Q17**	**Q18**	Q19	Q20	**Q21**	P23
RQ	RR	RS	RT	RU	RV	RW	RX	RY	RZ	**R0**	**R1**	**R2**	**R3**	**R4**	R5	R6	R7	R8	**R9**	R10	R11	R12	R13	R14	R15	R16	R17	
EM	EN	EO	EP	EQ	ER	ES	ET	EU	EV	EW	EX	EY	EZ	**E0**	**E1**	**E2**	**E3**	**E4**	E5	E6	E7	E8	E9	E10	E11	E12	E13	
FM	FN	FO	FP	FQ	FR	FS	FT	FU	FV	FW	FX	FY	FZ	**F0**	**F1**	**F2**	**F3**	**F4**	F5	F6	F7	F8	F9	F10	F11	F12	F13	
ML	MM	MN	MO	MP	MQ	MR	MS	MT	MU	MV	MW	MX	MY	MZ	**M0**	**M1**	**M2**	**M3**	M4	M5	M6	M7	M8	M9	M10	M11	M12	
CI	CJ	CK	CL	CM	CN	CO	CP	CQ	CR	CS	CT	CU	CV	CW	CX	CY	CZ	**C0**	**C1**	**C2**	**C3**	C4	C5	C6	C7	C8	C9	
V6	V7	**V8**	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	V21	V22	V23	V24	V25	V26	V27			**V3**			**V3**	

2.8.2 Conversion Between Negative Integer and Alphabet

Table 2–11 shows the conversion between negative integers and the alphabet.

Table 2–11 Negative Integers to Alphabetics Conversion

-26	-25	-24	-23	-22	-21	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

2.8.3 Basic Pipeline Stage Conversion Equations

The basic pipeline stage conversion equations are as follows. The conversions are tabulated in Table 2–13.

Table 2–12 Pipeline Stage Conversion Equations

To	From
I	P+4
P	Q+2
Q	R+4
R	E+4
E	F+0
F	M+1
M	C+3

2.8.4 Conversion Table

As listed in Table 2–12, from box *X*<a> to box *Y*<b>, <b>=<a>+<intersection of boxes *X* and *Y*>. The intersection in Table 2–13 is bolded.

Table 2–13 Pipeline Stage Conversion

To ↓	From →								
	I	**P**	**Q**	**R**	**E**	**F**	**M**	**C**	**V**
I		**+4**	+6	+10	+14	+14	+15	+18	-6
P	-4		**+2**	+6	+10	+10	+11	+14	-10
Q	-6	-2		**+4**	+8	+8	+9	+12	-12
R	-10	-6	-4		**+4**	+4	+5	+8	-16
E	-14	-10	-8	-4		**+0**	+1	+4	-20
F	-14	-10	-8	-4		-0	**+1**	+4	-20
M	-15	-11	-9	-5	-1	-1		**+3**	-21
C	-18	-14	-12	-8	-4	-4	-3		**-24**
V	+6	+10	+12	+16	+20	+20	+21	+24	

2.9 Instruction Execution Pipelines and Latency

Instruction Latency

Defines the parent-to-child issue latency. Also identifies any cross-pipeline delay associated with broadcasting the parents results to other pipelines. Instructions that are not pipelined are also identified as "bubbling" for completion. Latency is shown in Table 2–14 in the following formats:

Format	Meaning
n	*N* cycle latency to a child in any pipeline
m+*n*	*M* cycle latency plus extra *n* cycle to other pipelines.
n+B	*N* cycle latency non-pipelined, requires bubble (B) to signal completion.

Execution Pipelines

In Table 2–14, the pipelines column identifies those of the eight pipelines in which the instruction can execute. The actual slotting algorithm is a function of the types and positions of the instructions in each map block. Details about instruction slotting can be found in Section 2.10. Because an instruction is slotted to a particular pipeline does not mean it must execute there. Follow-me capabilities in the Qbox allow instructions for which operands are data-ready in another allowed pipeline in the same half of the queue to issue from that pipeline. Pipelines 0, 2, 5 and 7 are in one-half of the queue, pipes 1, 3, 4, 6 are in the other half.

Pipelines are described in Table 2–14 in the following formats:

Format	Meaning
0 – 7	Can execute in any pipe
0 – 3	Can execute in pipes 0, 1, 2, or 3.
0 , 3	Can execute in only pipes 0 or 3
0 ~ 1	Can execute in only pipes 0 or 1 and not both in the same cycle.
Alt 0 – 3	Can execute in pipes 0, 1, 2, or 3, but does not issue to the same pipe in consecutive cycles

Table 2–14 Instruction Execution Pipelines and Latency

	Mnemonic	Pipelines	Latency		Mnemonic	Pipelines	Latency
PALcode (Opcodes as follows:)							
00	CALL_PAL	0 ~ 1	5	1D	HW_MTPR	6 , 7 0~1	1 / 3[1] 1[1]
1B	HW_LD	6 , 7	3	1F	HW_ST	4 , 5	—
19	HW_MFPR	0~1	5	1E	IFETCHB	4 , 5	—
Add/Subtract/Compare (Opcode 10)							
	ADDL	0–7	1 + 1		S4ADDQ	0–7	1 + 1
	ADDQ	0–7	1 + 1		S4SUBL	0–7	1 + 1
	CMPBGE	0–7	1 + 1		S4SUBQ	0–7	1 + 1

Table 2–14 Instruction Execution Pipelines and Latency

Mnemonic	Pipelines	Latency	Mnemonic	Pipelines	Latency
CMPEQ	0–7	1 + 1	S8ADDL	0–7	1 + 1
CMPLE	0–7	1 + 1	S8ADDQ	0–7	1 + 1
CMPLT	0–7	1 + 1	S8SUBL	0–7	1 + 1
CMPULE	0–7	1 + 1	S8SUBQ	0–7	1 + 1
CMPULT	0–7	1 + 1	SUBL	0–7	1 + 1
S4ADDL	0–7	1 + 1	SUBQ	0–7	1 + 1
Integer Logical (Opcode 11)					
AMASK	0–7	1 + 1	CMOVLE	0–7	1 + 1
AND	0–7	1 + 1	CMOVLT	0–7	1 + 1
BIC	0–7	1 + 1	CMOVNE	0–7	1 + 1
BIS	0–7	1 + 1	CMOV2	0–7	1 + 1
CMOVEQ	0–7	1 + 1	EQV	0–7	1 + 1
CMOVGE	0–7	1 + 1	INOP	0–7	1 + 1
CMOVGT	0–7	1 + 1	ORNOT	0–7	1 + 1
CMOVLBC	0–7	1 + 1	XOR	0–7	1 + 1
CMOVLBS	0–7	1 + 1			
Integer Shift (Opcode 12)					
extbh[2]	0–3	1 + 1	INSWH	0–3	1 + 1
EXTBL	0–3	1 + 1	INSWL	0–3	1 + 1
EXTLH	0–3	1 + 1	mskbh[2]	0–3	1 + 1
EXTLL	0–3	1 + 1	MSKBL	0–3	1 + 1
EXTQH	0–3	1 + 1	MSKLH	0–3	1 + 1
EXTQL	0–3	1 + 1	MSKLL	0–3	1 + 1
EXTWH	0–3	1 + 1	MSKQL	0–3	1 + 1
EXTWL	0–3	1 + 1	MSKWH	0–3	1 + 1
insbh [2]	0–3	1 + 1	MSKWL	0–3	1 + 1
INSBL	0–3	1 + 1	SLL	0–3	1 + 1
INSLH	0–3	1 + 1	SRA	0–3	1 + 1
INSLL	0–3	1 + 1	SRL	0–3	1 + 1
INSQH	0–3	1 + 1	ZAP	0–3	1 + 1
INSQL	0–3	1 + 1	ZAPNOT	0–3	1 + 1

Table 2–14 Instruction Execution Pipelines and Latency

Mnemonic	Pipelines	Latency	Mnemonic	Pipelines	Latency
Integer Multiply (Opcode 13)					
MULL	4–5	5	UMULH	4–5	5
MULQ	4–5	5			
Integer to Floating Register Transfer (Opcode 14)					
ITOFF	6 , 7	5	SQRTG	Alt 0–3	18+1
ITOFS	6 , 7	5	SQRTS	Alt 0–3	33+1
ITOFT	6 , 7	5	SQRTT	Alt 0–3	33+1
SQRTF	Alt 0–3	18+1			
VAX Floating-Point (Opcode 15)					
ADDF	0-3	3 + 1	CVTQF	0-3	3 + 1
ADDG	0-3	3 + 1	CVTQG	0-3	3 + 1
CMPGEQ	0-3	3 + 1	DIVF	Alt 0-3	9 + B + 1
CMPGLE	0-3	3 + 1	DIVG	Alt 0-3	13 + B + 1
CMPGLT	0-3	3 + 1	MULF	0-3	3 + 1
CVTDG	0-3	3 + 1	MULG	0-3	3 + 1
CVTGD	0-3	3 + 1	SUBF	0-3	3 + 1
CVTGF	0-3	3 + 1	SUBG	0-3	3 + 1
CVTGQ	0-3	3 + 1			
IEEE Floating-Point (Opcode 16)					
ADDS	0-3	3 + 1	CVTTQ	0-3	3 + 1
ADDT	0-3	3 + 1	CVTTS	0-3	3 + 1
CMPTEQ	0-3	3 + 1	DIVS	Alt 0-3	9 + B + 1
CMPTLE	0-3	3 + 1	DIVT	Alt 0-3	13 + B + 1
CMPTLT	0-3	3 + 1	MULS	0-3	3 + 1
CMPTUN	0-3	3 + 1	MULT	0-3	3 + 1
CVTQS	0-3	3 + 1	SUBS	0-3	3 + 1
CVTQT	0-3	3 + 1	SUBT	0-3	3 + 1
Miscellaneous Floating-Point (Opcode 17)					
CPYS	0-3	1 + 1	FCMOVGE	0-3	1 + 1
CPYSE	0-3	1 + 1	FCMOVGT	0-3	1 + 1

Table 2–14 Instruction Execution Pipelines and Latency

Mnemonic	Pipelines	Latency	Mnemonic	Pipelines	Latency
CPYSN	0-3	1 + 1	FCMOVLE	0-3	1 + 1
CVTLQ	0-3	3 + 1	FCMOVLT	0-3	1 + 1
CVTQL	0-3	3 + 1	FCMOVNE	0-3	1 + 1
FCMOV2	0-3	1 + 1	MF_FPCR	0,3	3 + 1
FCMOVEQ	0-3	1 + 1	MT_FPCR	0,3	—
Miscellaneous (Opcode 18)					
CCB	4 , 5		QUIESCE	4 , 5	
ECB	4 , 5		RC	4 , 5	1 + 1
EXCB			RPCC	0~1	5
FETCH_M[3]			RS	4 , 5	1 + 1
FETCH[3]			TRAPB		
LDL_ARM	6 , 7	3	WH64	4 , 5	
LDQ_ARM	6 , 7	3	WH64EN[4]	4 , 5	
MB[5]			WMB	4 , 5	
Multimedia (Opcode 1C)					
CMPLGE	2 , 3	5	TSQERRzzz	2 , 3	5
CMPWGE	2 , 3	5	TSUBzzz	2 , 3	5
CTLZ	2 , 3	5	UNPKBL	0 , 1	5
CTPOP	2 , 3	5	UNPKBW	0 , 1	5
CTTZ	2 , 3	5	UPKSBW4	0 , 1	5
FTOIS	4 , 5	5	UPKSWL2	0 , 1	5
FTOIT	4 , 5	5	UPKUBW4	0 , 1	5
GPKBLB4	0 , 1	5	UPKUWL2	0 , 1	5
MAXzzz	2 , 3	5	VADDSL2	2 , 3	5
MINSB8	2 , 3	5	VADDUL2	2 , 3	5
MINSW4	2 , 3	5	VADDzzz	2 , 3	5
MINUB8	2 , 3	5	VMINMAXSL2	2 , 3	5
MINUW4	2 , 3	5	VMINMAXUL2	2 , 3	5
PERMB8	0 , 1	5	VMINMAXzzz	2 , 3	5
PERR	2 , 3	5	VMULHUW4	2 , 3	5
PKLB	0 , 1	5	VMULLUW4	2 , 3	5
PKSLW4	0 , 1	5	VSLB8	0 , 1	5

Table 2–14 Instruction Execution Pipelines and Latency

	Mnemonic	Pipelines	Latency		Mnemonic	Pipelines	Latency
	PKSWB8	0, 1	5		VSLL2	0, 1	5
	PKULW4	0, 1	5		VSLW4	0, 1	5
	PKUWB8	0, 1	5		VSRAB8	0, 1	5
	PKWB	0, 1	5		VSRAL2	0, 1	5
	SEXTB	0–7	1 + 1		VSRAW4	0, 1	5
	SEXTW	0–7	1 + 1		VSRB8	0, 1	5
	TABSERRzzz	2, 3	5		VSRL2	0, 1	5
	TADDzzz	2, 3	5		VSRW4	0, 1	5
	TMULUSB8	2, 3	5		VSUBSL2	2, 3	5
	TMULUSW4	2, 3	5		VSUBUL2	2, 3	5
	TMULzzz	2, 3	5		VSUBzzz	2, 3	5
Load and Store (Opcodes as follows:)							
08	LDA	0–7	1 + 1	26	STS	4, 5	3[6]
09	LDAH	0–7	1 + 1	27	STT	4, 5	3[6]
0A	LDBU	6, 7	3	28	LDL	6, 7	3
0B	LDQ_U	6, 7	3	29	LDQ	6, 7	3
0C	LDWU	6, 7	3	2A	LDL_L	6, 7	3
0D	STW	4–5	3[6]	2B	LDQ_L	6, 7	3
0E	STB	4–5	3[6]	2C	STL	4–5	3[6]
0F	STQ_U	4.5	3[6]	2D	STQ	4–5	3[6]
20	LDF	6, 7	5	2E	STL_C	4–5[7]	3
21	LDG	6, 7	5	2F	STQ_C	4–5[7]	3
22	LDS	6, 7	5				
23	LDT	6, 7	5				
24	STF	4 – 5	3[6]				
25	STG	4 – 5	3[6]				
Branch and Jump (Opcodes as follows:)							
1A.0	JMP	0~1	5	36	FBGE	0, 1	—
1A.1	JSR	0~1	5	37	FBGT	0, 1	—
1A.2	RET	0~1	5	38	BLBC	0–7	—
1A.3	JSR_CO	0~1	5	39	BEQ	0–7	—
30	BR	0~1	5	3A	BLT	0–7	—

Table 2–14 Instruction Execution Pipelines and Latency

	Mnemonic	Pipelines	Latency		Mnemonic	Pipelines	Latency
31	FBEQ	0, 1	—	3B	BLE	0–7	—
32	FBLT	0, 1	—	3C	BLBS	0 – 7	—
33	FBLE	0, 1	—	3D	BNE	0 – 7	—
34	BSR	0~1	5	3E	BGE	0 – 7	—
35	FBNE	0 – 3	—	3F	BGT	0 – 7	—

[1] HW_MTPR instructions can specify a writer class to create an issue dependency to future HW_MxPR instructions. HW_MxPR instructions that indentify a reader class dependency are scheduled to issue no earlier than 1 cycle after the HW_MTPR instruction that wrote the class dependency. HW_MTPR instructions can also specify writer class dependencies that are satisfied on completion, rather than issue. HW_MxPR instructions that identify a reader class dependency against this type of writer class are scheduled to issue no earlier than three cycles after the issue of the completion bubble signal to the writer. The 21464 only allows specifying completion dependencies against HW_MTPR instructions that target the Mbox; those that target the Ibox are ignored.

[2] The mskbh, insbh and extbh decodes are not formally defined by the Alpha SRM because all combinations of inputs produce a zero result. The generalized decoding in the 21464 Integer Shifter does not special case these code points and produces a zero result.

[3] FETCHx instructions never actually issue from the Qbox but are completed immediately and therefore act as NOPs.

[4] The WH64EN instruction is currently proposed as ECO#127 to the Alpha SRM.

[5] MB instructions never formally issue from the Qbox but are instead sent to the Mbox as soon as they enter the Qbox. MB instructions do not complete until the Mbox notifies the Qbox that the necessary conditions have been met.

[6] Although store instructions do not produce a register result and therefore do not have normal dependents, the Ibox store-set logic can create dependency groups of loads and stores. A load that is a store-set dependent on a store instruction has an effective issue latency of three cycles from the issue of the store.

[7] Store conditional instructions issue as stores to pipelines 4 and 5, but bubble back completion to the Qbox. Final completion of the STx_C instruction appears on the load pipes 6 and 7.

2.10 Instruction Issue and Retire Rules

2.10.1 Issue Rules

In order to issue from the Qbox instruction queue (the IQ), instructions must bid in, and be granted by, a picker. Slotting determines each instruction's "preferred pipe", i.e. in which picker it may bid. Each cycle, the oldest bidding instruction in a picker is granted. Only instructions that are bidding in a given cycle are candidates for grants.

2.10.1.1 Bidding Rules

The following rules apply to initial issue; there are additional qualifications for instructions that must re-issue.

General Bids

In general, instructions may bid when all of their source operands are result-ready. Additional conditions apply in the following cases.

- Stores and loads that are slotted for a load picker may bid if the following are true:
  - They are result-ready
  - They are below their high-water mark (which signifies that the Mbox has sufficient resources to exectute them)
  - They are not dependent on a DTB writer block
- Loads must satisfy any store-set dependencies prior to being enabled to bid.
- All loads and stores are speculatively assumed to be below their high-water mark when they first allocate into the IQ; their actual status is available one cycle later. Any loads or stores that are granted as the result of a bid that was based on false high-water mark speculation are retracted and do not issue from the IQ.
- Jumps (JMP, JSR, JSR_COROUTINE, RET), direct branches (BR, BSR), RPCC, CALL_PAL, HW_MTPR/HW_MTPR for Ibox IPRs, and HW_LD/WrChk may only bid if they are result-ready and their slotted picker is load-enabled in the current cycle.
- Because floating-point divide and square root instructions are not pipelined operations, they must not issue from the same picker on subsequent cycles and are thus enabled to bid only on every other cycle.

  Unfortunately, the IQ logic does not have time to disable bids for an FDIV or FSQRT functional unit for which an instruction has been been granted on the immediately preceding cycle. Therefore, the 21464 globally disables all FDIV and FSQRT bids every other cycle to give the IQ time to determine exactly which instructions may safely bid.
- Instructions expressly identified as NOPs do not bid or issue but are allocated into the IQ as invalid (i.e. empty) entries.
- MB instructions do not issue from the IQ but are subject to special retire conditions as described in Section 2.10.2.
- Instructions stop bidding if they are killed. Instructions that are killed after being granted, but before being issued from the IQ, do not issue.

Follow Me Bids:

Instructions that have a cross cluster delay become "locally" result-ready in the cluster in which their result is produced one cycle earlier than they become "globally" result-ready in the rest of the IQ. Instructions that are locally result-ready, and meet all other bid criteria, may bid in the relevant picker for that one-cycle window, even if it is not their preferred picker. This is known as a "follow me" bid, since the dependent instructions follows their parent into a cluster.

Instructions are only enabled to make a follow me bid in pickers from which they may actually issue — in other words, they must be of a type supported by the functional units serviced by the picker.

2.10.2 Retirement Rules

An instruction is eligible to retire if it is complete and all older, unretired instructions within its TPU are also complete. The Qbox Completion Unit (CMP) retires instructions one INum block at a time, but signals retire eligibility to the Retire/Kill Bus on as fine as a per-instruction granularity (see the Completion Unit descripton for more details).

2.10.2.1 Completion Rules

In general, instructions that have passed their poison point and their trap point — that is, the last point in time when they can cause a disruption — are completed, with the following exceptions.

- Some memory instructions pass their trap point very late in the pipeline and are therefore speculatively completed and subsequently uncompleted when any disruption information becomes available.
- Instructions identified as NOPs complete immediately upon allocation into the IQ.
- Killed instructions are automatically completed.
- MB and STx_C instructions are completed only when the Mbox indicates to the CMP that it may do so.
- The Mbox flags I/O operations for the CMP. I/O operations may complete normally, but the CMP may not retire any block containing them until the Mbox signals that this is permitted.
- There is a facility to drain the Completion Unit pipeline in the event of an external probe, in order in insure consistency between TPUs and/or CPUs.

Note that the time interval between an instruction's issue and completion depends on the particular picker from which the instruction issues. Instructions issuing on the four primary ALU pickers have a faster completion path than the others.

2.11 Implementation-Specific Architecture Features

2.11.1 New Instructions

2.11.1.1 Thread Synchronization

Using a multithreading architecture, the 21464 implements three new instructions that enhance the performance of multithread processing.

Table 2–15 Thread Synchonization Instructions

Mnemonic	Operation
LDL_ARM	Load Longword and Arm the Watch Register
LDQ_ARM	Load Quadword and Arm the Watch Register
Quiesce	Wait on Access to the Watch Register

2.11.1.2 Short Vector SIMD (Single Instruction Stream, Multiple Data Streams)

The short vector SIMD instructions provide a complete set of vectorized integer operations for multimedia and signal processing applications. They allow the processing of multiple elements in each machine cycle by vectoring smaller data types that are packed into a quadword.

Table 2–16 Short Vector SIMD Instructions

Mnemonic	Operation
Tree Operations	
TABSERRSB8	Tree Absolute Error Byte
TABSERRSW4	Tree Absolute Error Word
TABSERRUB8	Unsigned Tree Absolute Error Byte
TABSERRUW4	Unsigned Tree Absolute Error Word
TADDSB8	Tree Add Byte
TADDSW4	Tree Add Word
TADDUB8	Unsigned Tree Add Byte
TADDUW4	Unsigned Tree Add Word
TMULSB8	Tree Multiply Byte
TMULSW4	Tree Multiply Word
TMULUB8	Unsigned Times Unsigned Tree Multiply Byte
TMULUSB8	Unsigned Times Signed Tree Multiply Byte
TMULUSW4	Unsigned Times Signed Tree Multiply Word
TMULUW4	Unsigned Times Unsigned Tree Multiply Word
TSQERRSB8	Tree Squared Error Byte
TSQERRSW4	Tree Squared Error Word
TSQERRUB8	Unsigned Tree Squared Error Byte
TSQERRUW4	Unsigned Tree Squared Error Word
TSUBSB8	Tree Subtract Byte
TSUBSW4	Tree Subtract Word
TSUBUB8	Unsigned Tree Subtract Byte
TSUBUW4	Unsigned Tree Subtract Word
Vector Operations	
CMPLGE	Compare LongWord
CMPWGE	Compare Word
GPKBLB4	Graphics Pack Byte
PERMB8	Permute Bytes
PKSLW4	Pack Signed Longwords to Words

Table 2–16 Short Vector SIMD Instructions

Mnemonic	Operation
PKSWB8	Pack Signed Words to Bytes
PKULW4	Pack Unsigned Longwords to Words
PKUWB8	Pack Unsigned Words to Bytes
UPKSBW4	Unpack Signed Bytes to Words
UPKSWL2	Unpack Signed Words to Longwords
UPKUBW4	Unpack Unsigned Bytes to Words
UPKUWL2	Unpack Unsigned Words to Longwords
VADDSB8	Parallel Add Byte
VADDSL2	Parallel Add Longword
VADDSW4	Parallel Add Word
VADDUB8	Unsigned Parallel Add Byte
VADDUL2	Unsigned Parallel Add Longword
VADDUW4	Unsigned Parallel Add Word
VMINMAXSB8	Parallel MIN/MAX Byte
VMINMAXSL2	Parallel MIN/MAX LongWord
VMINMAXSW4	Parallel MIN/MAX Word
VMINMAXUB8	Parallel Unsigned MIN/MAX Byte
VMINMAXUL2	Unsigned Parallel MIN/MAX LongWord
VMINMAXUW4	Unsigned Parallel MIN/MAX Word
VMULHUW4	Parallel High Multiply Word
VMULLUW4	Parallel Multiply Word
VSLB8	Parallel Shift Left Byte
VSLL2	Parallel Shift Left Longword
VSLW4	Parallel Shift Left Word
VSRAB8	Parallel Shift Right Arithmetic Byte
VSRAL2	Parallel Shift Right Arithmetic Longword
VSRAW4	Parallel Shift Right Arithmetic Word
VSRB8	Parallel Shift Right Byte
VSRL2	Parallel Shift Right Longword
VSRW4	Parallel Shift Right Word
VSUBSB8	Parallel Subtract Byte
VSUBSL2	Parallel Subtract Longword
VSUBSW4	Parallel Subtract Word

Table 2–16 Short Vector SIMD Instructions

Mnemonic	Operation
VSUBUB8	Unsigned Parallel Subtract Byte
VSUBUL2	Unsigned Parallel Subtract Longword
VSUBUW4	Unsigned Parallel Subtract Word

2.11.2 CMOV Instruction Processing

With register renaming, the CMOV instructions must be treated as having three source operands. A CMOV*x* Ra, Rb, Rc instruction tests Ra for the *x* condition and, if true, moves the contents of Rb into Rc. If the condition is false, Rc is left alone. Because of renaming, the newly assigned Rc register does not already have a copy of the old Rc, so a move has to be done in this case as well. This requires the hardware to read Ra, Rb, and Rc as sources, and to write Rc as a destination as well.

Because the Pbox can only map two source registers and one destination register per instruction, the third source is a problem.The 21264 solved the problem by breaking the CMOV instruction into two separate instructions, CMOV1 and CMOV2.

The 21464 adopts a similar solution — when the 21464 encounters a CMOV, it inserts an additional instruction, CMOV2, into the instruction stream. However, unlike the 21264, if the instruction following the CMOV is the NOP that is described in Section 2.11.2.2, the 21464 replaces that NOP with the CMOV2, instead of creating a new space. That allows the 21464 to map up to four CMOV instructions per cycle. This pair of instructions is called the native CMOV; its implementation is described in Section 2.11.2.5. The pair of native CMOV instructions is mapped at full bandwidth and they require no further treatment in the 21464 pipeline.

2.11.2.1 Integer CMOV Specification

CMOV instructions use the architected integer operate instruction format:

CMOVxx Ra.rq, Rb.rq, Rc.wq

Ra.rq, #b.ib, Rc.wq

The operation consists of testing Ra for the condition specified by the *xx* condition and, if true, the value in Rb is written to register Rc, as follows:

IF TEST(Rav, Condition_based_on_Opcode) THEN Rc <— Rbv

The different conditions specified by the function field are:

CMOV*xx*	Opcode.Function Field	Condition Under Which Rc <— Rbv
CMOVEQ	11.24	Rc <— Rbv if Rav = 0
CMOVGE	11.46	Rc <— Rbv if Rav ≥ 0
CMOVGT	11.66	Rc <— Rbv if Rav > 0
CMOVLBC	11.16	Rc <— Rbv if Rav bit 0 is clear
CMOVLBS	11.14	Rc <— Rbv if Rav bit 0 is set
CMOVLE	11.64	Rc <— Rbv if Rav ≤ 0
CMOVLT	11.44	Rc <— Rbv if Rav < 0
CMOVNE	11.26	Rc <— Rbv if Rav ≠ 0

As described in Section 2.11.2, the 21464 breaks CMOV instructions into CMOVxx1 and CMOV2. For each of these instructions, the CMOVxx1 instruction has the form:

CMOVxx Ra.rq, Rc.rq, Rc.wq

For each of these instructions, the CMOV2 instruction has the form:

CMOV2 Rc.rq,Rb.rq,Rc.wq

Rc.rq, #b.ib, Rc.wq

CMOV2 has opcode/function field 11.68, which is currently an unused function field in the Alpha architecture. Because the architecture does not require that unused function code to trap, there is no conflict with the 21464 opcode detector.

2.11.2.2 Native CMOV

The native CMOV–nop that is recognized and replaced with CMOV2 is:

NOP R31, R31, R31

NOP has opcode/function field 11.20 (same as BIS).

2.11.2.3 Floating-Point FCMOVxx Specification

Floating-point CMOV instructions use the architected floating-point operate instruction format:

FCMOVxx Fa.rq, Fb.rq, Fc.wq

The operation consists of testing Fa for the condition specified by the *xx* condition and, if true, the value in Fb is written to register Fc, as follows:

IF TEST(Fav, Condition_based_on_Opcode) THEN Fc <— Fbv

The different conditions specified by the function field are:

FCMOVxx	Opcode.function field	Condition Under Which Fc <— Fbv
FCMOVEQ	17.02A	Fc <— Fbv if Fav = 0
FCMOVGE	17.02D	Fc <— Fbv if Fav ≥ 0
FCMOVGT	17.02F	Fc <— Fbv if Fav > 0
FCMOVLE	17.02E	Fc <— Fbv if Fav ≤ 0
FCMOVLT	17.02C	Fc <— Fbv if Fav < 0
FCMOVNE	11.02	Fc <— Fbv if Fav ≠ 0

As described in Section 2.11.2, the 21464 breaks FCMOV instructions into FCMOVxx1 and FCMOV2. For each of these instructions, the FCMOVxx1 instruction has the form:

FCMOVxx1 Fc.rq, Fc.rq, Fc.wq

The FCMOV2 instruction has the form:

FCMOV2 Fc.rq, Fb.rq, Fc.wq

FCMOV2 has opcode/function field 17.068, which is currently an unused field in the Alpha architecture for that opcode. Because the architecture does not require that unused function code to trap, there is no conflict with the 21464 opcode detector.

2.11.2.4 Native FCMOV

The native FCMOV–nop that is recognized and replaced with the above FCMOV2 is:

FNOP F31, F31, F31

FNOP has opcode/function field 17.020 (same as CPYS).

2.11.2.5 Implementation

2.11.2.5.1 Native CMOV

At Icache fill time, the Ibox does a partial decode of the 16 instructions being loaded into the Icache. Within each halfblock of eight instructions, pairs of CMOVs and CMOV-nops are detected, and the CMOV-nop is replaced by the CMOV2 instruction. The CMOV–nop instruction is only decoded to a degree sufficient to guarantee that it is an effective NOP. This includes detecting that the destination register is number 31 and making sure that the opcode is 11 or 17.

Predicate Bit

When the Ebox (or Fbox in the case of FCMOV) sees the CMOVxx Ra, Rc, Rc instruction, it tests Rav for the *xx* condition, copies Rcv into the low 64 bits of the renamed Rc register, and if the *xx* condition is true, sets a sixty-fifth bit (the predicate bit) in the register. If the condition is false, the bit is cleared.

When the Ebox (or Fbox in the case of FCMOV) sees the CMOV2 Rc, Rb, Rc instruction, it tests the predicate bit in Rc, and if set, copies Rbv into the new Rc. If the predicate bit is not set, the Ebox (or Fbox) copies Rcv into the new Rc.

The predicate bit is never set unless the 21464 is in the middle of executing the two parts of a CMOV instruction. A CMOV2 with the predicate bit clear is a NOP, since it copies Rcv into Rc. Since interrupts are taken on aligned eight-instruction boundaries, and CMOV does not cause exceptions, the 21464 never takes an interrupt or exception with the predicate bit set.

A CMOV2 instruction can be executed in isolation if software branches to the CMOV2 half of a native CMOV sequence. The original placeholder with a destination of R31/F31 has been remapped to a CMOV2 with the same destination as the original CMOVxx instruction. Since the predicate is guaranteed to be false, the CMOV2 instruction is effectively a NOP that just copies Rc to Rc.

Execution within PALmode

Because the shadow register replacement process in PALmode is keyed to different registers numbers for Rb and Rc, the 21464 does not correctly replace the inserted reference to Rc for native CMOVxx1 instructions in PAL mode. See Section 17.4 for information.

2.11.2.5.2 Legacy CMOV

Legacy CMOVs are CMOV instructions not followed by the designated native CMOV–nop instruction. Legacy CMOV instructions are detected at Icache fill time, and a predecode bit is set for each such instruction. When this instruction is fetched, the Collapsing Buffer notices the set bit and creates a CMOV2 instruction by making a whole new instruction chunk. This new chunk can still be merged with the next fetch-chunk, but this method is limited to mapping at-most one CMOV per cycle.

2.11.3 Mapper Alignment

Although the 21464 hardware tries to schedule instructions in an optimal way, there are occasions where software would like some control of how instructions are mapped and assigned to functional units. For this purpose, the 21464 defines the MAP_ALIGN instruction. When MAP_ALIGN is placed in the last slot of an aligned half-block of eight instructions, it causes that chunk to start a new map-chunk when mapped. That is, the last chunk is not merged in the collapsing buffer with the previous fetch chunk.

The encoding for the MAP_ALIGN instruction is:

XOR R31, R31, R31

The Opcode/function field is 11.40

Implementation

At Icache fill time, the Ibox looks for the MAP_ALIGN instruction in the last slot of the aligned fetch chunk. If found, it sets the MA predecode bit. When this chunk is fetched, the Collapsing Buffer sees the set predecode bit and starts a new map-chunk, beginning with the current fetch chunk. See Table 3–17.

This instruction is only partly decoded. Probably all instructions of the type

XOR *,*, R31 have the effect of starting a new map block when fetched as the last instruction in a fetch chunk.

2.12 Interrupts

Interrupt handling in the 21464 is unlike most earlier processors in three important respects:

- It has no external mechanism for continuously-asserted interrupt requests; all requests are made as network transactions, and held in the processor awaiting service. This implies a requirement for handshaking around the clearing of interrupt requests, to ensure that future requests are propagated to the processor.
- The processor has multiple threads, each capable of running PAL code, interrupt-level service, or user code while the others are active. This implies a requirement for interlocking among the threads which might be servicing interrupts which was not necessary in earlier uniprocessors.
- I/O devices can implement a programmable Interrupt ID register, whose value can be sent with an interrupt request to permit PALcode to vector directly to the appropriate service routine.

External interrupt requests are transmitted through the network as IOWr messages to a block of processor-specific registers. The requestor will receive WrIOAck, except in the case that the message is directed to the Interrupt ID (IID) queue, and that queue is full, when the response will be WrIONack. After receiving WrIOAck, the requestor is expected not to retransmit the request until it has received an explicit release from interrupt software, or it times out. After WrIONack, the requestor can choose to send the request to another processor, retry the same one, or wait for a software timeout.

2.12.1 IPR Access Mechanism

2.12.1.1 HW_MFPR and HW_MTPR PALcode Instructions

PALcode uses the HW_MFPR and HW_MTPR instructions to access the internal processor registers. The HW_MFPR instruction reads the value from the specified IPR into the integer register specified by the Ra field. The HW_MTPR instruction writes the value from the integer register specified by the Rb field into the specified IPR. See Section 17.2 for information.

2.13 AMASK and IMPLVER Instruction Processing and Values

The AMASK and IMPLVER instructions appear to the rest of the 21464 as normal Integer Logic Box (Opcode 11) instructions, but are handled specially by the Ebox.

The Ebox ignores the registers specified in the instruction and forces the CPU feature mask constant onto the Ra operand bus whenever an AMASK instruction is decoded and the implementation version constant onto the Rb operand bus whenever the IMPLVER instruction is decoded. For both these instructions the logic box performs the following operation:

```
Rc = Rb & ~Ra;
```

Given that the Alpha SRM requires Ra == R31, the equations reduce to:

AMASK	Rc = Rb & ~CPU_feature_mask
IMPLVER	Rc = Implementation_version

The current constant values are:

CPU_feature_mask (AMASK)	0x1F07
Implementation_version (IMPLVER)	0x04

2.14 Performance Monitoring

Performance monitoring hardware provides information about the running CPU in order to:

- Drive profiling-directed-feedback optimizations to improve application performance.
- Guide the OS Scheduler to better utilize the TPU contexts.
- Provide architectural feedback for future alpha microprocessor and system implementations.

To satisfy those goals, the 21464 supports three types of performance monitoring:

- An instruction-based profiling algorithm called ProfileMe.

  Instruction-based profiling is performed by sampling the dynamic instruction stream running on the 21464. Sampled instructions are chosen at fetch time based upon a software-programmable IPR and are monitored while in-flight in the CPU. Latencies and events are recorded for two separate instructions into a set of profile record IPRs. When both instructions have finished utilizing CPU resources, a general interrupt to PALcode is triggered.

The general interrupt service routine reads the INTERRUPT-SUMMARY IPR to determine that the interrupt was caused by an instruction profile event. A privileged PAL routine can then read out the associated data for each profiled instruction by issuing MFPRs to the profile record IPRs. In continuous sampling, software would record the data from the current sample and reinitialize the software-programmable IPR to begin the process for selecting the next pair of sampled instructions.

- Aggregate event-based performance counters for monitoring IPC per TPU, as well as intra-thread resource contention of Caches, TBs, and the branch predictor.

  Aggregate performance counters provide expedient insight into chip resource contention problems, especially among processes running on the different TPUs simultaneously. The most potentially problematic resources are the caches (Icache, Dcache and Scache), the translation buffers (ITB, DTB) and the branch predictor. Misses/mispredicts in each of these structures can be counted. Overall performance can also be monitored by using the cycle counter and the retired instructions counter to obtain retired instructions per cycle per TPU.

  There are three aggregate performance counters: the cycle counter, the retired instructions counter, and one general event counter that can count one of the other specified events (Icache miss, Dcache miss, Scache miss, ITB miss, DTB miss or Branch mispredict). The retired instructions, and general event counter are actually four counters that count events per TPU simultaneously.

- Hardware for monitoring memory addresses that was developed for the 21364 and is being supported by the 21464.

  Memory reference performance monitoring hardware is identical to that of the 21364. While the 21464 designers intend to support the same functionality, this specification may change to reflect architectural differences in the memory subsystem of the two processors.

  Instead of IPRs, this performance monitoring hardware is controlled and collected via IO mapped CSRs. There are separate sections for the Cbox, Zbox and Rbox.

3
Instruction Fetch Unit — the Ibox

The Ibox is the instruction fetch engine for the 21464. It is responsible for providing high instruction stream bandwidth to the remainder of the chip. Specifically, the Ibox delivers instructions directly to the Pbox, which is responsible for instruction number (*INum*) resource management, dependence analysis, and register renaming. From there, instructions proceed to the Qbox, where they await the resolution of their source register dependencies. Once an instruction's register dependencies have been resolved, it is issued, provided that it wins arbitration for an appropriate functional unit in the Ebox (arithmetic and logic integer operations), Fbox (arithmetic floating point operations), or Mbox (memory operations). When an instruction has completed execution, it retires when it is the oldest non-retired instruction in the machine for the appropriate Thread Processing Unit (TPU) context.

Instruction stream bandwidth is one of the major factors in overall chip performance. A program cannot execute faster than the rate of instructions entering the machine. Achieving sufficient instruction bandwidth for a machine that can execute up to eight instructions per cycle poses several challenges. In order to meet those challanges, the Ibox contains many new features that were not designed into prior Alpha implementations.

3.1 Features

The Ibox is responsible for:

- Delivering up to eight instructions per cycle to the remainder of the machine
- Maintaining the correct program counter (PC) while the CPU executes programs
- Receiving interrupts and exceptions to properly redirect the machine

The following new features have been added to the Ibox to support high bandwidth instruction stream fetching, advanced control flow prediction, simultaneous multithreading (SMT), and memory dependence prediction:

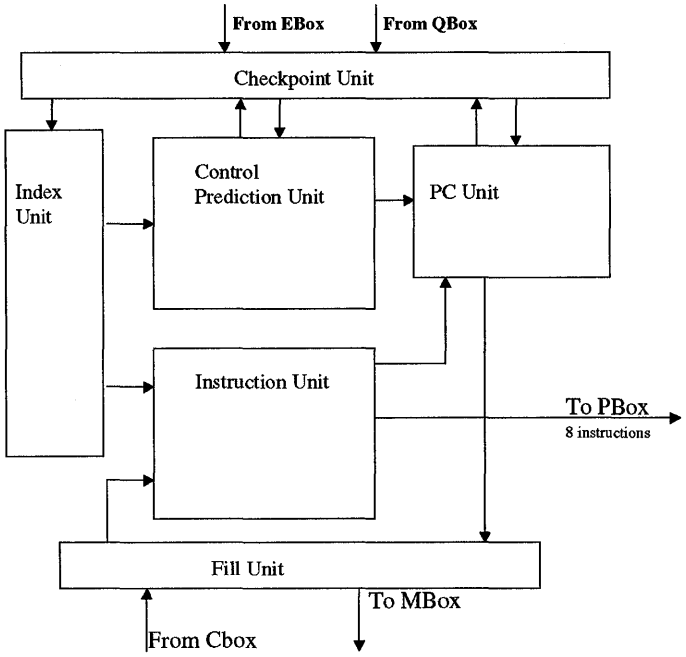
- Fetching up to two potentially non-contiguous cache blocks per cycle
- Fetch TPU Chooser – to create a resource-balanced SMT fetch engine
- Advanced Branch Prediction – predicting up to 16 branches per cycle
- History based Jump Target Prediction.
- Collapsing Buffer – to facilitate over-fetching and merging fetch blocks
- Memory Dependence Prediction using Store Sets

- Advanced Hardware I-Stream pre-fetching
- Simultaneous Multithreaded Fill Unit
- Anti-thrashing Instruction Cache fill policy

3.2 Major Sections

Figure 3–1 Shows the Ibox block diagram. Following the figure is a list of the major sections.

Figure 3–1 Ibox Block Diagram



The Ibox can be thought of as containing the following major sections:

Table 3–1 Ibox Major Sections

Name	Description	Section
Checkpoint Unit	The Checkpoint Unit maintains state for restarting the CPU in the event of an exception, and trains the control flow predictors and the memory dependence predictor. Upon an exception, the Checkpoint Unit resets the following to the state that existed just before the fetch of the instruction that caused an exception: the PC, branch predictor, jump target predictor, and return stack . The Checkpoint Unit also keeps training information for the branch and jump target predictors, to train the predictors at the retirement time of branch or jump instructions.	3.9
Control Flow Prediction Unit	The Control Flow Prediction Unit predicts PC changes at fetch-time for instructions that can change control flow when executed. Control flow instructions are conditional branches, computed jumps, and subroutine returns. There is a dedicated predictor for each: the conditional branch predictor, the jump target predictor, and the return address stack.	3.6
Fill Unit	The Fill Unit fetches instructions from lower-level memory. The Fill Unit can simultaneously fetch instruction blocks for multiple TPUs . The Fill Unit also maintains a dynamic hardware prefetcher that attempts to fill the Icache with blocks that would have missed in the future. The Fill Unit also contains the Icache Translation Buffer (ITB) that must translate virtual PC miss addresses to physical addresses before making memory requests.	3.8
Index Unit	The Index Unit produces up to two indices per cycle. The indices are usually predictions from the Line Predictor that are used to access the Icache, Branch Predictor, and Store Sets Array. The index unit also contains the Fetch TPU Chooser that arbitrates among multiple TPUs that are ready to fetch instructions. The produced indices have an associated TPU that is sent along with them down the Ibox pipeline. The Line Predictor itself consists of a sequential and non-sequential component, to address the sequential and non-sequential code sequences of the running programs.	3.4
Instruction Processing Unit	The Instruction Processing Unit stores and retrieves instructions and their associated tags and data, and contains the following: 1 The 64KB Icache and it's associated tag array. Instruction pre-decode bits are also stored in the Icache Data and Tag Arrays to speed instruction processing in the Ibox and instruction format decoding in the Pbox. 2 The Store Sets Array, which produces memory synchronization identifiers (store sets) for potentially every load and store operation. The store sets instruct the Pbox to create explicit dependencies between certain loads and stores. 3 The Collapsing Buffer, which stores instruction blocks that are driven by the Icache and collapses up to two instruction blocks per cycle to deliver up to 8 instructions per cycle to the Pbox.	3.5
PC Unit	The PC Unit maintains the program counters for each TPU. Mostly it calculates PCs based upon the exiting instructions of the fetch blocks (for example, branches, jumps, returns, fall-through, and so forth), but it also can be reset by interrupts and exceptions. The PC Unit is also determines Icache misses, index mispredicts, and way mispredicts in the Ibox pipeline.	3.7

3.3 Forward Path Pipeline

The main Ibox pipeline is shown in Table 3–2:

Table 3–2 Ibox Main Pipeline

I0	I1	I2	I3	I4
TPU Select	Index Gen.	Icache Access	Collapse	Drive to Pbox
		BPR Predict	PC Calc	
		JPR Predict		
		RPR Predict		

I0 The Index Unit comprises the functionality in stages I0 and I1. In I0, the Fetch TPU Chooser arbitrates among TPU's that are ready to fetch instructions, and selects one each cycle.

I1 The Line Predictor generates up to two valid Icache indices for the selected thread. The Icache indices are predicted because the accessing PCs are not known this early in the pipeline.

I2 The Icache is accessed, and two blocks of up to eight instructions each are read out, along with their corresponding tags and other information. In parallel with the Icache access, the control flow predictors operate to provide conditional branch, jump target or return address predictions for branch, jump or return instructions that are being read out of the Icache simultaneously.

I3 The instructions, along with the control flow predictions, provide enough information to calculate two PCs. The PCs are compared with Icache tags and Line Predictor indices to determine whether the fetches hit in the Icache and the predicted indices were correct. If the Icache accesses were correct, the instructions are buffered in the Collapsing Buffer, which reads out up to two fetch blocks per cycle and collapses the instructions into an eight-instruction map block.

I4 The map blocks are sent on to the Pbox for mapping.

3.4 Index Unit

3.4.1 Fetch TPU Chooser

The Fetch TPU Chooser (FTC) is responsible for choosing one of the TPUs each cycle. The chosen TPU's indices will be driven to the Icache and Branch Predictor. Each cycle, the FTC will choose the TPU that is consuming the fewest Ibox pipeline resources, and is ready to fetch instructions. Ties can occur, and are broken using a round robin algorithm.

In order to monitor Ibox pipeline utilization, the FTC receives input from the collapsing buffer each cycle that indicates the number of entries consumed. The FTC also receives input from the pipeline latches at each stage to monitor the number of in-flight fetch chunks that may eventually consume entries in the instruction buffer. The FTC is responsible for not selecting a TPU if fetching its' corresponding instructions will overflow the instruction buffer. The FTC evaluates whether a TPU is ready to fetch instructions each cycle by receiving input each cycle that either enable or disable a TPU for arbitration. A TPU will be disabled from arbitration if it is awaiting a pending Icache

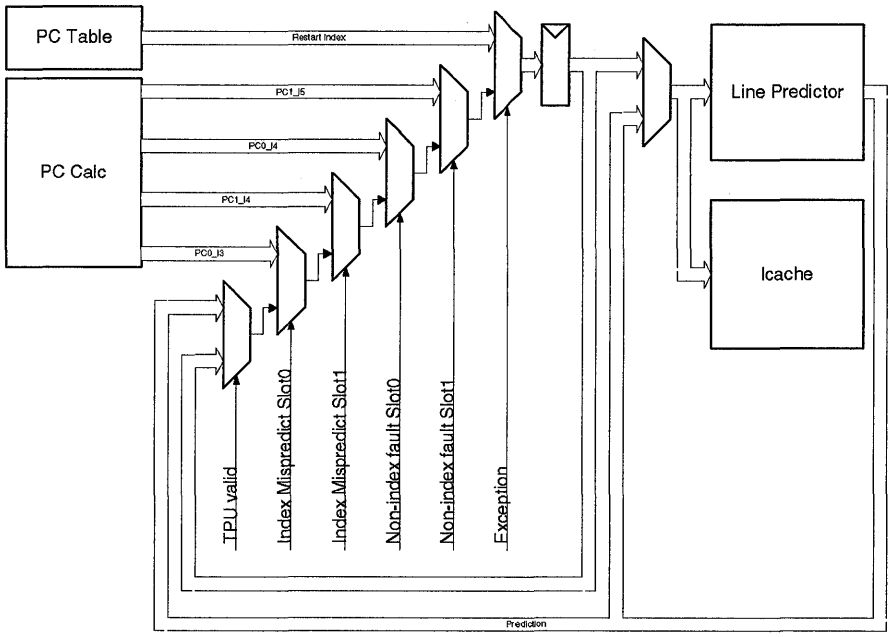
fill, or if that TPU's collapsing buffer is about to become full. The TPU will be re-enabled for arbitration when the pending fill returns or when some collapsing buffer entries are freed.

The FTC is also responsible for ensuring that every TPU makes forward progress. It does this by detecting when a TPU has not mapped real instructions for a very long time, and stalling the other TPUs until the starving TPU maps instructions.

The CPU can be configured to run 1,2,3 or 4 TPUs. An IPR will indicate whether each of the TPUs is "alive" or not. Clearing a TPU's "alive" bit will disable that TPU, that is, the machine will no longer fetch instructions from that context.

3.4.2 Line Predictor

Figure 3–2 Line Predictor Block Diagram



The primary function of the Line Predictor is to provide two indices to the Icache by which it can look up instructions. The index for the Icache are bits <15:2> of the full address. Bit <15> in the Icache index is a *way* bit; it selects which *way* or set stores the instructions. One way bit implies that there are only two ways a fetch block can be stored in the Icache. The way bit <15> can be inverted in contrast to the original bit 15 of the PC to place it the other way inside the Icache. This mechanism stops two different addresses that have the same lower bits <15:2> from occupying the same cache slot. This is also known as *thrashing*.

To maximize effectiveness, there are three different prediction arrays. Already it can be seen that there is a need to predict two indices. One array could be used to predict both indices, but performance can be dramatically increased if some optimizations are done. First, predicting the two indices separately allows different index schemes and thus better independent predictions for the two slots. Secondly, a sequential predictor requires less area by storing a single bit that indicates a sequential index is to be predicted (a

sequential index can be generated with an adder and the current index). A non-sequential array can then have more room for storing purely non-sequential indices while the sequential array can be made quite large due to its small storage requirements. It is infeasible to implement a sequential index generator for Slot1 due to timing constraints, therefore only Slot0 prediction will have a sequential predictor in addition to a non-sequential predictor.

While the line predictor is in a free-running state, meaning that it's prediction is perfect, the line predictor can simply obtain its input index from its own output. This is the secondary function of the Line Predictor, to provide itself with a lookup index. The index that the Line Predictor will use to index itself happens to be the second index it sends to the Icache (Slot1). The read index for the Line Predictor is actually broken up into three indices to access each of it's three arrays. Additionally, the Line Predictor uses a "squash" bit to index itself. There is some hashing involved of the bits to get the final read indices. The three arrays are: slot0 nonsequential array, slot0 sequential array, and the slot1 nonsequential array. Each array is indexed slightly differently, but they all use the same bits. Since the two non-sequential arrays are actually smaller than the addressable index space, hashing is employed to yield the best performance.

For Slot0, there are two hash indices, sequential and nonsequential.

- Sequential - <14:5>,<15>,<4:2>
- Nonsequential - <14:5>,<15>,(<4> | <3> | <2>)

For Slot1 there is just one hashed nonsequential index

- <14:5><15>,(<3> | <2>),(<16> ^ <4>)

Bits <14:5> are commonly decoded for all three predictors. Since the arrays can do the hashing on the fly, only non-hashed index bits need to be stored in the nonsequential arrays.

The Slot0 sequential array has 16k entries and the Slot0 nonsequential array has 4k entries. The Slot1 nonsequential array has 8k entries - more than Slot0 nonsequential because Slot1 does not have the benefit of a sequential predictor.

The Line Predictor index also has a additional "squash" bit. Sometimes the backend of the Ibox pipe can't handle slot0 and slot1 at the same time. Without a squash mechanism, the Line Predictor would have restart Slot1 via a line mispredict (which costs 2-3 cycles for that *TPU*). Instead, the Line Preditor will be indexed by the squash bit instead of bit <15>. In the normal, no-squash case, the squash bit is the same as bit <15>. When PCC detects the squash case, it will simply invert the squash bit from the normal case. So now the Line Predictor can be trained with an "alternate" index by inverting the squash bit so that it now is the inverse of bit <15>. This new "alternate" index will be trained to re-predict slot1 (in the slot0 position) again. It's up to the PC calc section to flip the squash bit for the Line Predictor after it first realizes it can't handle both slots. This way the Line Predictor can keep moving without taking a mispredict for Slot1 every time the backend can't process it.

3.4.3 Thread Index Latches

3.4.3.1 *(Re)Starting/Resuming the Pipe*

When the Line Predictor mis-predicts, it needs to be restarted with an index other than its own output (because it's bad path now). There needs to be some mechanism of generating an index for the Icache and Line Predictor from somewhere other than the Line Predictor itself. The simplest way to provide this capability is to put a mux on the look-up index that picks between the Line Predictor output and an alternate PC. Old predictions for a sleeping thread also need to be stored until the thread is awakened. We have many different sources for an alternate PC. This forms the basis for the thread index latches. In general, PCs from all restarts come from one of three places :

- PAL BASE + OFFSET
- Checkpoint Tables. Jump/Return addresses, alternate PC's, etc. that are stored here.
- PC0 and PC1 - Calculated values for the PC from PC Calc section

There are correspondingly three types of restarts: exceptions, misprediction, and thread resume.

3.4.3.1.1 Exceptions

There are three types of exceptions that can change the PC: Post Map, Ibox internal, and interrupts.

- Post Map

  Post Map exceptions have top priority. All indices for Post Map exceptions are received from the Checkpoint Table, while the signaling of the event can come to the Ibox through two interfaces: the fast path and the Efunnel (Exception Funnel). Fast path exceptions are signaled by the Ebox and Qbox, while the Exception Funnel is entirely contained within the Pbox.

  Exceptions that are caused by mispredicted conditional branches can use the special fast path bypass, which reduces the mispredicting branch penalty. These exceptions can only be acted upon if nothing is coming from the exception funnel that cycle. The Qbox sends the Ibox the INum, TPU, and prediction of the oldest issued CBR every cycle while the Checkpoint Table sends a restart index to the Line Predictor. Two cycles later, the Ebox will send the result of the prediction for that CBR and if there is a mispredict, the exception is taken and the new index is ready to load into the index latches.

  The Exception Funnel is a Pbox widget that filters exceptions such that only the oldest exceptions are signaled to the Ibox. It works by the Pbox sending the Ibox a signal indicating what restart address to use and which TPU is excepting. The Checkpoint Table will use this information to select an index to send to the Line Predictor.

  Since there are all sorts of delays between boxes inside the 21464, there must be some kind of guard against taking a bad-path (younger) exception after an exception has already occurred until the kill has destroyed all remnants of the bad path. To take a bad-path exception is bad, very bad. Older exceptions are ok, however, since they are on the good path by definition. The main problem faced here is that it is not known when all bad-path instructions have been killed. Luckily, there is a large window of opportunity.

Kills happen relatively quickly compared to how long it takes for the first good path instructions to get issued. What's needed is a window of time after an exception is taken during which younger exceptions will be masked out. This is just implemented as a counter. The count wil be determined as follows: *Ibox pipeline stages + Pbox pipeline stages + Qbox pipeline stages until earlist possible issue*, which is hopefully longer than the kill time for bad-path instructions. Note that this covers both the fast-path CBR exception interface and the Efunnel interface

- **Ibox internal**

  Ibox internal exceptions are medium priority, only losing priority to the Post Map exceptions. All indices for these exceptions come to the thread index latches by way of line mispredicts and non-index faults (see below). This means that the Ibox internal exception indices are not directly fed to the thread index latches. Instead they are sent to PC Calc where they will cause an index mispredict. Pipe control will guarantee that only one TPU can take an Ibox internal exception per cycle.

  There are five types of internal exceptions: Reset, Warm Reset, Uncorrectable ECC error, ITB miss, and Read Access Violation. Each of these will be described in more detail in another section. Although the indices physically from PC Calc, they in fact originate from the Checkpoint Tables as a PAL BASE + OFFSET.

- **Interrupts**

  Interrupts have the lowest priority. The Cbox sends the Ibox a 4 bit TPU vector indicating an interrupt on that TPU. Then pipe control will load Pal Base + Offset into the PC latches. This will cause a line mispredict and PCC will send the correct index to the **Line Predictor** to start on. This is the same mechanism as for Ibox internal interrupts. It's important to note that even though Ibox internal exceptions and interrupts have a priority ordering, the Line Predictor can not distinguish the difference between the two. It is up to pipe control to prioritize these.

3.4.3.1.2 Misprediction - PC Calc

When the **Line Predictor** mispredicts, the new start index comes from the **PCC** (PC Calc) section. There are two possiblites: **PC0** and **PC1**, depending on which slot mispredicted. Additionally, a restart can ocurr a cycle later because of a tag problem - this is called a non-index fault. In this case, the PC must be piped one cycle to line up with the restart indication. Here are all the restart cases signaled to the line predictor from PCC in order from youngest to oldest:

1. SLOT0 mispredicts

   This mispredict is the youngest mispredict and therefore has the least priority. The index comes from PC CALC which must go directly into the thread latch. PCC will signal this case late in I3 which makes it a critical path into the thread latches.

2. SLOT1 mispredict

   Again, the index comes from PC CALC which must go directly into the thread latch and is signaled late in I4A so this is also a critical path to write PC1 into the thread latch.

3. SLOT0 non-index fault

The index here is the same as SLOT0 mispredict, but is piped by one cycle (I4A) in the index latches. A way mispredict can be signaled additionally by PCC in I4A. In this case, bit <15> of the PC needs to be inverted before writing into the thread index latch. Also, a bit is sent with this new "inverted" index to tell the PC CALC section not to way mispredict again.

4. SLOT1 non-index fault

   The index is the same as SLOT1 mispredict, but is piped by one cycle (I5A) in the index latches. Again, a way mispredict can be signaled additionally by PCC in I5A. In this case, bit <15> of the PC needs to be inverted before writing into the thread index latch. Also, a bit is sent with this new "inverted" index to tell the PC CALC section not to way mispredict again.

3.4.3.1.3 Thread Resume - Line Predictor (two indexes)

When the Fetch Thread Chooser switches threads, the predictions for the previously active thread need to be saved so that when that thread is reselected in the future, the indices for SLOT0 and SLOT1 are ready to index the Line Predictor and Icache. Other wise, you would lose performance, as the Line Predictor would be generating indices for the thread that just stopped and not the thread that just started. This is the default index source if the thread is selected and no other exceptions have happened.

3.4.3.2 Other Index Latch Tracking Functions

There are few more things the index latches need to track besides indexes: Slot1 Valid, bank conflict, ITB enable, squash prediction, way mispredict restarts, and a guard mechanism:

- Slot1 Valid

  Slot1 valid is set if the index came from the line predictor. For all other cases it is invalidated.

- Bank Conflict

  This means a read and write to the same bank has occurred. Writes take precedence so the predictions that come out of the line predictor in the next cycle are not valid. A bank conflict signal is sent to pipe control (PCC) so that it can invalidate the cycle and the next cycle the index is retried.

- ITB enable

  When a *u*ITB miss has ocurred, the pipe has to be restarted and the ITB needs to be enabled to process the Icache miss. The thread latches hold on to the ITB enable state so that when the *TPU* is selected, the ITB can be enabled.

- Squash Prediction

  As explained previously, the line predictor arrays hold a squash bit for squash prediction. Squash prediction is calculated by XORing bit <14> from the prediction with the squash bit. This prediction is then stored in the index latches so that the prediction can be sent down the pipe when the *TPU* is resumed from sleep.

- Way Mispredict Restart

  When a way mispredict is signaled from PCC, a bit of state needs to accompany the restart index to indicate that a way mispredict is to be ignored.

- Guard

As a safety precaution, there is a guard mechanism set for one cycle after any restart. The guard causes the index latches to ignore any PC Calc exceptions (like mispredicts). In theory, PC calc should not signal a exception after a restart since all pipe stages are killed. The first possible index that could cause an exception is the restart index. The guard is in place only in case pipe control can't kill it's pipe stages in time.

3.4.4 Thread Training Latches

In order for the Line Predictor to actually predict correctly, it needs to be trained to predict the correct indices when it is wrong. Training will require the corrected index to write into the array, the index to write this new data with, and a write enable signal. In truth, there are three separate arrays that are trained independently.
Slot0 Sequential, Slot0 nonsequential, and Slot1 nonsequential are all read simulataneously. This means that the training index (the write index) for all three arrays is the same. In fact, word lines are shared between all three arrays for both reads and writes. Writes, however, are exclusive between Slot0 and Slot1. This is true because if Slot0 mispredicts, Slot1 is killed so it is not known if it's prediction was correct. Slot1 could be trained with the data that was already in the array but this is difficult to implement so Slot1 and Slot0 will have exclusive write enables. Similarly, if Slot1 mispredicts it must mean that Slot0 was predicted correctly so Slot0 doens't need to be trained.

Slot0 also has another case: sequential/nonsequential training. There are three training cases:

- Slot0 predicted sequential and mispredicts (nonsequential) - The sequential array must be written with a 0, or nonsequential prediction. The nonsequential array must be trained with the nonsequential index. Two arrays are trained at the same time.
- Slot0 predicted nonsequential and mispredicts nonsequentially - The nonsequential prediction was wrong and needs to be trained with a new nonsequential index.
- Slot0 predicted nonsequential and mispredicts sequentially - The sequential array needs to be trained for a sequential prediction. The nonsequential array is left alone. It is important that the nonsequential isn't written in this case even though it may seem harmless. The truth is that the sequential array has many more entries than the nonsequential array. The nonsequential prediction may have actually been the prediction for a different index that happened to alias to the same entry in the array. In this case, the nonsequential prediction should be left alone since it may be an accurate prediction for a different index.

So now it can be seen that three write enables are needed. Slot0 Seq, Slot0 Nonseq, and Slot1 (nonseq). There are two more pieces to training. The training index and the training data. The training index is just the index used to access the predictions that were wrong. The job of the training latches is to hold on to this index for each thread. The training data is just the restart PC index bits sent back by PCC plus the squash bit.

Training will only ocur for a thread when training data is available, an index mispredict or way mispredict has ocurred, and that thread is selected by the thread chooser. Therefore, it is the training latches job to keep the write enables and write index for each thread until the thread is selected. The training data comes from the thread index latches since the write data is the same as the current read index in the line pred arrays.

If it happens that the index being restarted is the same as the write index used for the train then a condition known as *bank conflict* will ocurr. This means that both a read and a write are trying to access the same bank and the line predictor array can't handle both at the same time. When this happens, writes will take priority over reads. The cycle of the write there will be no vaild indices coming out of the line predictor. The pipe control must insert a bubble in the pipe for this thread since there are no valid Icache read indices. During the bubble cycle what will happen is that the read index that caused the bank conflict will be tried again so that the next cycle two valid indices will be read out of the line predictor and sent to the Icache.

3.5 Instruction Processing Unit

The Instruction Processing Unit consists of the Icache data array, the Icache tag array, store sets based memory dependence predictor, and the collapsing buffer.

3.5.1 Icache Data Array

The Icache is 64KB. It is pseudo 2-way associative, with a thrash-remap fill policy. A cache block can be stored in one of two possible locations. Most blocks will be stored using direct mapped indexing. However, if two blocks are detected as repeatedly competing for the same direct mapped cache location then one of the blocks will be remapped by inverting the MSB of its index. This condition is detected in the Fill Unit using a thrash detector (see Section 3.8).

The Icache array is made up of 8 banks. Each cycle the Ibox attempts to fetch two half blocks, or fetch chunks from the Icache, one for slot0, the other for slot1. If there isn't an Icache fill occurring in a given cycle, the slot0 fetch is always allowed. The slot1 fetch is only allowed if its fetch is for an entry in a different bank from the slot0 fetch, or if it is for the exact same block (either half) in the Icache as the slot0 fetch. This allows fetching two "fetch chunks" per cycle without double pumping the Icache, as long as the two fetch chunks are not for two different cache blocks in the same cache bank. It has been observed that pairs of consecutive fetch blocks in a variety of benchmarks are about 50% likely to be consecutive. Since consective fetch chunks are either on the exact same cache block, or are in the next cache bank (due to bank interleaving), sequential program access to the cache is guaranteed not to have a read bank conflict and should always be capable of reading two fetch chunks per cycle. Non-sequential fetch chunks that are separated by a multiple of 8 cache blocks will attempt to access multiple cache blocks in the same cache bank, in this case the Slot0 read will be given priority over the slot1 read.

The 21464 provides the MAP_ALIGN instruction, which allows software some control over mapping fetch chunks, in disregard for the efficiencies just described. See Section 2.11.3 for information.

When a cache miss occurs, a cache fill operation fetches and writes a full cache block of 16 instructions through the Fill Unit. Fills are given higher priority than either a slot 0 or slot 1 read. If a fill is occurring to the same bank as either a slot 0 or slot 1 read in a given cycle, neither read will be allowed, and the two reads could be replayed the following cycle. The Index Unit provides the two read indices (slot 0 and slot 1) to the Icache along with a valid bit per read index. The Fill Unit provides the write index during an Icache fill along with a valid bit per Icache bank. The Icache decoders contain logic that arbitrates slot0/slot1 read conflicts, and the read/write conflicts.

The Icache Data Array is parity protected.

The bits in an entire cache block of the Icache data array would consist of:

Table 3–3 Icache Data Array Cache Block Contents

Bits	Description
I[15:0]<31:0>	Sixteen 32-bit modified instructions
CY[15:0]	One overflow bit for each of the 16 instructions
CI[15:0]	One incremented branch target carry bit for each of the 16 instructions
CM[15:0]	One CMOV/FCMOV predecode bit for the CBF, for each of the 16 instructions
PQ[15:0]<3:0>	Four Predecode bits for the P box, for each of the 16 instructions
DP<9:0>	10 Parity bits

3.5.2 Icache Tag Array

The Icache is virtually indexed, and virtually **and** physically tagged. The primary function of the Icache Tag Array is to hold and deliver the virtual address tags for corresponding instruction blocks in the Icache Data Array. These tags are compared with the full virtual PC calculated in the PC Unit to determine if Icache accesses were hits. Address space numbers (ASNs) and the address space match bit (ASM) are also stored in the tag array to determine whether the virtually addressed block that was filled into the cache can be used by the current process that is accessing the cache, which has its own ASN assignment. Physical tags are also stored in the cache to facilitate Icache sharing between two processes that are addressing the same physical memory but were not assigned the same ASN. The two processes must also be using the same virtual index bits to be able to share the Icache. If the physical tag stored in the Icache Tag Array is the same as the physical address of the translated virtual PC, the Ibox allows *physical Icache hits*. The reason for this is to allow multiple threads to share the Icache when running on different TPUs. There is more explanation about how Icache hits are determined in the PC Compare section of the PC Unit documentation.

As described above in the Icache Data Array section, the Icache is pseudo 2-way set associative. Occasionally, when a thrash is detected by the thrash detector in the Fill Unit, the cache will be filled using the alternate location (same as the direct location's index, except the top bit is inverted). The Line Predictor in the Index Unit will learn to predict the alternate way for that fetch block. In order to do this, when the wrong "way" is accidentally fetched, the PC compare logic needs to determine that the wrong way was fetched, and that instead of taking an Icache miss, simply try to fetch the instructions and tags at the alternate way's location. Instead of reading out two full tags, the Icache tag array stores a partial tag for the alternate way. If the partial tag matches the PC, but the primary tag does not, the Ibox attempts to fetch from the alternate way before initiating a cache miss. The alternate tag is 9 bits in the tag array: VA<23:15>.

The Tag Array also stores and retrieves a variety of other information to support several Ibox design choices and features:

- Each TPU belongs to one of four TPU groups. Instead of having one valid bit per TPU in the Icache, there is one valid bit per TPU group. Software can configure multiple TPUs to be part of the same group, in which case they can virtually hit on each others Icache blocks, provided the ASNs match or the ASM bit is set.
- For Icache use by PALcode, a physical fill bit is stored. When the accessing TPU is in fetching PALcode, it can hit on Icache blocks that have the physical-fill bit set and bypass ASN comparison. This is needed because PALcode is always physically mapped.
- When a block is filled, its corresponding protection level is written into the Icache tags. The protection level is either U,S,E,or K as specified in the Alpha SRM.
- The Fill Unit can write an istream block into the Icache if it suffered an uncorrectable ECC error while residing in the Scache. Once the uncorrectible ECC error is detected it will trigger an interrupt, but to keep the Ibox from fetching and processing the bogus block, a bit indicating the uncorrectable error is set in the Icache Tags.
- In order to expidite pc calculation and fetch block processing, a number of instruction attributes are predecoded during Icache fills and then stored into the Icache Tag Array. These bits determine whether each instruction that was filled into the Icache was a conditional branch, unconditional branch, computed jump, or other instruction. It also determines whether the return predictor should be used (ie does the instruction perform a push or pop operation on the stack), and whether the jump target predictor needs to be used.
- The Icache Tag Array is protected by parity.

Here is a complete list of the contents of the Icache Tag Array for a 16-instruction block:

- TPU group valid<3:0>
- Virtual address <51:16>
- ASN<7:0>
- ASM
- Physical address <47:13>
- Alternate virtual address <23:15>
- Physical fill bit
- USEK<3:0>
- ECC uncorrectable bit
- Icache Tag Predecodes [15:0]<3:0>
- Parity<3:0>

Predecodes in Icache Tag for each instruction of each fetch block:

Table 3–4 Icache Tag Array Predecode for Fetch Blocks

UE	CB	P2	P3	PUSH	POP	Meaning
0	0	0	0	0	0	Fall through (CBR o JMP in PALMODE)
0	0	0	1	X	X	Not used — don't care
0	0	1	0	X	X	Not used — don't care
0	0	1	1	X	X	Not used — don't care
0	1	0	0	X	X	Not used — don't care
0	1	0	1	0	0	CBR (conditional branch)
0	1	1	0	X	X	Not used — don't care
0	1	1	1	X	X	Not used — don't care
1	0	0	0	0	0	JMP
1	0	0	1	0	0	BR (unconditional branch)
1	0	1	0	0	1	RET (pops return stack)
1	0	1	1	X	X	IFETCHB
1	1	0	0	1	0	JSR (pushes return stack)
1	1	0	1	1	0	BSR (pushes return stack) (JSR in PALMODE)
1	1	1	0	1	1	JCR (pops and pushes return stack)
1	1	1	1	1	0	CALLPAL (pushes return stack)

The Icache Tag array has 8 banks. The slot0, slot1 and fill arbitration happens exactly as described in the Icache Data array section above.

3.5.3 Store-Sets Based Memory Dependence Predictor

The 21464's out-of-order core could execute a load before a prior store that writes to the same memory location. If this happens the load will get the wrong value. When the store finally executes, *this memory order violation* will be detected and the load and all subsequent dependent instructions will be aborted and re-executed, resulting in a performance penalty. This dilemma has created the need for *memory dependence prediction.* The goals of memory dependence prediction are 1) to predict the load instructions that if allowed to execute would cause a memory-order violation and 2) to delay the execution of these loads only as long as is necessary to avoid a such a violation.

Our memory dependence predictor is based upon the concept of store sets. A *store set* for a specific load is the set of all stores upon which the load has ever depended. A load's store set can be approximated in hardware by first allowing speculation of all loads around older stores. If a load executes before a store upon which it depends, the processor detects a memory-order violation when the store is executed and adds the store to that load's store set. Essentially the processor discovers and remembers a load's store set during program execution. The store set is then used to predict which stores a load must wait for before executing. The table that holds store set IDs is in the Ibox.

For more information about store sets see: George Chrysos and Joel Emer. Memory Dependence Prediction using Store Sets. In Proc. ISCA25, July 1998

Store-sets based prediction replaces the load wait table in the 21264.

The store sets predictor implementation has 16 store set identifiers, and has 4K entries and the table is cleared periodically based upon an IPR. The 4k entry store sets array and the Icache array are read simultaneously based upon an index generated by the Index Unit. Each store set array entry is 5 bits long, one valid bit and four bits for the store set identifier. 8 store set id's are read out contiguously for each fetch chunk. Logic in the Pbox determines whether the instructions are loads or stores to know whether to utilize the store set id or not. The store set id's then create predicted dependencies between loads and stores in the Pbox dependence mapper.

The store set entry table in the Ibox is trained when a store/load order violation is broadcast from the Mbox. The training of store set entries is governed by the following rules:

1. If neither the load nor the store has been assigned a store set id, one is created and assigned to the store instruction.
2. If the load has been assigned a store set id, but the store has not, the store is assigned the load's store set id.
3. If the store has been assigned a store set id, but the load has not, the load is assigned the store's store set id.
4. If both the load and the store have already been assigned store set ids, one of the two store set ids is declared the "winner". The instruction assigned the losing store set id is assigned the winning store set id. The winner is the lower numbered store set ID.

Rule one mentions that when neither the store nor the load involved in the load/store order violation that a store set id is created. The store set id is created by hashing the lower bits of the load's PC:

```
      21 20 19 18
XOR   17 16 15 14
XOR   13 12 11 10
XOR    9  8  7  6
XOR    5  4  3  2
```

As mentioned above, the store set entry table's valid bits are cleared periodically based upon an IPR. The bits in the IPR that govern the store set tables clearing frequency are defined here:

IPR Bits	Clear Freq
0000	Every Cycle (Store Sets Disabled)
0001	Every 1k cycles
0010	Every 2k cycles
0011	Every 4k cycles
0100	Every 8k cycles

IPR Bits	Clear Freq
0101	Every 16k cycles
0110	Every 32k cycles
0111	Every 64k cycles
1000	Every 128k cycles
1001	Every 256k cycles
1010	Every 512k cycles * - recommended setting
1011	Every 1m cycles
1100	Every 2m cycles
1100	Every 4m cycles
1101	Every 8m cycles
1111	Every 16m cycles

3.5.4 Collapsing Buffer

The job of the collapsing buffer is two-fold. It buffers instruction chunks (called fetch chunks) from the Icache and merges usable instructions from these buffered chunks into map-able chunks (called map chunks) that are sent to the Pbox. The collapsing buffer is capable storing and merging 2 fetch chunks per cycle. Map chunks are only sent to the Pbox upon request of the Pbox.

3.5.4.1 Instruction Buffer

3.5.4.1.1 Data Path

Each TPU has it's own buffer (implemented as a queue) of 16 entries with each entry holding one fetch chunk (8 instructions). The ICache sends the instruction buffer up to two fetch chunks every cycle during I3 from a single TPU (slot 0 and slot 1). If the corresponding TPU's buffer is empty and the Pbox requests a map chunk from that TPU, then slot 0 can be bypassed as a map chunk during I3. Slot 1 cannot be bypassed. Both fetch chunks are written during I3 into the instruction buffer queue corresponding to the TPU they were fetched from (regardless of whether one slot was bypassed or not).

The queue addressing logic keeps track of the head and tail of each buffer by using two single bit pointers arranged in a ring. Each buffer is addressed for write by taking into account the position of the tail pointer, and whether there are one or two slots being written. The queue read addressing is similar to the write addressing, except that the head and tail pointers and their relative locations determine whether one or two slots are read. Normally, two fetch chunks will be read from the buffer on each cycle during I3, except in the case there is only one fetch chunk available in the buffer.

The 21464 provides the MAP_ALIGN instruction, which allows software some control over mapping fetch chunks, in disregard for the efficiencies just described. See Section 2.11.3 for information.

In the event that there has been an internal Ibox fault (misprediction, cache miss, etc... Not a branch or jump mispredict) corresponding to slot 0, then the collapsing buffer can catch this by not advancing the write pointers, although the fetch chunks are written. If the buffer was empty before the write, a bypass occurs and all of the valid bits sent to the Pbox are pulled low, as that slot is invalid.

Unfortunately, faults for Slot1 occurs one cycle after Slot0 events. This leads to trouble because slot1 can be written and then read out of the buffer at the same time a Slot1fault happens. In the event that there has been an internal Ibox fault corresponding to slot 1, each buffer has the ability to "undo" the last write by backing up the write pointers. Additionally, all instructions in the map chunk are invalidated just as they are sent to the Pbox. This undo ability allows the collapsing buffer to capture wrong instructions before they get sent to the Pbox. This is vital, since it means that the Ibox will not have to hunt down and kill instructions in other boxes.

3.5.4.1.2 Control Path

To ensure that only map chunks with valid instructions get sent to the Pbox, some additional signaling is needed. First, the Pbox must tell the collapsing buffer some information.

The first thing the Pbox needs to do is request a map chunk. It does so by selecting a TPU in I3 and informing the collapsing buffer of the TPU choice. Sometimes, however, the Pbox selects a TPU and finds that it can't map the map chunk it received from the collapsing buffer. To ensure that this map chunk is not lost, the Pbox will tell the collapsing buffer when to advance the read pointers. When the Pbox is able to map the chunk successfully, it will tell the collapsing buffer to advance the read pointers. Otherwise, the pointers are left where they are so that the map chunk can be retried later.

In order for the Pbox to make choices about which TPU to map, the collapsing buffer will send the Pbox a signal indicating which TPU has instructions in its buffer. Unfortunately, there exists a lag between detecting the emptiness of a buffer and the Pbox actually requesting map chunk. One reason for this is that the Pbox needs to know the state of the buffer one cycle before it can be calculated! This can lead to two problems:

- The Pbox overlooks a TPU with valid instructions.
- The Pbox requests a map chunk from a TPU that it was told had instructions but is actually empty when the buffer gets read. This causes the Pbox to map a chunk of instructions that are all invalid.

A similar case occurs when there is a late kill and the buffer has only bad path instructions. The late kill, as mentioned earlier, clears the valid mask and will cause the buffer to be emptied. In the next cycle, the write pointers will be fixed. In the cycle after this, the buffer is finally declared empty. This will cause the TPU to indicate it is not empty for the kill cycle, the cycle the pointers get fixed, and the cycle after that (remember the bid signal is actually on cycle stale). In this case, the collapsing buffer will tell the Pbox to abort any future map attempts on this TPU after the kill is detected so that no more invalid map chunks get mapped in the mapper.

As mentioned earlier, there is a valid signal sent to the Pbox. This signal contains a valid bit for each instruction in the collapsing buffer which indicates that the instruction is valid for mapping. The collapsing buffer sometimes uses this valid mask for last minute kills as described previously.

3.5.4.2 Collapser

3.5.4.2.1 Data Path

The collapser's overall job is merge two fetch chunks into one 8 instruction map chunk. The collapsing buffer collapser receives two fetch chunks from the instruction buffer in I3. To collapse, the first valid instruction (START) and exit point instruction (END) for each fetch chunk are read from the start/end buffer. The invalid instructions (instructions outside START and END) are stripped off of the two fetch chunks and then the second fetch chunk is appended to first. Finally, the instructions are left justified within the map chunk such that the first valid instruction is always the first instruction.

The operation of the collapser is fairly simple. However, when you fold in the fact that there may be more than 8 valid instructions in two 8 instruction chunks. In this case, the collapsing buffer needs to modify and store the START position for the left over fetch chunk. In the case that a legacy CMOV instruction causes the end of the block, an additional bit needs to be stored that indicates that the instruction corresponding to the modified START is a CMOV2 instruction.

3.5.4.2.2 Start/End Buffer

The start/end buffer not only stores the start and end of the valid instructions in a line, but also the CMOV predecode bits. This buffer is broken up into 4 queues, much like the instruction buffer. Each queue holds 16 entries. Each entry is 25 bits.

Table 3–5 Fields in the Start/End Buffer

Field	Contents
CMOV_PRE<7:0>	CMOV mask for the corresponding FETCH_CHUNK
START<7:0>	1-hot start of valid instruction in the FETCH_CHUNK
END<7:0>	1-hot end of valid instruction in the FETCH_CHUNK
PAL_MODE	Single bit field that indicates if the fetch chunk on this line is a PAL mode chunk or not.

The START data is written into the buffer from the Line Predictor and the END data is written in from the Branch Predictor exit logic. The CMOV predecodes com from the Icache Tags and Pal mode will come from Pipe Control.

3.5.4.2.3 New Start Calculation

When two 8-instruction fetch chunks are collapsed into one 8-instruction map chunk, there is always the possibility that there will be left over instructions in the second fetch chunk. So, the start of FETCH_CHUNK1 needs to be modified to the new start of the valid instructions in that chunk. This is actually a rather simple calculation, and there is plenty of time to do it.

A wrench gets thrown into the works if there is a CMOV in the 8 instruction map chunk. In this case, the new start will correspond with the location of the CMOV in the fetch chunk.

3.5.4.2.4 CMov

It is assumed that the reader of this document has previously read Handling CMov.

The CMOV instruction needs to be spit into two halves CMOV1, and CMOV2. To accommodate the CMOV2 instruction, map chunks are always ended on the CMOV1 (which sits in the same position as the original CMOV). The next map chunk will then begin with a CMOV2. The remaining 7 instructions in the map chunk will be collapsed as normal. The legacy CMOV will also require that the CMOV predecode mask be modified and stored for the next collapsing.

3.6 Control Flow Prediction Unit

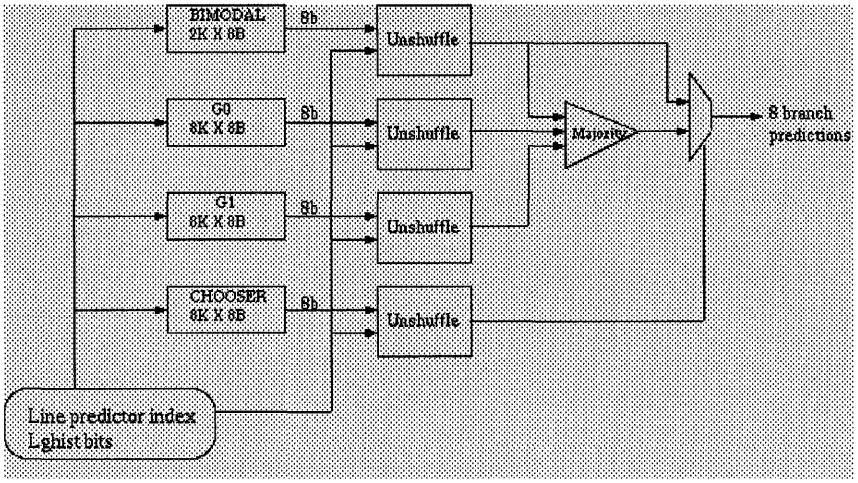
3.6.1 Conditional Branch Prediction

Conditional branches are ubiquitous in most programs. However, it takes at least 13 cycles in the deeply pipelined 21464 before the outcome of the branch is known. Hence, the 21464 processor utilizes an aggressive branch predictor to provide the ability to speculatively fetch beyond conditional branches

The 21464 branch predictor belongs to the class of *skewed* branch predictors. In this class of predictors, multiple prediction tables are used that operate independently to generate a prediction of their own while a majority vote decides the final branch prediction. . For more details, please refer to the technical report by Sezec and Michaud[1]. The 21464 used a modified form of a skewed predictor in which an additional level of prediction is used to choose between the majority vote and one of the prediction tables. There are four tables that constitute the branch predictor that are termed the *G0, G1, BM* and *CH* arrays whose sizes are 64K, 64K, 16K and 64K bits respectively. Tables G0, G1 and BM serve as prediction tables while CH serves as the chooser. Each entry in the table has 8 prediction bits corresponding to 8 instructions in the fetch chunk. An entry for each of these tables is indexed using a unique function that is based on a combination of the branch history bits as well as the address bits used for accessing the instruction cache in the current and previous fetch slot. The 8 prediction bits from each of these tables are further rearranged (unshuffled) using another function of the history and address bits. The final set of 8 predictions for the fetch slot is thus derived after the unshuffle which is followed by choosing between the majority of G0, G1 and BM or the prediction bits of BM itself using the CH (chooser) bits. The overall block diagram of the prediction mechanism is illustrated in the figure 1.

1 A. Seznec and P. Michaud, "Dealiased hybrid branch predictors", IRISA report, Feb 1999, http://www.irisa.fr/caps/PROJECTS/Architecture

Figure 3–3 High level diagram of the 21464 branch predictor



The prediction tables are further complemented with additional hysteresis tables. The sizes of the individual hysteresis tables for G0, G1, CH and BM are 32K, 64K, 32K and 16K bits respectively. It must be noted that unlike traditional schemes, the 21464 predictor does not have a unique hysteresis bit associated with every prediction bit. Rather, the prediction entries are permitted to share hysteresis bits as can be seen from the G0 table that has 64K prediction bits but only 32K hysteresis bits. A reduction in the number of hysteresis bits was shown to have little performance impact while saving valuable die-space. The hysteresis bits prevent the prediction bits to change on the first incorrect prediction thereby disregarding transient changes in branch behavior. The hysteresis bits may be strengthened on a correct prediction and weakened on an incorrect prediction. Unlike the hysteresis bits, the prediction bits may change only on incorrect predictions based upon the state of their corresponding hysteresis bits. The complete training of the hysteresis and predictor arrays is done at instruction retire time. In the following sections, we describe in greater detail the different components and functionality of the branch predictor and training mechanism.

3.6.1.1 Branch Prediction Components

3.6.1.1.1 Branch History (LGHist)

It has been shown that using the past behavior of the branches is extremely helpful in predicting future branches. The traditional method of maintaining the global history (also known as ghist) is to record the outcome of each and every branch that is executed in the program.

To ease implementation, the 21464 uses a modified version of ghist called the line-based ghist (or lghist for short) that records branch history on a fetch line basis. The lghist scheme takes into consideration only the behavior of the last branch of the fetch line. If the branch is in the first half of the fetch line (words 0, 1, 2 or 3), a 0 is entered for a *not taken* branch while a 1 is entered for a *taken* branch. On the other hand, if the last branch happens to be in the second half of the fetch line (words 4, 5, 6 or 7), a 1 is entered for a *not taken* branch while a 0 is entered for a *taken* branch.

It must be mentioned that the branch history used for predicting the branches in the slot that is currently being fetched would not include the information of the slots fetched in the previous cycle. This is because an extra pipeline stage is required to record the predicted outcome of the branches in the lghist. The predicted outcome can be recorded only after discarding those predictions that would play no role when the instructions in the fetch slot are executed. This is achieved by considering (a) entry point in the fetch slot (b) identifying the true conditional branches among the 8 instructions in the fetch slot using pre-decode information and (c) unconditional exit instructions (such as jump, return or unconditional branch instructions). Furthermore, in a given cycle, if two slots are being fetched, the second slot would not only lack the history of the slots that were fetched in the previous cycle but also of the first one that is currently being fetched. Hence, to maintain consistency, the history information used for prediction is always made three slots old for both the slots in a given cycle. The fact that the lghist was modified for a particular fetch slot is maintained using the *shift distance* bits. Note that the lghist would change only in the presence of valid conditional branches in the fetch slot. The shift distance information is particularly valuable on restarts when the restored lghist has to be aged by three slots before being used to access the branch predictor. A maximum of 3 shift distance bits needs to be maintained corresponding to the three-slot aging factor.

In addition to the shift distance bits, another bit called the *no shift* bit is used. This bit prevents the shift distance bits from being modified more than once for the same fetch slot when it is restarted (on an exception). On a restart from an exception, the checkpoint table restores the lghist and shift distance after updating them appropriately depending upon the presence of a conditional branch before and after the restart position in the fetch slot. If the restart position occurs in the same fetch slot and no branches are present after the restart position, the lghist (as well as the shift distance bits) needs to be updated to incorporate any branches before the restart. To prevent future updates to the shift distance for the remainder of the fetch slot that has no conditional branches, the checkpoint table also sets the *no shift* bit. It is this bit that is used to determine if the shift distance bits needs to be updated for the current fetch slot. Note that the *no shift* bit is applicable only for the first fetch slot as a restart on an exception always results in only one slot to be fetched.

Even though all prediction tables are common to all threads, the lghist, shift distance and no shift bit are maintained on a per-thread basis.

3.6.1.1.2 Prediction Tables

As mentioned before, the branch predictor logically consists of four tables: G0, G1, BM and CH. However, this is implemented as one array where each word line in a bank is made up of the four different components. The array is further sub-divided into four single-ported banks with each bank containing 64 word lines. Even though logically each table entry contains 8 prediction bits, implementation constrains each wordline to have several 8-bit entries clustered together. The address bits for indexing the table allow us to select from among the different clusters or "columns" of 8-bit entries.

The address bits that are used to access each of the tables is generated using bits <14:5> of the line predictor index (denoted as *A*) and bits <20:0> of the three slot old lghist (denoted as *H*). The lower bits <4:2> of the line predictor index are not used for the array access. These bits, which denote the entry point in a fetch slot, are used solely to discard predictions prior to the starting point. The address bits are as follows:

a. Word address (6 bits): This is to access one of the 64 wordlines in the array. Since each component resides in the same wordline, these bits are common to all the tables. Moreover, since the address bits must be available at the beginning of the cycle, the address bits are generated directly without any hashing involved. The 6 address bits are: H<3:0>, A<8:7>

b. Column select address: Each wordline consists of multiple entries of G0, G1, BM and CH. These address bits choose from among the many 8-bit "columns" present in a wordline. Since 8 bits are to be chosen from 256 bits of G0, G1 and CH, each of these tables need 5 address bits to choose the appropriate one from a 32-1 column multiplexer. Only 3 address bits are needed for selecting 8 bits from 64 bits of BM (which requires only an 8-1 column multiplexer). The column select bits for each table are as follows:

G0	H<7>⊕ H<11>	H<8>⊕ H<12>	H<4>⊕ H<5>	A<9>⊕ H<9>	H<10>⊕ H<6>
G1	H<19>⊕ H<12>	H<18>⊕ H<11>	H<17>⊕ H<10>	H<16>⊕ H<4>	H<15>⊕ H<20>
CH	H<7>⊕ H<11>	H<8>⊕ H<12>	H<5>⊕ H<13>	H<4>⊕ H<9>	A<9>⊕ H<6>
BM	N.A	N.A	A<11>	A<9>⊕ A<5>	A<10> ⊕ A<6>

3.6.1.1.3 Bank Selection

As mentioned before, the predictor arrays are sub-divided into four banks, each of which has only one read port. Since the branch predictor must be able to predict two fetch slots every cycle, it is necessary that the two slots do not access the same bank in a given cycle. To achieve this, the bank identifier is constructed in such a way that no two consecutive slots would access the same bank.

Since the bank identifier must be available at the beginning of the cycle when the array access is performed, it would not be possible to use any information from the current to slots to generate the bank identifier. For this reason, we use bits 5 and 6 of the line predictor index and the bank identifier of previous slots. Assume that, in the current cycle, the predictor array is being accessed for slots N-2 and N-1 with bank identifiers B_{N-2} and B_{N-1} used for the access. Also, let Z_{N-2} and Z_{N-1} be bits 5 and 6 of the line predictor index used to access slots N-2 and N-1 respectively. The generation of bank identifiers for slots N and N+1 for accessing the array in the following cycle is done as follows: To generate the bank identifier for slot N (B_N), Z_{N-2} is compared to B_{N-1}. If they match, B_N is set to (Z_{N-2}+1) otherwise, it is set to A_{N-2}.The generation of the bank identifier for slot N+1 is also similar; the newly calculated B_N is compared to Z_{N-1}. In this fashion, the bank identifiers for the subsequent two slots are generated in advance by using information that is available in the current cycle.

3.6.1.1.4 Unshuffle Network

Imagine that the branch predictor is implemented such that each entry in the array has only 1-prediction bit. In this case, we would hash bits <14:2> of the line predictor index with the lghist to generate an index for each table entry. In reality, however, each entry stores a set of 8 predictions for each table. Hence the low bits <4:2> of the line predictor index is not used for the access. But these bits are eventually used as they denote the entry point in the fetch slot, and in conjunction with the instruction pre-decode bits that denote the actual conditional branches in the fetch slot, allow us to choose only a sub-

set of the 8 predictions. For performance reasons, however, it is desirable that the low bits are also hashed with the lghist bits when accessing the predictor arrays. Thus, the set of 8 predictions would need to be rearranged (unshuffled) to give the same set of 8 predictions that we would have got in the event of indexing a 1-prediction bit based tables 8 times (to span the low bits<4:2> ranging from 000 to 111)

Let $f_2f_1f_0$ be the bits that are used for XOR-ing with the low bits of the line predictor index while $a_6a_5a_4$ denote the position of a particular branch prediction bit. After the unshuffling, the prediction bit occupies the new position $a_6 \oplus f_2$, $a_5 \oplus f_1$, $a_4 \oplus f_0$. For instance, if $f_2f_1f_0 = 101$ and the 8 prediction bits are $b_7b_6b_5b_4b_3b_2b_1b_0$, the new positions after the unshuffling would be $b_2b_3b_0b_1b_6b_7b_4b_5$.

The hash function used here can be quite complex as the unshuffling is performed only in the later part of the cycle after the array access and column selection has been performed. The address bits used for the hashing include the line predictor index (denoted as *A*), lghist (denoted as *H*) and bits 5 and 6 of the index used to access the previous slot (denoted as Z). The function used for the different tables are listed below:

Table	Unshuffle bits <2:0>
G0	A<9>⊕ A<12>⊕ A<13>⊕ H<5>⊕ H<8>⊕ H<11>⊕Z<5>
	A<5>⊕ A<11>⊕ H<9>⊕ H<10>⊕ H<12>⊕ Z<6>
	A<6>⊕ A<10>⊕ A<14>⊕ H<4>⊕ H<6>⊕ H<7>
G1	A<6>⊕ A<11>⊕ A<14>⊕ H<4>⊕ H<6>⊕ H<9>⊕ H<14>⊕ H<15>⊕ H<16>⊕ Z<6>
	A<10>⊕ A<13>⊕ H<5>⊕ H<11>⊕ H<13>⊕ H<18>⊕ H<19>⊕ H<20>⊕ Z<5>
	A<5>⊕ A<9>⊕ H<4>⊕ H<7>⊕ H<10>⊕ H<12>⊕ H<13>⊕ H<14>⊕ H<17>
CH	A<5>⊕ A<10>⊕ H<7>⊕ H<10>⊕ H<13>⊕ H<14>⊕ Z<5>
	A<6>⊕ A<12>⊕ A<14>⊕ H<4>⊕ H<6>⊕ H<8>⊕ H<14>
	A<9>⊕ A<11>⊕ A<13>⊕ H<5>⊕ H<9>⊕ H<11>⊕ H<12>⊕ Z<6>
BM	NONE
	Z<6>
	Z<5>

3.6.1.1.5 Backend logic and checkpoint information

The final set of 8 branch predictions for each fetch slot is available after the chooser is used to decide between the majority of G0, G1, BM and the BM predictions. However, not all of the final 8 predictions may be useful the following reasons:

1. The instruction for which a branch was predicted taken may not be a conditional branch instruction
2. The entry point in the fetch slot may not be the first instruction
3. Not all instructions may be executed in the fetch slot due to a *taken* prediction for a conditional branch or the presence of an unconditional exit point in the fetch slot (for instance, a jump or a return instruction)

The branch predictor backend logic incorporates additional information using the low bits <4:2> of the line predictor index, instruction types using predecode information as well as the branch predictions itself to calculate the exact exit position in the fetch slot. This information is used by the PC calculation logic to determine the PC of the following fetch slot.

The information that needs to be check-pointed includes the following: lghist, shift distance, no shift, bank identifiers and bits <6,5> of the previous line predictor indices. Furthermore, on restarts, branches after the restart point would have to be considered if the restart occurs in the same fetch slot. If there are branches prior to the restart position but none after, then it must be incorporated in the restored lghist and shift distance. Hence, we also checkpoint the conditional branch attributes that spans all instructions until the first unconditional exit point. Finally, the 8 bits read from each of the predictor tables (G0, G1, CH and BM) are also stored in the checkpoint table for training the branch predictor at retire time. This avoids reading the single-ported predictor array at training time as doing so may result in conflict with the accesses made during fetch time.

3.6.1.2 Branch Training

The validity of the branch predictions is known when the branches are executed in the Ebox. A misprediction causes the branch predictor states such as lghist to be restored. However, the actual branch training does not take place until the Pbox retires the instruction.

As mentioned earlier, the predictor makes use of hysteresis tables to prevent modification of the prediction bits on transient branch behavior. Each prediction table has a corresponding hysteresis table with sizes of the individual tables for G0, G1, CH and BM being 32K, 64K, 32K and 16K bits respectively. Note that the sizes of G0 and CH hysteresis tables are half the size of the corresponding predictor tables. This results in two entries in the predictor table to share an entry in the hysteresis table. As with the predictor tables, the hysteresis tables are implemented as a single array that is interleaved between four single-ported banks. The only difference is the size of the wordline that results due to the reduction in the sizes of G0 and CH tables. Consequently, one partition in the wordline contains 256 bits that comprise of 128 bits each of G0 and CH while the other partition contains 320 bits that consists of 256 bits of G1 and 64 bits of BM. The address bits used to access the wordline, column select and for performing the "unshuffle" are the same as that for the predictor tables except that the high order bit for the column select is no longer applicable for G0 and CH due to their reduced sizes.

When a map chunk is retired, the checkpoint table produces the relevant information regarding the fetch slots comprising the map chunk. This includes information on whether a branch was mispredicted for the fetch slot and if so, the mispredicted position. To avoid reading the single-ported predictor array during training, the predictions that were read from each table at fetch time is also available from the checkpoint table. Using this information, both the prediction and the actual outcome for the fetch slot can be reconstructed.

3.6.1.2.1 Predictor Training

The predictor tables need not be updated on a correct prediction. On an incorrect prediction, only one of the fetch slots that is retired would have an incorrect prediction. This implies that only one of four predictor banks would be accessed for writing the

training information. However, the write to the predictor table may conflict with a read access performed at fetch time. To minimize this conflict, each of the banks has a one-entry write buffer to hold the write data whenever it conflicts with a read to the bank. However, this may not be sufficient when there are back to back retirement of map chunks with a mispredicted branch. Dropping one of the writes to the predictor array is not preferred as it may impair the performance of the predictor. To accommodate this situation, the predictor bank-conflict detection mechanism keeps track of pending writes to each bank. If necessary, a bubble is inserted during the fetch stage to put future reads on hold so as to allow a write pending in the buffer to be cleared.

The predictor tables are trained using the following rules:

1. Nothing to be done on a correct prediction
2. If either the majority or BM is correct, update chooser to the correct state provided the hysteresis is weak
3. For each of G0, G1 and BM, modify entry when the table's prediction is incorrect and its hysteresis is weak provided also that:
    a. Neither the majority or BM is correct
    b. Either the majority or BM is correct but the chooser continues to point to the wrong predictor after the update (i.e. the chooser had a strong hysteresis)

3.6.1.2.2 Hysteresis Training

Unlike the predictor tables, the hysteresis tables would have to be updated for both correct and incorrect predictions. For correct predictions, the hysteresis tables can be written without being read as they are always strengthened. However, for incorrect predictions, we need to perform a read-modify-write of the hysteresis bits for the fetch slot with the mispredicted branch. As with the predictor arrays, a write buffer holds pending data for each bank. This still does not prevent bank conflicts due to read and writes occurring at the same time. Overall, there are three different types of accesses to the hysteresis array that may lead to bank conflicts:

1. Writes for a fetch slot with a mispredicted branch
2. Read table for a mispredicted branch
3. Writes to strengthen the hysteresis bits for fetch slots correctly predicted branches

Unlike the predictor training where a bubble inserted at the fetch point permits reads to be put on hold, we cannot stall retires to avoid hysteresis bank conflicts. Hence, we prioritize accesses and drop the access with the lower priority in favor of the higher one. For the three types of accesses mentioned above, type (i) has the highest priority followed by (ii) and finally (iii). The ordering is such that no training done for a mispredicted branch is dropped. If a bank's write buffer holds a mispredicted hysteresis write with another mispredicted write to the same bank to follow, the read is disabled and a weak hysteresis is assumed as the default read value. If, on the other hand, the incoming write is for a correctly predicted branch, it is dropped in favor of the read access.

The hysteresis tables are trained using the following set of rules:

1. Incorrect prediction
    a. If the majority and BM differ, the chooser hysteresis is weakened

b. For the G0, G1 and BM hysteresis, strengthen if table prediction is correct. If the table prediction is incorrect, then weaken the hysteresis, provided:

Neither the majority or BM is correct

Either the majority or BM is correct but the chooser continues to point to the wrong predictor after the update (i.e. the chooser had a strong hysteresis)

2. Correct prediction
   a. If G0, G1 and BM are all correct, hysteresis is unchanged for all tables
   b. Strengthen BM if it is correct
   c. Strengthen G0 or G1 if it is correct and majority was used by the chooser for prediction

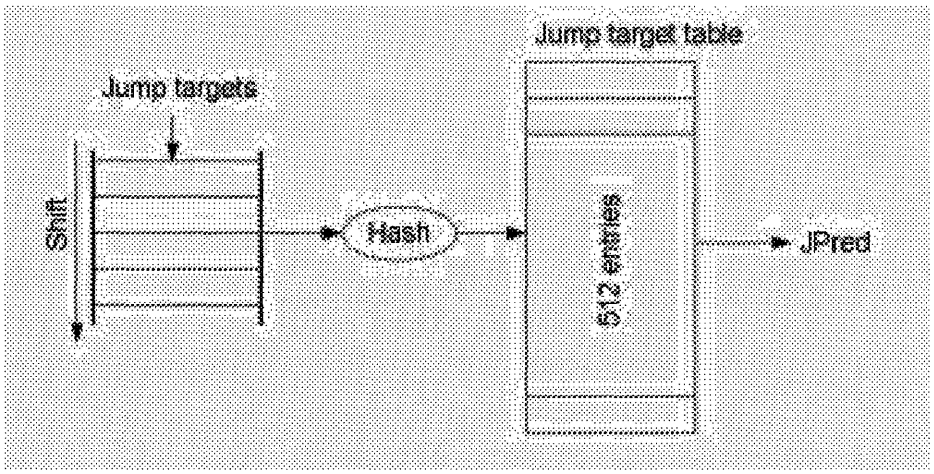
3.6.1.3 PAL mode

In PAL mode, the predecode bits for conditional branches are not set by the instruction fill unit. This implies that the branch predictor is not utilized during PAL code and all branches are predicted as *not taken*. Since branches in PAL mode are rare, this would have little effect on performance. Moreover, we do not want the application specific branch history (lghist) to be corrupted by PAL code branches.

3.6.2 Jump Target Predictor

The Jump Target Predictor is responsible for predicting the targets of Alpha's computed jump instructions: JMP and JSR. The Jump Target Predictor keeps track of partial addresses from the last four jump target predictions – called *jghist*. It hashes together those partial addresses to form an index into a 512 entry target table. The target table is trained with the real computed targets from the execution units. A jghist is maintained for each of the four TPUs, but the 512 entry target table is shared among the threads. The jump predictor can predict one jump per cycle . If both fetch blocks that are fetched in a cycle contain a jump instruction, the first one is processed, and the second fetch block is *squashed* (see PC Unit).

Figure 3–4 Jump Predictor Block Diagram



Indexing into the jump predictor table is a result of hashing of the most recent predicted jump targets as follows:

Assume the four most recent jump targets, from most recent to least recent are: D<51,2>, C<51,2>, B<51,2>, A<51,2>

The 9 bit index into the 512 jump target table for the next predicted jump target is:

```
              D<19, 11>
XOR            D<10, 2>
XOR    CONCAT (C<18,11> | 0)
XOR    CONCAT (C< 9, 2> | 0)
XOR    CONCAT (B<17,11> | 00)
XOR    CONCAT (B< 8, 2> | 00)
XOR    CONCAT (A<16,11> | 000)
XOR    CONCAT (A< 7, 2> | 000)
```

The hash was chosen to ensure position independence of the prior targets, (eg so that the target history A, B, A, B hashes to a different table location than B, A, B, A, since the first should predict target A and the second should predict target B.) Zeros are shifted into the older targets to ensure that older targets count less in the hash.

The jghist registers are checkpointed to facilitate restarting the jump target predictor in case of an exception. When the machine is restarted the appropriate last four targets will overwrite the jghist state that had progressed since the instruction that caused the exception.

Jump mispredictions are trained by writing the correct target into the jump address predictor's table when the mispredicting jump retires. The Checkpoint Unit receives the correct jump target from the Ebox when the jump executes. The Checkpoint Unit will detect a jump mispredict at that time and will keep a record of the correct target to facilitate training once the jump retires.

3.6.3 Return Address Stack

The Return Address Stack is responsible for predicting the targets of returning instructions. The return address predictor is affected by instructions that jump to subroutines and those that return from subroutines. There are several calling instructions:

BSR Branch Subroutine
JSR Jump Subroutine
CPL Call Pal
JSC Jump Co-Routine

There are also multiple returning instructions:

RET Return
JSC Jump Co-Routine

In order to predict return addresses, we use the simple concept of a stack. When a calling instruction is fetched, the PC following the calling instruction is pushed onto a stack. When a return instruction is fetched, the stack is popped, and the PC is redirected to the popped value. The stack holds 64 return PCs per thread.

The return stack must be check-pointed. Upon a abort (branch mispredict, load/store order trap, etc), any pushing or popping that has been done to the stack by instructions on the badpath must be undone to restore the stack to a coherent state. In order to facilitate a fully checkpointed return address stack, we are implementing a structure that behaves like a linked list. We have an array of elements. Each element consists of a PC, and a previous top of stack pointer (PTOS_PTR). Externally we access the array with a top of stack pointer (TOS_PTR). In order to pop the "stack" the array is accessed at the address specified by the TOS_PTR. The PC that is read out is the return target PC, or PopPC, and the PTOS_PTR that is read out corresponds to the next top of stack. When performing a pop, the PTOS_PTR that is read out is written into the TOS_PTR latch. In order to push a value onto the "stack" we need another pointer into the array, which is the next element of the array to be allocated (NALLOC). On a push the array is written at the location specified by NALLOC. The PC written is computed by incrementing the address of the pushing instruction (PushPC) and the current TOS_PTR is written into the PTOS_PTR component of the array element. Then NALLOC is written into the TOS_PTR latch, and NALLOC itself is incremented by 1 (modulo the size of the array). Checkpointing is performed by storing the array pointer corresponding to the current top of stack element in the Checkpoint Unit (See Checkpoint Unit Documentation) for each instruction chunk. The current NALLOC pointer is also stored into the Checkpoint Unit for each instruction chunk in order to reclaim space used by badpath pushes and pops. The stack state is restored by restoring the TOS_PTR and the NALLOC pointer that were stored when the instruction causing the abort was fetched.

Since the 21464 is a multithreaded machine, we need to have a return address predictor that can accommodate multiple code paths without getting confused. In the interest of simplicity, we have decided to simply replicate the return stack array itself. Each of the four (one per TPU) return stack arrays contains 64 entries.

In the 21464, we fetch up to two 8-instruction chunks from the Icache each cycle. Each of those chunks has can contain an instruction that manipulates the return stack. The return stack cannot, however, handle any combinations of pushes and pops in one cycle. It can handle:

- One Push
- One Pop
- One JCR (Pop followed by a Push)
- Pop in slot0, Push in slot1

In the event that two Icache blocks are fetched that do not correspond to one of the four scenarios above, the second block is squashed. (See Documentation for squashing in the PC Unit PC Calculation section).

3.7 PC Unit

3.7.1 PC Calculation

The Program Counter(PC) is a register which holds the address of the instructions to fetch next. In the 21464, there are four TPUs, each of which has an independent instruction stream. To keep track of all the TPUs' instruction addresses, the Ibox maintains four PCs. The PC changes based upon either sequential fetching of the code, or based

upon PC changing instructions such as branches, jumps and returns. The computation of a new PC must occur ever time instructions are fetched; this computation is refered to as PC Calculation.

As mentioned above, the Ibox fetches up to two non-contiguous *fetch-blocks* of instructions each cycle. A fetch block begins with the PC of the first instruction, and all subsequent instructions' PCs in the fetch block must be a sequential increment to the first. Between fetch-blocks, however, the PC's sequential stream can be broken. Since the ibox can fetch up to two non-contiguous fetch blocks in the same cycle, a PC must be generated for each of the fetch blocks. A PC is needed for each fetch block to compare with the fetched Icache blocks' tags for hit determination, and check that the Icache indicies produced by the line predictor were correct and pertained to the correct Icache way. The transition from one fetch-block to another is governed by the exiting condition of the first fetch-block. The list of potential exits to a fetch block are listed in Table 3–6.

Table 3–6 Fetch-Block Exit Conditions

Last Instruction of a 32B Cache Line (Sequential)
Predicted Taken Conditional Branch Instruction (CBR)
Unconditional Branch Instruction (BR, BSR)
Jump Instruction(JSR, JMP, JSR_COROUTINE)
Return Instruction (RET)
IFETCHB Instruction (Halt Fetching until Retirement of IFETCHB)
Call PAL Instruction (CALL_PAL)

Starting with the PC (PC0) of the beginning of the first fetch block, the starting PC (PC1) of the second fetch block is determined by the exiting condition of the first fetch block:

Table 3–7 PC1 Calculation

Something	Something
Sequential	No PC Changing instructions in the first block
	PC1 = CONCAT(PC0<51:5> I 00000) + 1 fetch block (32B)
Taken Branch	Predicted Taken CBR, BSR, or BR
	PC1 = CONCAT(PC0<51:5> I 00000) + Branch Instruction Position Offset in Fetch Block + Branch Displacement
Jump Predicts	JSR, JMP
	PC1 = Output of Jump Predictor
Stack Pops	RET, JSR_COROUTINE
	PC1 = Pop of Return Stack
Call Pal	

Table 3–7 PC1 Calculation

Something	Something
	PC1 = PAL Base Address + Trap Vector Offset
IFETCHB	
	PC1 = PC of the IFETCHB + 4

In order to calculate the PCs, the fetch block exits must be known, as well as the branch predictions, jump target prediction, and the current return address on the top of the return stack. The fetch block exits come from the instructions themselves, ie the Icache data array, and the predictors operate ahead of time to ensure that all the information for PC Calculation is produced as soon as possible. For speed of PC calculation on taken branches, the lower bits of the taken address are pre-calculated and stored in the offset field of the instruction text in the Icache. This happens at fill time. This means that the lower bits: <21:2> of the target pc are not calculated, but simply read from the Icache with the instruction. The higher bits <51:22> need to be calculated. They could be incremented by one, decremented by one, or not change at all based on whether the offset of the taken branch was positive or negative, and whether it caused a carry or borrow (we refer to both as "overflow") above bit 21. The overflow bit and sign bit are stored with the offset in the instruction text at Icache fill time.

Two PCs are calculated per cycle. At the beginning of the cycle the current PC is "PC0", which pertains to the start of the slot 0 fetch block. The two PCs that are calculated are "PC1", which pertains to the start of the slot 1 fetch block, and "NextPC0" which pertains to the start of next cycle's slot 0 fetch block. In effect NextPC0 becomes PC0 for the next cycle. In order to maintain two fetched blocks every cycle both PC1 and NextPC0 are calculated together. The table above showed how exit condition of slot 0 determined the calculation of PC1. In order to determine the calculation for NextPC0, both the slot 0 and slot 1 exit conditions need to be considered. This is because the computation of NextPC0 must start with PC0, and not PC1, which is being computed simultaneously. Considering all of the possible exit combinations, that is 6x6 cross products, is quite a large task.

Several restrictions on combinations of slot 0 and slot 1 fetch chunk exits reduce this considerably. Some of the restrictions are imposed due to hardware limitations (eg, the jump predictor can only handle one jump per cycle, so the slot 0 and slot 1 fetch blocks cannot both end in jmp or jsr). Others were imposed to make the PC calculation logic feasible. The first time the Ibox attempts to fetch two fetch-blocks in the same cycle that violate one of the restrictions, the PC comparison logic will abort the fetch of the second block, and cause a three cycle restart penalty. Thereafter the Line Predictor will remember that the two fetch blocks are incompatible and only the first fetch block will be accessed in that cycle. The second fetch block will be fetched in the following cycle

and it can be combined with a subsequent fetch block. The term for this is ***squashing***. The following table specifies the cases when the line predictor will learn to squash the natural occurance of a slot 1.

Table 3–8 Conditions that Sqaush the Second Fetch Chunk

Both fetch chunks are to the same Icache bank (of 8).
Both fetch chunks end in a JMP or JSR or JSR-COROUTINE.
Both fetch chunks end in a RET or JSR-COROUTINE.
The first fetch chunk ends in JSR, BSR, and the second in RET
The first or second fetch chunk ends in a CALL_PAL
The PC cannot cross a 4Mb virtual address space region delimiter going from fetch chunk 0 to fetch chunk 1
A JSR, JSR-COROUTINE, or BSR is the last instruction of a 4Mb virtual address space region for slot 0 or slot 1

In the hardware, PC calculation is broken down into three components:

Table 3–9 Hardware PC Calculation Components

Component	Bits
The high bits	<51:22>
The middle bits	<21:5>
The low bits	<4:0>

For full functionality, PC1 is always calculated correctly (the Ibox will never squash slot 0, only slot 1). NextPC0 calculation is governed by the squash rules above. The matrixes in Table 3–10 show the three components for the calculation for NextPC0, given those rules.

Table 3–10 Matrix Legend

Matrix	Description
SEQ	Slot exited sequentially, ie no PC changing instruction
TBR	Slot exited with a taken branch – taken CBR or BR or BSR
JPR	Slot exited with a JMP or JSR
RET	Slot exited with a RET
CPL	Slot exited with a CALL_PAL
PC0	Input from the original PC0
PC0+1	PC0<21:5>+1
PC0+2	PC0<21:5>+2
JP	Input from Jump Predictor
JP+1	Input from Jump Predictor <21:5>+1
RP	nput from top of Return Stack

Table 3–10 Matrix Legend

Matrix	Description
RP+1	Input from top of Return Stack <21:5>+1
OF0	Input from the computed branch target <21:5> stored in the Icache Data Array for slot 0
OF0+1	Input from the computed branch target <21:5> stored in the Icache Data Array + 1 for slot 0
OF1	Input from the computed branch target <21:5> stored in the Icache Data Array for slot 1
XXX	Not a legal combination of slot exits, output comes from the computed PC1, indicated in Table 3–9

Table 3–11 NextPC 0 Calculation Matrix

S1_Exit S0_Exit	SEQ	TBR	JPR	RET	CPL
PC0<51:22>					
SEQ	PC0	PC0	JP	RP	XXX
TBR	PC0	PC0	JP	RP	XXX
JPR	JP	JP	XXX	XXX	XXX
RET	RP	RP	JP	XXX	XXX
CPL	XXX	XXX	XXX	XXX	XXX
PC0<21:5>					
SEQ	PC0+2	OF1	JP	RP	XXX
TBR	OF0+1	OF1	JP	RP	XXX
JPR	JP+1	OF1	XXX	XXX	XXX
RET	RP+1	OF1	JP	XXX	XXX
CPL	XXX	XXX	XXX	XXX	XXX
PC0<4:0>					
SEQ	0	OF1	JP	RP	XXX
TBR	0	OF1	JP	RP	XXX
JPR	0	OF1	XXX	XXX	XXX
RET	0	OF1	JP	XXX	XXX
CPL	XXX	XXX	XXX	XXX	XXX

3.7.2 PC Compare

The PC Comparison Logic uses the newly calculated PCs to determine the following:

- If the slot 0 and slot 1 predicted Icache indices were correct
- If the slot 0 and slot 1 cache accesses were hits.
- If there was an instruction access violation
- If slot 1 should be squashed. (see PC Calc section above)

- If slot 0 and slot 1 accessed the correct way in the Icache
- When to make a fill request for an Icache miss

3.7.2.1 Index Mispredicts

Each cycle, the line predictor produces up to two Icache indices, which are necessary to maintain a fully pipelined instruction fetch engine. The actual PCs needed to determine the correct next Icache locations to access are not available for a cycle (slot0) or two(slot1) after they are really needed. So the predicted indices that are generated by the line predictor are checked by the real calculated PCs later in the pipeline. The indices produced by the line predictor are bits <14:2> of the anticipated PC. These bits are compared directly with bits <14:2> of the calculated PCs. If the bits match, the index was predicted correctly. If not, an index mispredict is signaled. The slot 0 index is compared in pipe-stage I3 below and the slot 1 index is compared in I4:

I1	I2	I3	I4
Index Pred	Icache Access	PC0 IDX Comp	PC1 IDX Comp

The Icache can be accessed with the correct index the cycle following an index mispredict. So, for a slot 0 index mispredict there is a 2 cycle penalty, and a 3 cycle penalty for a slot 1 index mispredict.

3.7.2.2 Icache Hit Determination

Whether an Icache access hits or misses is also determined by PC comparison. The Icache tag array produces the tag contents of the accessed cache block. The tag contains several components including virtual and physical tags, as described in Section 3.5.2. The Ibox supports two methods of hitting in the Icache for non-PALcode instructions:

1. Virtual tag hit – occurs when:

   The virtual tag matches the bits of the calculated PC <51:15> AND
   the ASN of the tag matches the Current ASN of the accessing TPU's process context
   OR
   The ASM bit in the tag is set AND
   the accessing TPU's tpu group matches the tags TPU group valid designation

2. Physical tag hit – occurs when:

   PC matches the Micro Translation Buffer's (Micro TB) virtual address <51:13>
   AND
   the Micro TB's valid bit is set AND
   the Micro TB's physical address matches the Tags Physical address<47:13> AND
   the Micro TB's ASN matches the Current ASN of the accessing TPU's context
   OR
   The ASM bit in the Micro TB is set AND the tag is valid for any TPU group

Virtual tag hits are expected to be the normal way of hitting in the Icache. Essentially, the virtual tag matches and the address space number is correct, or the address space match bit is set and the valid bit is set for that TPU's group. Physical tag hits are supported in the Ibox to facilitate sharing common code among different TPUs. Basically, two programs running on different TPUs should be able to share instructions in the Icache if they map to the same physical address. To facilitate this sharing, the Icache tag

array holds the physical as well as the virtual tags for all Icache blocks. Since the PC is virtual, a fast virtual to physical address translation also needs to be done to compare with the physical tags coming out of the Icache.

The Micro Translation Buffer (Micro TB), holds just one page table entry (PTE) per TPU, and so is inexpensive and provides very fast translation for the newly computed PCs. The MicroTB holds the virtual and physical tags as well as the ASN, ASM and TPU group valid bits of the last block that was fetched from the Icache and was a virtual tag hit. Effectively, its a cache of the tag of the most recent virtual Icache hit. If two TPUs address memory at the same physical address, and use the same virtual index to access the Icache, they can share Icache blocks. The first time a TPU attempts to fetch from a page of Icache blocks that are shared by another TPU, it will miss because the ASN or TPU group valid bits will not match for a virtual cache hit, and the MicroTB will be out of date since this is the first access to a new virtual page. But when this first block is brought into the Icache, it will result in a virtual Icache hit, and the PTE information from the newly fetched block's tag will be written into the microTB so that subsequent Istream accesses to that physical page will physically hit in the cache.

PALcode uses a slightly different mechanism to hit in the Icache. All PALcode instruction blocks are mapped physically, so ASN and ASM are not relevant. The virtual tag in the Icache will contain the actual physical address of the instruction block. When PALcode is fetched into the Icache, a bit in the Icache tag is set indicating that the block was physically filled. In order to access physically filled blocks in the Icache, the TPU must be operating in PALmode. An Icache hit occurs in PALmode if the PC matches the virtual tag or physical tag and the TPU and the block was physically filled.

Icache miss determination occurs roughly the same time as index mispredict determination in the Ibox pipeline. Once the PCs have been calculated, they are compared with cache tags to determine if there is an Icache miss. If there is an Icache miss, the pipeline stages prior to the cycle that the Icache miss has been determined are aborted, the fill unit is informed of the newly requested address. The Fetch Thread Chooser is also informed so that it will not choose TPUs with Icache fill requests in progress.

3.7.2.3 Icache Access Violation:

An Icache access violation occurs when an Icache block is fetched and is a hit, but the context of the running process does not have privileges to access that particular block. Each block in the Icache has one of the four privileges:

U — user read enable
S — supervisor read enable
E — executive read enable
K — kernel read enable

The USEK bits are set in the PTE entry for a particular block, and are filled into the Icache Tag Array during a normal Icache fill flow. The current process context for a TPU also has a designated USEK privilege level. An access violation interrupt is initiated when the process context USEK for a TPU does not match the USEK designation written into the tag array for an Icache block that is a hit.

3.7.2.4 Icache Way Mispredict Determination:

The Icache is pseudo-2way set-associative. It is 2way set-associative because instruction blocks can map into two different indexed locations in the Icache. In a standard 2way set-associative cache, both potential block locations are read out, the both sets of tags are compared and if either of the tags matches the accessing address, a hit is signaled and the appropriate block is selected. In the 21464's "pseudo" 2way set-associative Icache, instead of the simultaneous access method used in a standard 2way set-associative cache, one way is predicted and that block is read out. If that block is the wrong one, the other block is read out subsequently. This avoids extending an already critical path in the Icache access path, and keeps the processor's cycle time short.

The PC Compare logic is responsible for determining if an Icache access was to the correct way. If the index generated by the line predictor is correct (ie, bits <15:2> of the index match the PC), but the tag does not match, there is a potential way mispredict. Each blocks tag in the Icache Tag Array stores a subset of the tag that was last filled in the alternate way (See Icache Tag Array subsection in the Instruction Unit section). If those alternate tag virtual address bits <23:15> match those of the accessing PC, a way misprediction is signaled. The Line Predictor, which is responsible for predicting the Icache way, is trained to predict the alternate way next time. On a way mispredict the Ibox pipeline is aborted for the faulting TPU and then restarted accessing the alternate way in the Icache. Since not all the alternate tags virtual address bits were matched, the second access is not guaranteed to be an Icache hit. It could result in an Icache miss if the upper bits of the virtual address tag did not end up matching. It could also result in another way mispredict, if bits <23:15> of the originally accessed way also match the PC, but the upper bits do not match. This can result in a deadlock, where the two Icache locations ping pong back and forth, each time resulting in a way mispredict. To avoid deadlock, the Line Predictor remembers if we already suffered a way mispredict while trying to access the current PC. The second time, an Icache miss will be signaled instead.

Icache way mispredicts are signaled in the cycle after index mispredicts are normally signaled:

Table 3–12 Icache Mispredict Signalling

I1	I2	I3	I4	I5
Way Pred	IC Access	PC0 IDX Cmp	PC0 Way Misp	
			PC1 Index Cmp	PC1 Way Misp

3.7.2.5 Instruction Cache Fill Request:

When a correctly indexed Icache access is not a hit and not a way mispredict, a fill request is signaled to fetch the instruction block from lower level memory. Since the 21464's second level cache is physically indexed and tagged, the Ibox must send a physical address along with the fill request to receive the appropriate data. The Ibox has two sources for translating the virtual PC into a physical address:

- The MicroTB (See Section 3.7.2.2.)
- The 128 Entry Instruction Translation Buffer (ITB) (See Section 3.8.)

In order to save power, the main 128 Entry ITB is not accessible every cycle. Furthermore, it is only necessary to access the main ITB when the MicroTB does not contain the proper PTE. If there is an Icache miss and the MicroTB VA tag matches the upper PC bits and the address space comparison matches, the PA found in the PTE is the correct missing physical page frame number. The page frame number <47:13> is the upper portion of the physical address. It is concatinated with the page offset <12:6> to form the complete physical address of the missing Icache block. If the correct PTE is not found in the 1 entry MicroTB, the main ITB must be accessed to retrieve the page translation. The ITB cannot be accessed immediately because it was not operating to reduce power consumption. The PC Compare logic causes an Ibox pipeline abort and restart, and sends a signal to the Line Predictor indicating that the main ITB needs to be enabled for the next fetch attempt. The next time the missing TPU is chosen by the Index Unit, the Line Predictor will send a signal to the main ITB, which prepares it to be accessed. The next time the Icache miss is detected the ITB will lookup the PC's VA and use the page frame number found in the matching entry to generate the physical address for the fill request.

3.8 Fill Unit

3.8.1 Instruction Translation Buffer

The 21464 has a virtually addressed instruction cache. All memory references outside the CPU core (including the Scache and off-chip memory) are physically addressed. In the event of an Icache miss, the translation buffer's main task is to determine, as quickly as possible, the physical address of the cache line in which the miss occurred so that it can be fetched by the Cbox.

The ITB contains only a subset of all possible address translations, called page table entries (PTEs). Because the ITB itself is a 'cache' of PTEs, it is possible that when an Icache miss occurs and the virtual address is given to the ITB for translation, the PTE is not found. In this case, a trap causes a PALcode routine to lookup the correct PTE from a software table and use an IPR to write the translation into the ITB.

So far, this operation is consistent with the 21264 ITB. However, unlike the 21264, the 21464 includes hardware support for simultaneous multithreading, which has the following implications for the ITB:

- When an Icache miss occurs, only one TPU is affected. It is important that performing the PTE lookup and doing the Icache fill does not stall the pipeline so that other TPUs can continue execution.
- Because ITB fills are completely independant of ITB lookups, care must be taken to reduce the possibility of one TPU's writes interfering with another TPU's read.
- TPUs operating independantly of each other in separate thread groups (TGs) can access the same physical page. To prevent the Icache from storing the same data twice (and thus requesting two ITB lookups), a new mechanism is needed to detect physical address sharing between TPUs. For operating system code, sharing already occurs between processes within a TG (identified by a distinct ASN) by using the ASM mechanism.

- Because each TG is an independant entity to which TPUs can belong (like separate CPUs), PTEs belong to exactly one TG. The ITB stores four one-hot valid bits that indicate to which TG the PTE belongs. TGs do not share PTE entries. The Icache, however, does not signal a miss when there is physical address match, thus preventing the ITB lookup. A PTE hit is determined as follows:
  - pte_VA<51:13> == current_PC<51:13> AND
  - pte_TG<3:0> == current_TG<3:0> AND
  - (pte_ASN<7:0> == current_ASN<7:0> OR pte_ASM == '1')

An Icache miss penalty is significantly reduced because the 21464 includes an on-board, second-level cache (Scache). Thus, time taken for address translation becomes a significant part of the Icache miss penalty, and it is important that the ITB provides a physical address for the fill as soon as possible.

As in the 21264, 8k page sizes are supported. The 21464 can additionally support 64k pages sizes. Granularity hinting is allowed on 64k pages to provide up to 512MB effectively sized pages.

3.8.1.1 Architecture

For the first time in an Alpha implementation, the ITB consists of a pseudo two-level 'cache' of PTEs: a first-level micro ITB (uITB) and a second-level main ITB (the ITB).

The first-level uITB is a single PTE entry for each TPU. It effectively contains the last good address translation that the Icache accessed for each TPU; the uITB is updated any time a TPU virtually hits in the Icache. The PTE information for the update comes from the Icache tag and not from the main ITB array. For ease of implementation, only the first fetch can update the uITB (the 21464 fetches twice per cycle). When there is an Icache miss, the uITB is quickly checked to see if it contains the correct PTE for the fill. If the PTE is good, it is sent to the fill unit and a cache miss is signaled. The physical address is available at the fill unit just two cycles after the miss was detected.

The second-level ITB is a 128-entry fully associative 'cache'. Writes are organized as round-robin by using simple head/tail pointer logic. Simultaneous read/write is not possible, so read scheduling is important. Reads are pipelined across one and a half cycles as follows:

i3b	Cam ASN/ASM and Group Valid
i4a	Cam VA
i4b	Read PTE
i5a	Send PTE to fill unit

To save power, the main ITB is only activated for lookup operations upon a uITB miss (cache miss is implied). This causes a penalty of at least six cycles between cache miss detection and when the physical address is available to the fill unit. For simplicity, the non-index restart mechanism in pipe control is used to enable main ITB lookup. What happens is as follows:

Cycle	Event
0 (I3)	Cache miss detection, *u*ITB determined to be wrong.
1 (I4)	Non-index fault is signalled causing the PC to be replayed in the pipe. Icache miss is NOT signaled.
2 (I1)	Index is sent to Icache
3 (I2)	Icache is read
4 (I3)	Main ITB is enabled. Icache miss detection
5 (I4)	Icache miss signaled. Main ITB lookup in progress
6 (I5)	PTE sent to fill unit

It is possible for the this penalty to be longer than six cycles if the TPU is not selected immediately after the non-index fault.

The main ITB is very similar the 21264 ITB. Super page detection and invalidation operates the same. Additionally, a new invalidate, TBIAG, invalidates all entries in all TGs. Superpages are supported in the main ITB as follows:

Table 3–13 Superpage support in the Main ITB

Superpage	Description
Superpage0	Direct maps one quarter of WindowsNT's 32bit address space. The kernel code is kept in this area of memory. It is believed that 64-bit NT will use the Unix superpage mode.
Superpage1	Direct maps the least significant 41 bits of the physical address space (bits <47:41> sign extended) to support older versions of UNIX and VMS. This superpage is consistent with the 43 bit virtual address supported by EV4, EV5 and the size of the 3 level VPTEs used in Digital UNIX. (see SRM Digital UNIX II-B section 3.1.1).
Superpage2	Direct maps the whole of the physical address space for more recent versions of UNIX and VMS which may use four level PTEs.

If I_CTL[SPE<2>] = 1 AND VA<51:50> = "10"	Then PA<47:13>=VA<47:13>, USEK="0001"
If I_CTL[SPE<1>] = 1 AND VA<51:40> = "111111111101"	Then PA<47:13>= "1111111",VA<40:13>, USEK="0001"
If I_CTL[SPE<1>] = 1 AND VA<51:40> = "111111111100"	Then PA<47:13>= "0000000",VA<40:13>, USEK="0001"
If I_CTL[SPE<0>] = 1 AND VA<51:30> = "11111111111111111110"	Then PA<47:13>= #00000,VA<29:13>, USEK="0001"

Address Space Match (ASM) is supported as in the 21264. When an entry in the main ITB has the ASM bit set, matches against the ASN are ignored when determining TB hit. The uITB also includes this support for hit detection.

Both the main ITB and the uITB utilize the full physical address space permitted <47:13>. PFNs are limited to 32 bits by software, which yields a 64K-page PFN of <47:16> and an 8K-page PFN of <44:13>. When in 8K-page mode, the main ITB sign extends the 45 bit physical address up to bit <47> when filling PTEs, and in 64K-page mode, the ITB bypasses VA bits <15:13> from the PC into the physical address bits <15:13> when reading. To correctly match entries in 64K-page mode, the main ITB ignores VA bits <15:13> because they aren't part of the PFN. The uITB also ignores VA bits <15:13> when performing a VA match in 64K-page mode.

Granularity Hint (GH) is taken care of in the main ITB in the same manner as the 21264. Special CAM structures on the VA bits affected by GH can disable miss detection on those bits thus giving the appearance of an ITB hit on a seemingly larger page. Upon reading the PA out of the main ITB, the affected VA bits are muxed into the corresponding bits of the PA to return the physical PFN. Note that the ITB will always return PTEs for base size (8k or 64k) pages. Essentially, GH allows an ITB entry to cover multuple contiguous base size page translations. Here are the differnent GH mappings:

Table 3–14 Granularity Hint (GH) Mapping

GH Mode ⇒ / Page Mode ⇓	gh<1:0> == 00	gh<1:0> == 01	gh<1:0> == 10	gh<1:0> == 11
8k page size	TB entry covers 8K	TB entry covers 64K	TB entry covers 512K	TB entry covers 4M
64k page size	TB entry covers 64K	TB entry covers 2M	TB entry covers 64M	TB entry covers 512M

The uITB will not have support for granularity hinting or super pages explicitly. This is taken care of since the uITB will just contain an explicit page translation that comes from the main ITB in a round about fashion. For example, the first request in a super page region will result in a main ITB read. The result of this read will return the hardwired superpage PA for that VA. The fill unit will fetch the required Icache blocks and write the hardwired PA into the Icache tags. The next time that block is fetched from the Icache successfuly, the uITB will be updated to contain the hardwired superpage PA. Granularity Hinted pages will only be stored as a base page size translation in the uITB. Jumps outside of a base page will cause a uITB miss although the main ITB will hit on the same translation that filled the uITB.

3.8.1.2 IPRs That Affect the ITB

Table 3–15 IPRs that Affect the ITB

IPR	Affect on the ITB
ITB_TAG	This IPR contains the VA used for filling the main ITB and also performing invalidate operations. There is one for each TPU.
ITB_PTE	This IPR contains the PTE used for filling the main ITB. Retiration of the MTPR to this IPR causes the ITB_TAG and ITB_PTE contents to be written into the main ITB. There is one for each TPU.
ITB_IASN	When a MTPR to this IPR retires, all entries for the current ASN and *TG* are invalidated. The Icache must be invalidated for the current *TG* and the *u*ITB invalidated. There is one for each TPU.
ITB_IA	When a MTPR to this IPR retires, all entries in the current *TPU's* thread group are invalidated. The Icache must be invalidated for the *TG* of the current *TPU*. The *u*ITB must also be invalidated for the *TPU*. There is one for each TPU.
ITB_IS	When a MTPR to this IPR retires, any entry that matches the VA in ITB_TAG and matches the current *TPU's* ASN and *TG* will be invalidated. The Icache must be flushed and the *u*ITB invalidated. There is one for each TPU.
ITB_IAP	When a MTPR to this IPR retires, any entry that is valid for the current *TG* and whose ASM bit is not set will be invalidated. The Icache must be flushed and the *u*ITB invalidated. There is one for each TPU.
I_CTL	When a MTPR to this IPR retires, the SPE<2:0> bits in this IPR enable the 3 super page modes. Each *TPU* has it's own I_CTL IPR.
PCTX	When a MTPR to this IPR retires, the ASN<7:0> bits in this IPR will indicate which ASN is assigned to the *TPU*. Also, the TPU_GRP<3:0> bits in this IPR indicate which thread group the *TPU* belongs to. Each *TPU* has it's own PCTX IPR.

3.8.1.3 ITB Operations

3.8.1.3.1 Fills

Writes stem from an ITB miss flow (PAL code). Here's a break down of what happens from the miss code:

- A MTPR (Move To Processor Register) to the ITB_TAG IPR is issued. The data is written into a speculative register.
- A MTPR to ITB_PTE is issued. The data is written into a speculative register.
- When the MTPR to ITB_PTE retires, the data in the speculative registers are written into the main ITB array. The ASN of the current process is also written into the array from the PCTX IPR. Note that there is no *real* ITB_PTE or ITB_TAG IPR register, only a speculative register.
- The MTPR to the ITB_PTE register must be followed by an IFETCHB to ensure the main ITB state is updated before it is used.

The *u*ITB gets written by a much more ciruitous route. It starts with a cache miss which requires the main *u*ITB to supply the fill unit with a *PTE* (PA, USEK bits, and the ASM bit) for the page being requested. When the fill data returns, the fill unit supplies this

PTE to the Icache Tags for writing. The final step to writing the *u*ITB requires that the Icache virtually hits on an entry containing the this *PTE*. The virutal hit causes the *u*ITB to be written with *PTE* from the Icache Tags.

3.8.1.3.2 Reads

Reads are explained previously.

3.8.1.3.3 Invalidates

There are four invalidate operations for the ITB.

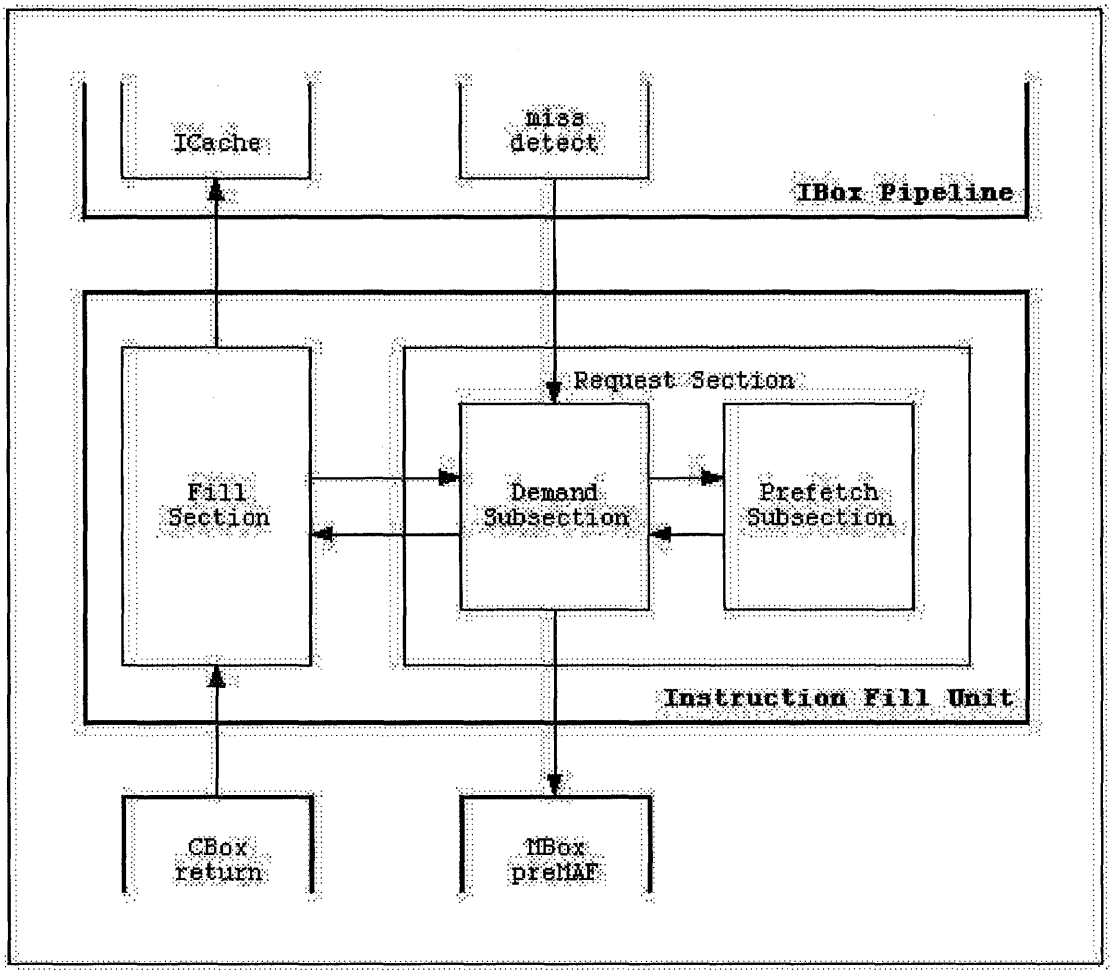
Table 3–16 ITB Invalidate Operations

Operation	Description
Invalidate All	Invalidates all entries within the current *TPU's TG*. Requires only a MTPR to the TB_IA IPR. The actual invalidate is performed upon retire of the MTPR. Additionally, the Icache must be invalidated for this *TG*. An IFETCHB must follow the MTPR to ensure the ITB is up to date before it is accessed again.
Invalidate ASN specific	Invalidates all entries in the ITB that match ASN and *TG* of the current TPU. Requires only a MTPR to the TB_IASN IPR. The actual invalidate is performed upon retire of the MTPR. Additionally, the Icache must be invalidated for the current *TG*. An IFETCHB must follow to ensure the ITB is up to date before it is accessed again.
Invalidate Single	Invalidates a single entry specified by VA, ASN, and *TG*. Two MTPRs are required. The MTPR to the ITB_TAG IPR must ocurr before the MTPR to the TB_IS IPR. Upon retireation of the TB_IS MTPR, the ITB_TAG VA, the current ASN, and the *TG* of the current *TPU* will cam against the contents of the main ITB. Any matching entries will be invalidated. Additionally, the Icache must be invalidated for this *TG*. An IFETCHB must follow to ensure the ITB is up to date before it is accessed again.
Invalidate Process Specific	Invalidates all entries within the current *TPU's TG* which do not have the ASM bit set. The actual invalidate is performed upon retire of the MTPR. Additionally, the Icache must be invalidated for this *TG*. An IFETCHB must follow the MTPR to ensure the ITB is up to date before it is accessed again.

3.8.2 Instruction Fill Unit

The Instruction Fill Unit (IFU) is responsible for fetching instructions when an ICache miss occurs. It consists of two sections: Request and Fill. The Request section itself is made up of two subsections: Demand and Prefetch. The Demand subsection handles ICache misses detected by the Ibox pipeline, while also recording and sending all Ibox memory requests to the Mbox preMAF for servicing. The Prefetch subsection generates a fixed number of consecutive memory requests ahead of the original miss and routes them to the Demand unit for Mbox handling. The Mbox preMAF funnels together both Instruction stream and Data stream requests and delivers them to the Cbox for fetching. As the I stream requests are satisfied, the resultant instructions are sent from the Cbox to the Fill section of the IFU for predecoding and loading into the ICache. The following simplified block diagram shows the IFU and its Request and Fill sections in relation to the ICache, Ibox Pipeline, Mbox preMAF, and Cbox return logic.

Figure 3–5 Instruction Fill Unit (IFU) Request and Fill Sections

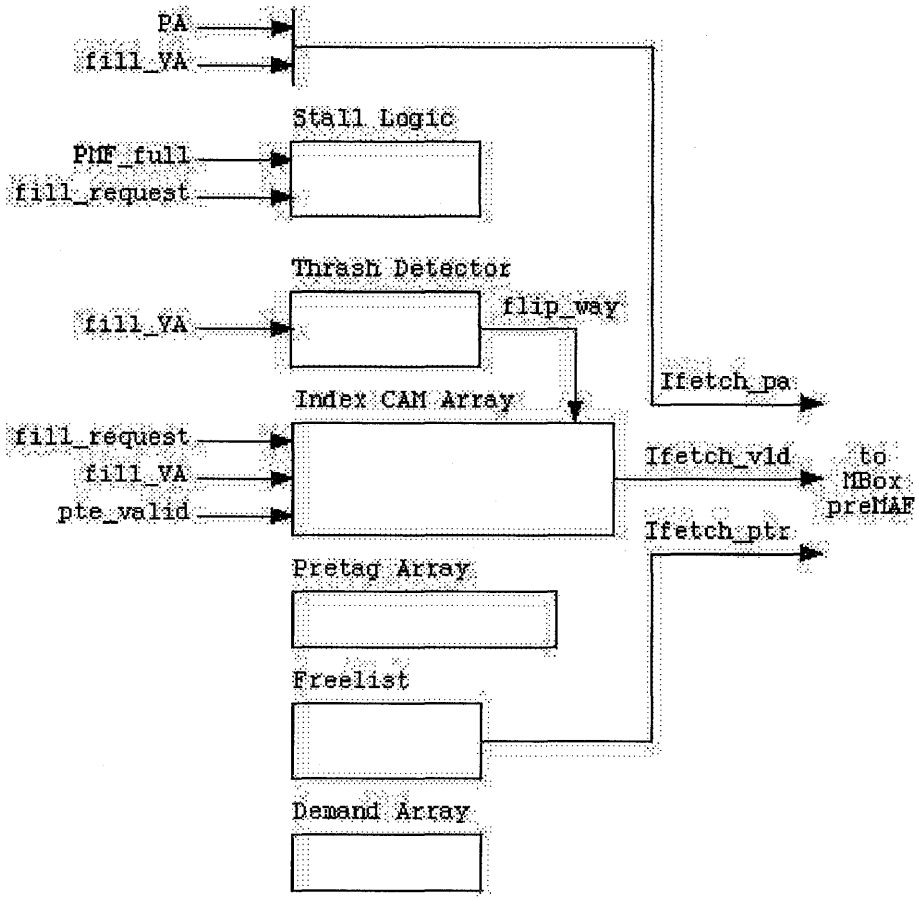


A fundamental assumption in the design of the IFU is that memory requests are never cancelled once they have been sent to the Mbox. Dropping an unneeded request would be dangerous because a remote part of the IFU might simultaneously decide that the request was required after all. Furthermore, the minor benefit of cancelling certain requests would not be worth the additional hardware cost of tracking dropped requests.

3.8.2.1 Demand Misses

The Demand subsection of the Request portion of the IFU is responsible for sending ICache miss requests to the Mbox preMAF for servicing by the SCache and/or Memory. A simplified block diagram of the Demand subsection appears in the figure below. The physical address (PA) and pte_valid input signals come from the ITB, while all of the others are from the Ibox pipeline. The fill_request signal serves as the valid bit for demand requests from that pipeline.

Figure 3–6 Instruction Fill Unit (IFU) Demand Subsection



The Index CAM Array is used to determine if there is another fill request currently outstanding to the same ICache index and way as the current valid miss. For a 64 kB instruction cache, fill_VA<14:6> is the index, while fill_VA<15> is the way bit unless the flip_way signal from the Thrash Detector indicates that bit should be inverted. The Pretag Array stores fill_VA<51:6> and ITB information for each request, all of which is retrieved when the fill instructions return from the Cbox. Together, these two arrays are referred to as the Entry Arrays; simulation studies have shown that 32 entries are appropriate.

The Freelist is a stack whose top indicates the next available free location in the Entry Arrays. Because there can exist at most only one demand for each TPU, the Demand Array contains four pointers (with valid bits) that indicate the index of a given TPU's demand resides in the Entry Arrays. Each Demand Array entry also contains a "piggyback" bit detailed below. The Stall Logic is used to record which TPUs attempted to use the IFU when it was full, or when either the Mbox preMAF or Cbox MAF were full.

3.8.2.1.1 Demand case: simple

The first case to consider is a simple demand. The fill_request signal goes high at the start of cycle I5, indicating a valid miss. The pte_valid signal is true for this simple case, while both PMF_full and flip_way are not. The Index CAM Array is probed: in a simple demand, there is no match with any valid entry. Starting in I6, the output of the

Index CAM Array therefore indicates that this is a legitimate miss, so Ifetch_vld goes high. The Mbox preMAF uses this signal to validate the Ifetch_pa, consisting of PA<47:13> from the ITB and fill_VA<12:6> from the Ipipe; and the Ifetch_ptr, which is a token used to uniquely identify this request.

Simultaneous with the request shipment to the Mbox in I6, the Entry Arrays are written with data at the index indicated by Ifetch_ptr; and in the following phase, the Freelist stack is popped. The Pretag Array caches the following signals: fill_VA<51:15> and ASN<7:0> from the Ipipe, PA<47:13> from the ITB, and 11 bits of control called the hit conditionals. These latter bits consist of physical_fill, console, and tg_valid<3:0> from the Ipipe, along with ASM and USEK<3:0> from the ITB. Finally, the fill_way bit is appended to fill_VA<14:6> and stored in both of the Entry Arrays. The Demand Array is written in I6 with the Ifetch_ptr at the location indicated by the TPU number.

3.8.2.1.2 Demand case: index and way match of active request: "piggybacking"

In order to avoid potential livelock cases, the IFU allows only one outstanding memory request to a given ICache index and way at any time. A novel technique, denoted "piggybacking", is used to handle the request if such a match occurs. Following the simple case above, the input control signals are the same, but here, the Index CAM Array indicates a match. This forces the Ifetch_vld signal to become false to prevent the fetch from occurring. The Demand Array is written at the requesting TPU with the pointer to the entry that matched the request, and the piggyback bit for the same TPU is set.

When the instructions from the original request arrive from the Cbox, the Fetch Thread Chooser (FTC) is notified to restart the corresponding TPU. A few cycles later, the FTC is allowed to restart any other TPU that piggybacked onto that request. The use of the term "piggybacking" for this method is now apparent, because any subsequent demand misses that match an active request ride along with that request.

Recall that only the ICache index and way are checked for piggybacking, not the higher-order VA bits. Simulation studies have shown that these bits often match as well. Most often, this is caused when a redirected goodpath restart requests a block already desired during badpath execution, or when a demand miss contains a short forward branch to code being prefetched. If the higher-order VA bits to not match, the TPU restart of any piggybacked entry results in a new miss.

3.8.2.1.3 Demand case: flip_way active

The ICache way into which a given miss will fill is determined at miss time. The majority of requests will fill into their "natural" way, which is fill_VA<15> in a 64 kB ICache. The Thrash Detector determines under which circumstances flip_way goes high, indicating that the complement of fill_VA<15> (known as the "alternate" way) should be used.

The decision whether or not a demand miss must piggyback is a function of what the fill_way is determined to be. Consequently, the Thrash Detector output must be read and the fill_way altered before the Index CAM Array is probed for a match. Otherwise, a miss that doesn't match an active request in its natural way might have its way toggled and have an alternate way match that is not detected.

3.8.2.1.4 Demand case: capacity stall

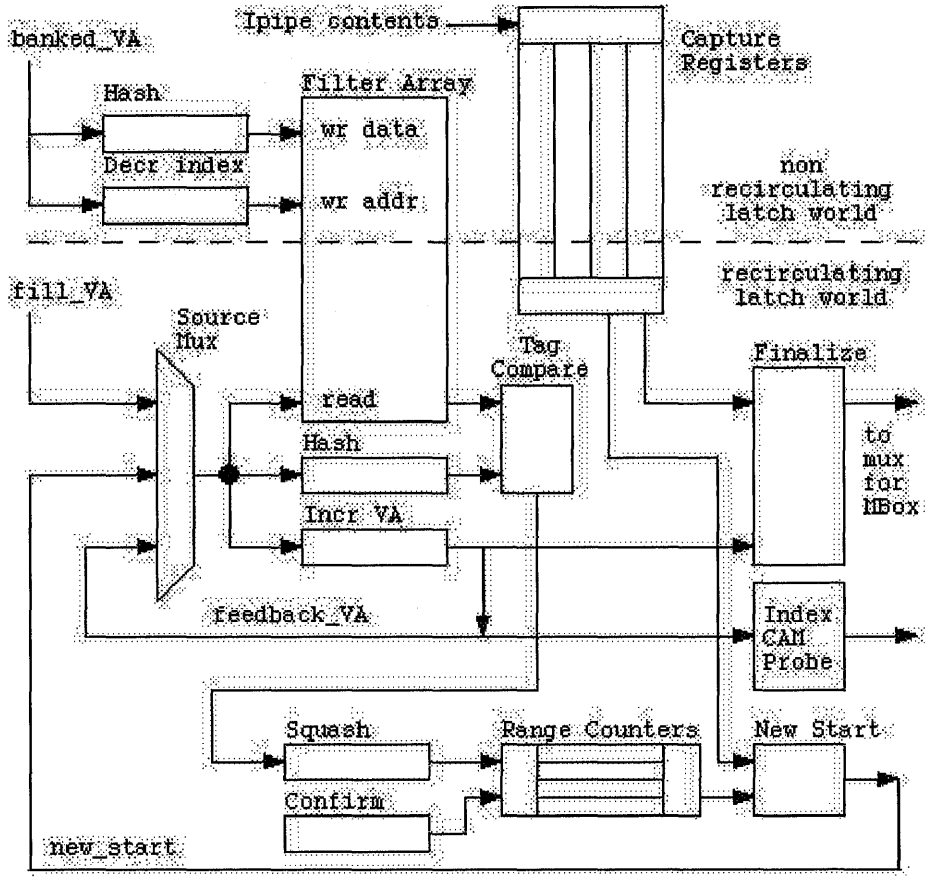
There is finite storage for handling memory requests, both in the IFU (in the Entry Arrays) and beyond (in the Mbox preMAF and the Cbox MAF). A full_resource signal is raised when any or all of these storage areas are full, taking into account any in-flight delays. If a demand miss arrives when full_resource is high, a stall bit corresponding to the TPU number of the request is set. When the full_resource line falls, the stall bits are sent to FTC, indicating which threads must retry their requests.

3.8.2.2 Prefetching

Once a demand miss has been confirmed by the Ibox pipeline, the Prefetch subsection can generate memory requests. Because the IFU interface to the Mbox preMAF can accept one memory request per cycle, the Prefetch subsection generates a single request per cycle and routes it to the Mbox through the Demand subsection interface. The prefetch requests are for consecutive ICache blocks beyond the confirmed miss address. The maximum number of such prefetch requests that are generated for a given miss is determined by a per-TPU IPR value. The actual number sent to the Mbox may be less than the maximum due to filtering.

A simplified block diagram of the Prefetch subsection appears in the figure below. There are four 101-bit Capture Registers (one per TPU), each of which saves all of the required information about a confirmed Demand miss that will be needed to generated prefetch requests (specifically, fill_VA<51:6>, ASN<7:0>, and fill_way from the Demand subsection, PA<47:13> from the ITB, and the 11-bit hit conditionals from both). The Filter Array is essentially a copy of the ICache Tag array in that it contains 2 sets of 512 entries each (for a 64 kB ICache), but this copy stores a hashed version of the tags in order save chip area. The Index CAM Probe determines if another active request shares the same ICache index and way as the feedback_VA. It is shared with the Index CAM Array in the Demand subsection.

Figure 3–7 Instruction Fill Unit (IFU) Prefetch Subsection



An unusual characteristic of the Prefetch Subsection is that it conceptually exists in two different time domains. Simulation studies have shown that demand misses should always proceed ahead of prefetch requests, so a portion of the design uses recirculating latches between clock stages to "freeze" the prefetch state while a demand miss is sent to the Mbox. Yet certain inputs arrive from the Demand and Fill Subsections that cannot be frozen without data loss, so they must be handled immediately. More specifically, the inputs to the Capture Registers and the Filter Array are stored as soon as they are valid. The different time domain "worlds" in the prefetcher are distinguished by a dashed line in the figure.

3.8.2.2.1 Prefetch case: simple

Activity in the prefetcher begins when an ICache miss is confirmed by fill_request in the Demand subsection. When that is using the Index CAM Array to check the index and way of the demand miss, the prefetcher feeds the VA through the Source Mux to look up the hashed tag, while the same fill_VA is both incremented and hashed in parallel. Ideally, an incremented fill_VA would be used to index the Filter Array, but there is insufficient time to do both the increment and the lookup in a single cycle; instead, the index to which the hashed version of the banked_VA is written is decremented before writing the array. ICache_miss confirmation also initializes the per-TPU Range Counter in I5.

The appropriate Ipipe contents are latched into one of the Capture Registers in I6. This also triggers the Tag Compare of the hashed Filter Array tags with the hashed fill_VA: in the simple case, there are no matches. The Index CAM Array probe also occurs in I6, as long as the Demand subsection does not need the shared hardware. The Capture Register data is then used to construct a request that is sent to the Mbox via the port in the Demand Subsection. Because this prefetch pipeline is one cycle longer than the demand one, the first prefetch can be sent to the Mbox in the cycle after the demand has been sent. This is critical, because the prefetched blocks most likely to be needed for execution are those most near the demand miss.

Once it is confirmed that this prefetch request has been accepted, the Confirm box sends a decrement signal to the per-TPU Range Counter, which stops the generation of new prefetch requests when the range becomes zero. Until then, the Source Mux will select the feedback_VA as its input, which is the fill_VA from the previous cycle incremented by the ICache block address. Simulation studies have shown that the optimal number of consecutive blocks to fetch ahead of the demand miss is usually between 2 and 4. The Range Counters are therefore 3-bit counters, allowing a maximum fetch-ahead distance of 7 ICache blocks.

3.8.2.2.2 Prefetch cases: tag match or page boundary crossing

A variety of conditions may keep the number of prefetch requests sent to the Mbox below the value specified by initial Range Counter value. First, if the Tag Compare unit detects that a hashed tag in the Filter Array matches a hashed version of the Source Mux VA, it is highly likely that the stream of prefetch requests will be for instructions already (or soon to be) resident in the ICache. In order to preserve memory bandwidth, prefetching is squashed (stopped) by zeroing the proper Range Counter and invalidating the matching request before it is sent to the Mbox. Any request that crosses an 8K page boundary is also squashed because its PA would require an ITB translation different from that stored in its Capture Register (superpage handling is TBD).

3.8.2.2.3 Prefetch case: Index CAM match

If the Index CAM Probe reports a match without a Tag Compare match or page crossing, the given request is skipped (by forcing Ifetch_vld false) but prefetching is not squashed. Recall that an Index CAM match indicates that there is another currently-outstanding request to the same index and way as the probe. Because prefetching is inherently speculative, it is considered too risky to have a prefetch request displace another ICache request, particularly if the other is a demand miss.

3.8.2.2.4 Prefetch case: alternate TPU demand during prefetching

One TPU may produce a demand miss and start prefetching when another TPU also confirms a miss. When this happens, the recirculating latches "freeze" the state of the prefetcher while the new demand miss is sent to the Mbox and the new demand state is captured in the proper Capture Register. The prefetcher then resumes running in the following cycles until the appropriate Range Counter is zero. The New Start logic then notices that the new demand state for the alternate TPU is ready, so prefetching proceeds for that TPU.

More than one Capture Register may have valid state ready for prefetching. This requires the New Start logic to implement a picker to select amongst multiple ready TPUs. Simulation has shown that this is a very rare occurrence, so any simple picking algorithm is acceptable.

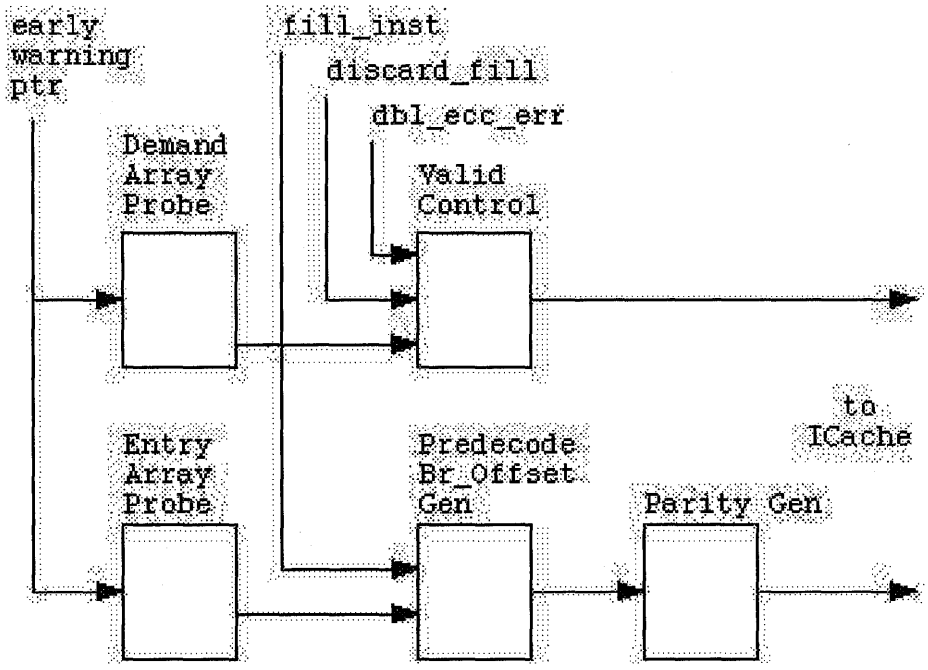
3.8.2.2.5 Prefetch cases: badpath indication during prefetching

Again, once a request has been sent to the Mbox, it cannot be cancelled. However, unsent requests in the Prefetch Subsection pipeline are dropped if a badpath indication is received for the same TPU as the prefetch requests. This allows new prefetch requests on the goodpath to proceed in the Mbox and/or Cbox without having to stall behind the badpath ones. If a badpath indication is received for a TPU having valid Capture Register state, that state is invalidated.

3.8.2.3 Fill

The Fill Section of the IFU contains the circuitry between the Cbox and the ICache for the predecoding and parity generation of instructions returning from the SCache or memory system. A simplified block diagram of the Fill Section appears in the figure below. The Cbox initiates the transfer of instructions to the IFU by supplying the early_warning_ptr corresponding to the request for those instructions. This pointer is used for probing two arrays in the Demand Subsection. The Demand Array Probe determines if the returning instructions are non-piggybacked demand requests for any of the TPUs, while the Entry Array Probe looks up the VA and tag data stored earlier for this request. This data, combined with the returning instructions themselves, are fed into the Predecode Br_Offset Gen and Parity Gen boxes to determine predecode bits, branch target offsets, and parity. The aggregation of these bits, combined with the tag and instruction bits, is called the Fill Packet.

Figure 3–8 Instruction Fill Unit (IFU) Fill Section



The branch offset calculations calculate a portion of the target address for both conditional and unconditional branches.

Carry predecodes are generated during branch targe precalculation. If an integer or floating-point conditional branch or unconditional branch is detected, the branch target is precalculated and the displacement field is replaced with the target as follows.

The overflow predecode bit is calculated as follows, where the circumflex (^) represents an XOR operation:

```
[(PC <21:2> + 1) + I<19:0>] ^ I<20>
```

The increment predecode bit is calculated as follows, where the increment is to the next address that falls on an 8-instruction boundary:

```
[(PC <21:2> + 1) + I<19:0>] + 8
```

Because the displacement is overwritten by the target address when the branch is stored and because the displacement field is 21 bits long, only the lower 20 bits of the result are calculated. By leaving the sign bit <20> intact and including the overflow bit in the Icache to hold the carry-out, the rest of the addition can be performed when the PC's are calculated. Because the sign and overflow bits can both be 0 or 1, the high bits of the target can be incremented or decremented or unchanged.

The predecodes are split between those needed in the Ibox and those passed further down the pipeline.

3.8.2.3.1 Predecode Bit Generation

Table 3–17 shows an overview of the predecode bits that are generated in the Instruction Fill Unit (IFU). In the table, the last column shows the Pbox predecode bits as A, B, C, and D, and the Ibox predecode bits as UE, CB, P2, P3, CM, and MA.

Table 3–17 is sorted according to opcode value. In the table:

- The first column lists the instructions.
- The second column lists each of the 23 possible Pbox-assigned instruction types.
- The third column lists the opcode for each instruction or group of instructions.
- The fourth column lists the function field bits in the instruction that the IFU uses to determine the instruction type.
- The fifth column lists the predecodes that the IFU generates. The Pbox assigns the instruction type (column two) according to the EDCBA predecode bits. Similarly, the Ibox assigns the other encoding bits for the control flow instructions, described in Table 3–18.

Table 3–17 Predecode Bits Defined by the Ibox Instruction Fill Unit

Instruction	Type[1]	Opcode Bits: 31— 26	Function Field Bits:[2] 15 14 13 12 11 10 9 8 7 6 5 P	Predecode Bits:[3] EDCBA	UE	CB	P2	P3	CM	MA
CALL PAL	XXP	000000	- - - - - - - - - - - -	00010	1	1	1	1	0	0
RES	XXX	000001	- - - - - - - - - - - -	01001	0	0	0	0	0	0
RES	XXX	000010	- - - - - - - - - - - -	01001	0	0	0	0	0	0

Table 3–17 Predecode Bits Defined by the Ibox Instruction Fill Unit (Continued)

Instruction	Type[1]	Opcode Bits: 31— 26	Function Field Bits:[2] 15 14 13 12 11 10 9 8 7 6 5 P	Predecode Bits:[3] EDCBA	UE	CB	P2	P3	CM	MA
RES	XXX	000011	– – – – – – – – – – – –	01001	0	0	0	0	0	0
RES	XXX	000100	– – – – – – – – – – – –	01001	0	0	0	0	0	0
RES	XXX	000101	– – – – – – – – – – – –	01001	0	0	0	0	0	0
RES	XXX	000110	– – – – – – – – – – – –	01001	0	0	0	0	0	0
P*xxxx*[4]	XXX	000111	– – – – – – – – – – – –	01001	0	0	0	0	0	0
LDA	XII	001000	– – – – – – – – – – – –	11011	0	0	0	0	0	0
LDAH	XII	001001	– – – – – – – – – – – –	11011	0	0	0	0	0	0
LDBU	SII	001010	– – – – – – – – – – – –	10000	0	0	0	0	0	0
LDQ_U	SUI	001011	– – – – – – – – – – – –	11001	0	0	0	0	0	0
LDWU	SII	001100	– – – – – – – – – – – –	10000	0	0	0	0	0	0
STW	IIS	001101	– – – – – – – – – – – –	11111	0	0	0	0	0	0
STB	IIS	001110	– – – – – – – – – – – –	11111	0	0	0	0	0	0
STQ_U	IIS	001111	– – – – – – – – – – – –	11111	0	0	0	0	0	0
INTA	III	010000	– – – 0 – – – – – – – –	00100	0	0	0	0	0	0
INTA	IXI	010000	– – – 1 – – – – – – – –	00101	0	0	0	0	0	0
INTL	III	010001	– – – 0 – – – – 0 – – –	00100	0	0	0	0	0	0
INTL	IXI	010001	– – – 1 – – – – 0 – – –	00101	0	0	0	0	0	0
INTL	III	010001	– – – 0 – – – 1 1 – – –	00100	0	0	0	0	0	0
INTL	IXI	010001	– – – 1 – – – 1 1 – – –	00101	0	0	0	0	0	0
CMOV*x*	III	010001	– – – 0 – – – 0 1 – – –	00100	0	0	0	0	1	0
CMOV*x*	IXI	010001	– – – 1 – – – 0 1 – – –	00101	0	0	0	0	1	0
MA	III	010001	– – – 0 1 0 – 0 0 – – –	00100	0	0	0	0	0	1
MA	IXI	010001	– – – 1 1 0 – 0 0 – – –	00101	0	0	0	0	0	1
INOP	III	010001	– – – 0 0 1 – 0 0 – – –	00100	0	0	0	0	0	0
INOP	IXI	010001	– – – 1 0 1 – 0 0 – – –	00101	0	0	0	0	0	0
INTS	III	010010	– – – 0 – – – – – – – –	00100	0	0	0	0	0	0
INTS	IXI	010010	– – – 1 – – – – – – – –	00101	0	0	0	0	0	0
INTM	III	010011	– – – 0 – – – – – – – –	00100	0	0	0	0	0	0
INTM	IXI	010011	– – – 1 – – – – – – – –	00101	0	0	0	0	0	0
FLTS	FFF	010100	– – – – – – – 1 – – – –	11100	0	0	0	0	0	0
ITOF*x*	IXF	010100	– – – – – – – 0 – – – –	00111	0	0	0	0	0	0
FLTV	FFF	010101	– – – – – – – – – – – –	11100	0	0	0	0	0	0
FLTI	FFF	010110	– – – – – – – – – – – –	11100	0	0	0	0	0	0

Table 3–17 Predecode Bits Defined by the Ibox Instruction Fill Unit (Continued)

Instruction	Type[1]	Opcode Bits: 31— 26	Function Field Bits:[2] 15 14 13 12 11 10 9 8 7 6 5 P	Predecode Bits:[3] EDCBA	UE	CB	P2	P3	CM	MA
FCMOV*x*	FFF	010111	– – – – – – – 0 1 – – – –	11100	0	0	0	0	1	0
CPYSx	FFF	010111	– – – – – – – 0 0 0 – – –	11100	0	0	0	0	0	0
MT_FPCR	FFC	010111	– – – – – – – 0 0 1 – 0 –	11110	0	0	0	0	0	0
MF_FPCR	FFF	010111	– – – – – – – 0 0 1 – 1 –	11100	0	0	0	0	0	0
CVTxx	FFF	010111	– – – – – – – 1 – – – – –	11100	0	0	0	0	0	0
FNOP	FFF	010111	– – – – – – – 0 0 0 0 0 –	11100	0	0	0	0	0	0
TRAPB	XXX	011000	0 0 – – – 0 – – – – – – –	01001	0	0	0	0	0	0
EXCB	XXX	011000	0 0 – – – 1 – – – – – – –	01001	0	0	0	0	0	0
MB	XXX	011000	0 1 – – 0 0 – – – – – – –	01001	0	0	0	0	0	0
(MB)	XXX	011000	0 1 – – 1 – – – – – – – –	01001	0	0	0	0	0	0
WMB	IIX	011000	0 1 – – 0 1 – – – – – – –	01111	0	0	0	0	0	0
FETCH	XXX	011000	1 0 0 – – – – – – – – – –	01001	0	0	0	0	0	0
FETCH_M	XXX	011000	1 0 1 0 – – – – – – – – –	01001	0	0	0	0	0	0
RPCC	XIY	011000	1 1 0 – – – – – – – – – –	01011	0	0	0	0	0	0
Rx	XXN	011000	1 1 1 – 0 – – – – – – – –	00011	0	0	0	0	0	0
xCB	IIX	011000	1 1 1 0 1 – – – – – – – –	01111	0	0	0	0	0	0
WH64x	IIX	011000	1 1 1 1 1 – – – – – – – –	01111	0	0	0	0	0	0
LD*x*_ARM	SII	011000	1 0 1 1 0 – – – – – – – –	10000	0	0	0	0	0	0
QUIESCE	IIX	011000	1 0 1 1 1 – – – – – – – –	01111	0	0	0	0	0	0
HW_MFPR	RXI	011001	– – – – – – – – – – – – –	00110	0	0	0	0	0	0
JMP	XII	011010	0 0 – – – – – – – – – – 0	11011	1	0	0	0	0	0
JMP	XII	011010	0 0 – – – – – – – – – – 1	11011	0	0	0	0	0	0
RET	XII	011010	1 0 – – – – – – – – – – –	11011	1	0	1	0	0	0
JSR	XII	011010	0 1 – – – – – – – – – – –	11011	1	1	0	0	0	0
JCR	XII	011010	1 1 – – – – – – – – – – –	11011	1	1	1	0	0	0
HW_LD	SII	011011	– – – – – – – – – – – – –	10000	0	0	0	0	0	0
INTV	III	011100	– – – 0 0 – – – – – – – –	00100	0	0	0	0	0	0
INTV	IXI	011100	– – – 1 0 – – – – – – – –	00101	0	0	0	0	0	0
INTV	III	011100	– – – 0 1 0 – – – – – – –	00100	0	0	0	0	0	0
INTV	IXI	011100	– – – 1 1 0 – – – – – – –	00101	0	0	0	0	0	0
INTV	III	011100	– – – 0 1 1 0 – – – – – –	00100	0	0	0	0	0	0
INTV	IXI	011100	– – – 1 1 1 0 – – – – – –	00101	0	0	0	0	0	0
FTOI*x*	FXI	011100	– – – – 1 1 1 – – – – – –	01110	0	0	0	0	0	0
HW_MTPR	RIW	011101	– – – – – – – – – – – – –	10110	0	0	0	0	0	0

Table 3–17 Predecode Bits Defined by the Ibox Instruction Fill Unit (Continued)

Instruction	Type[1]	Opcode Bits: 31— 26	Function Field Bits:[2] 15 14 13 12 11 10 9 8 7 6 5 P	Predecode Bits:[3] EDCBA	UE	CB	P2	P3	CM	MA
IFETCHB	XXX	011110	– – – – – – – – – – – – – – –	01001	1	0	1	1	0	0
HW_ST	IIS	011111	– – – – – – – – – – – – – – –	11111	0	0	0	0	0	0
LDF	SIF	100000	– – – – – – – – – – – – – – –	11000	0	0	0	0	0	0
LDG	SIF	100001	– – – – – – – – – – – – – – –	11000	0	0	0	0	0	0
LDS	SIF	100010	– – – – – – – – – – – – – – –	11000	0	0	0	0	0	0
LDT	SIF	100011	– – – – – – – – – – – – – – –	11000	0	0	0	0	0	0
STF	FIS	100100	– – – – – – – – – – – – – – –	11101	0	0	0	0	0	0
STG	FIS	100101	– – – – – – – – – – – – – – –	11101	0	0	0	0	0	0
STS	FIS	100110	– – – – – – – – – – – – – – –	11101	0	0	0	0	0	0
STT	FIS	100111	– – – – – – – – – – – – – – –	11101	0	0	0	0	0	0
LDL	SII	101000	– – – – – – – – – – – – – – –	10000	0	0	0	0	0	0
LDQ	SII	101001	– – – – – – – – – – – – – – –	10000	0	0	0	0	0	0
LDL_L	SII	101010	– – – – – – – – – – – – – – –	10000	0	0	0	0	0	0
LDQ_L	SII	101011	– – – – – – – – – – – – – – –	10000	0	0	0	0	0	0
STL	IIS	101100	– – – – – – – – – – – – – – –	11111	0	0	0	0	0	0
STQ	IIS	101101	– – – – – – – – – – – – – – –	11111	0	0	0	0	0	0
STL_C	IIL	101110	– – – – – – – – – – – – – – –	00000	0	0	0	0	0	0
STQ_C	IIL	101111	– – – – – – – – – – – – – – –	00000	0	0	0	0	0	0
BR	XXI	110000	– – – – – – – – – – – – – – –	00001	1	0	0	1	0	0
FBEQ	FXX	110001	– – – – – – – – – – – – – – 0	01100	0	1	0	1	0	0
FBEQ	FXX	110001	– – – – – – – – – – – – – – 1	01100	0	0	0	0	0	0
FBLT	FXX	110010	– – – – – – – – – – – – – – 0	01100	0	1	0	1	0	0
FBLT	FXX	110010	– – – – – – – – – – – – – – 1	01100	0	0	0	0	0	0
FBLE	FXX	110011	– – – – – – – – – – – – – – 0	01100	0	1	0	1	0	0
FBLE	FXX	110011	– – – – – – – – – – – – – – 1	01100	0	0	0	0	0	0
BSR	XXI	110100	– – – – – – – – – – – – – – –	00001	1	1	0	1	0	0
FBNE	FXX	110101	– – – – – – – – – – – – – – 0	01100	0	1	0	1	0	0
FBNE	FXX	110101	– – – – – – – – – – – – – – 1	01100	0	0	0	0	0	0
FBGE	FXX	110110	– – – – – – – – – – – – – – 0	01100	0	1	0	1	0	0
FBGE	FXX	110110	– – – – – – – – – – – – – – 1	01100	0	0	0	0	0	0
FBGT	FXX	110111	– – – – – – – – – – – – – – 0	01100	0	1	0	1	0	0
FBGT	FXX	110111	– – – – – – – – – – – – – – 1	01100	0	0	0	0	0	0
BLBC	IXX	111000	– – – – – – – – – – – – – – 0	01000	0	1	0	1	0	0

Table 3–17 Predecode Bits Defined by the Ibox Instruction Fill Unit (Continued)

Instruction	Type[1]	Opcode Bits: 31—26	Function Field Bits:[2] 15 14 13 12 11 10 9 8 7 6 5 P	Predecode Bits:[3] EDCBA	UE	CB	P2	P3	CM	MA
BLBC	IXX	111000	– – – – – – – – – – – 1	01000	0	0	0	0	0	0
BEQ	IXX	111001	– – – – – – – – – – – 0	01000	0	1	0	1	0	0
BEQ	IXX	111001	– – – – – – – – – – – 1	01000	0	0	0	0	0	0
BLT	IXX	111010	– – – – – – – – – – – 0	01000	0	1	0	1	0	0
BLT	IXX	111010	– – – – – – – – – – – 1	01000	0	0	0	0	0	0
BLE	IXX	111011	– – – – – – – – – – – 0	01000	0	1	0	1	0	0
BLE	IXX	111011	– – – – – – – – – – – 1	01000	0	0	0	0	0	0
BLBS	IXX	111100	– – – – – – – – – – – 0	01000	0	1	0	1	0	0
BLBS	IXX	111100	– – – – – – – – – – – 1	01000	0	0	0	0	0	0
BNE	IXX	111101	– – – – – – – – – – – 0	01000	0	1	0	1	0	0
BNE	IXX	111101	– – – – – – – – – – – 1	01000	0	0	0	0	0	0
BGE	IXX	111110	– – – – – – – – – – – 0	01000	0	1	0	1	0	0
BGE	IXX	111110	– – – – – – – – – – – 1	01000	0	0	0	0	0	0
BGT	IXX	111111	– – – – – – – – – – – 0	01000	0	1	0	1	0	0
BGT	IXX	111111	– – – – – – – – – – – 1	01000	0	0	0	0	0	0

[1] The predecode type (or logic group) is described in Section A.2.

[2] In the function field bit listing, *P* represents the physical bit, described below.

[3] See Table 3–18 for information about predecode bits other than EDCBA.

[4] Paired single-precision floating-point instructions.

3.8.2.3.2 Predecode Bits for Control Flow Instructions

Table 3–18 describes the meaning for those predecode bits that are generated by the IFU for control flow instruction processing. In the table:

- UE is an unconditional exit and CB is a conditional branch. The UE and CB predecodes are used by the branch predictor to quickly determine the exit point of the two fetch slots.

  P2 is popstack and normally means to pop the return stack. P3 (or branch) normally means B*xx*. P2 and P3 are used to determine how the return stack and jump predictor outputs are used. The following attributes can be determined for all 16 fetched instructions during the A phase of I3 when the branch predictor is determining the exit point:

```
JPRED    =  UE & !P2 & !P3
POP      =  P2 & !P3
PUSH     =  UE & CB
TBR      =  !P2 & P3
CPL      =  CB & P2 & P3
IFETCHB  =  !CB & P2 & P3
```

For detailed information, see Section 3.7

- For "legacy" CMOV/FCMOV instructions (see Section 2.11.2), a set CM bit causes the Collapsing Buffer to create a CMOV2 instruction by making a new instruction chunk. Legacy CMOV/FCMOV instructions are always the first instruction in the map chunk.
- The MA bit is set when an XOR (11.40) instruction with destination R31 is detected as the final instruction in either half-block of eight instructions received by the IFU. When MA is set and the chunk is fetched from the Icache, the Collapsing Buffer starts a new map chunk that begins with the current fetch chunk. See also Section 2.11.3 for more information.
- The physical bit, *P*, in the function field bits column indicates that no address translation was performed when fetching instructions. When set, the VA field in the TAG represents the actual PA from which the instructions were fetched, and not a translation.

Table 3–18 Ibox Predecode Bit Summary

UE	CB	P2	P3	CM	MA	Meaning
0	0	0	0	0	0	Fall through (integer and floating-point conditional branch in PALmode, physical bit = 1)
0	0	0	0	1	0	Fall through and Collapsing Buffer starts a new map chunk to begin with a CMOV2 instruction
X	X	X	X	X	1	Collapsing Buffer starts a new map chunk at current fetch chunk
0	1	0	1	0	0	Integer and flaoting-point conditional branch (physical bit = 0)
1	0	0	0	0	0	Jump
1	0	0	1	0	0	Unconditional branch
1	0	1	0	0	0	Return (pops the return stack)
1	0	1	1	0	0	IFETCHB, (stops thread, next PC = PC + 4)
1	1	0	0	0	0	JSR (pushes return stack)
1	1	0	1	0	0	BSR (pushes return stack)
1	1	1	0	0	0	JSR_COROUTINE (pops and pushes return stack)
1	1	1	1	0	0	CALL_PAL (pushes return stack)
0	0	0	1	X	X	Not used - do not care
0	0	1	0	X	X	Not used - do not care
0	0	1	1	X	X	Not used - do not care
0	1	0	0	X	X	Not used - do not care
0	1	1	0	X	X	Not used - do not care
0	1	1	1	X	X	Not used - do not care

3.8.2.3.3 Fill Data Routing

A few simple rules govern the routing of information in the Fill Section. First, the Cbox returns its data in 16-instruction chunks, whether or not the request hit the Scache. Next, if the Demand Array Probe indicates that the returning instructions are for a non-piggybacked demand miss, the wake_tpu signal for the corresponding thread is activated, which is read by the FTC. Finally, the ICache is designed to give writes priority over reads, so buffering of writes is not necessary.

The early warning signals sent in cycle C10 are latched in cycle IX. The array probes occur in IY, and the fill_inst instruction bus latches its C12 signals on the IZ edge. The discard_fill and dbl_ecc_err error signals are used in IZ. If the former is true, all processing for this fill is terminated, and all IFU state will behave as if the early_warning_ptr had never been active for this fill. At some later time, the Cbox will again try to complete the fill for this request. The discard_fill signal covers a number of late-kill cases, including single-bit error detection, that occur too late to affect sending of the early_warning_ptr.

If discard_fill is false, but dbl_ecc_err is true, the fill proceeds as normal, but with the resultant Fill Packet being written into the ICache with its ecc_uncor bit set. Any reads of this ICache line will force a late exception. This method allows for fast processing of double-bit errors when the machine is in PAL mode, because the normal handling via Cbox interrupt is not possible in PALmode.

3.9 Checkpoint Unit

The checkpoint table, as the name implies, serves as a repository for important information flowing through the pipeline every clock cycle. This information is later used for (a) restoring state when restarting on an exception and (b) for training predictors in the Ibox.

The checkpoint table plays a pivotal role in restarting the pipeline for all exceptions except those that are specific to the Ibox such as those caused due to line or way mispredictions. Specifically, the checkpoint table handles only restarts for instructions that have been mapped and assigned an INum by the Pbox. The class of exceptions that is handled by the checkpoint table is also known as the "post-map" exceptions. Another important role played by the checkpoint table is to provide sufficient information at instruction retire time for training the branch and jump predictor. The information stored in the checkpoint table is also leveraged to allow mispredicted jumps to be identified as well as to generate the return address whenever a subroutine call is made.

Effectively, the checkpoint table acts as a link between the pre-mapped and post-mapped world of instructions. Before the mapping is performed, an instruction is identified using its address. Once the instructions are mapped and dispatched from the Ibox to the Pbox, the INum associated with the instruction becomes its sole identifier. However, the address of an instruction may be needed occasionally during its lifetime. This is especially true when an instruction restarts on an exception or a return address is needed by the Ebox to be pushed into a stack register when a subroutine call is executed. The checkpoint table enables such operations to be performed with its ability to reverse-map the INum of an instruction to its address using the information stored in it.

It must be noted that the amount of information that needs to be checkpointed for restarts and training is non-trivial. However, due to area constraints on the die, the information cannot be stored in a naïve fashion. Hence, several optimizations are performed to condense the information for reduced storage without losing any details.

The following sections provide additional details of the checkpoint table.

3.9.1 Checkpoint Table Components

The checkpoint table consists of a pre-map and post-map table. The pre-map table reflects the instruction buffer and stores information on a fetch-slot basis while the post-map table stores information on a map-chunk basis. The post-map table forms the core of the check-pointing mechanism. The pre-map table acts only as a temporary store to hold data until the collapsing buffer creates a new map-chunk from fetch slots in the instruction buffer. Once this operation is completed, the information for the collapsed fetch slots flows from the pre-map table to the post-map table and is stored in a collapsed form to reduce storage requirements.

As with the instruction buffer, the pre-map table consists of 16 entries per thread for a total of 64 entries. Information corresponding to each of the two slots that may be fetched every cycle is written into the pre-map table using the same index that is used to write into the instruction buffer. Table 3–19 lists the different fields that are stored for each slot along with the producer of that information. Since a fetch slot cannot have a valid jump as well as a return instruction, the jump and return predictions share single field. The appropriate data is written based on the type of the exit instruction. For more information on each specific field, please refer to the appropriate producer section.

Table 3–19 Fields in a Pre-Map Table Entry

Producer	Information
PC calc logic	PC<51:5>, PC<0> (palmode bit)
Jump predictor	Jghist<35:0>,
Jump or Return predictor	Jump or return prediction<51:2>
Branch Predictor	Lghist<23:0>. Shift distance<2:0>, No shift
Branch Predictor	Bank<6:5>, Next Bank<6:5>, Next to Next Bank <6:5>
Branch Predictor	Previous index <6:5>, Next to Previous index <6:5>
Branch Predictor	Prediction entries: G0<7:0>, G1<7:0>, CH<7:0>, BM<7:0>
Branch Predictor	Jump, Push, Pop exit attributes and sequential exit flag
Branch Predictor	Conditional branch attributes<7:0>
Return Predictor	Nalloc<5:0>, Tos<5:0>, Ptos<5:0>

The post-map table contains 32 entries, each of which corresponds to an in-flight map chunk. The table is indexed using the map chunk INum. Most of the information is stored on a map-chunk basis though some information needs to be stored on a per-slot or per-instruction basis. The information stored in the post-map table mostly originates either from the pre-map table or from the collapsing buffer. Since up to two fetch slots may be collapsed to create a map-chunk, the information stored in one entry of the post-

map table spans the information from two adjacent entries of the pre-map table. However, the information for the two fetch slots is not stored as such. Instead, it is collapsed such that the storage space is vastly reduced without losing any information.

Most of the fields in a post-map table entry are written during map time. However, there are a few fields that are not created until instruction execution time or when the Pbox signals a kill due to some exception.

We now list the different fields for an entry in the post-map table. Table 3–20 lists those fields that store a collapsed form of the fields read from the pre-map table for two adjacent slots: slot A and slot B. Table 3–21 lists those fields that are stored in the same format (for each slot) as they are read from the pre-map table. Table 3–22 lists the remaining fields that do not use any pre-map table entries and are written directly using information provided by the collapsing buffer. Finally, Table 3–23 lists the fields that are created during execution or kill time. We also provide a brief description for some of the fields that include details on how the collapsing is performed.

Table 3–20 Collapsed fields Stored Into a Post-map Table Entry at Map Time

ID	Data from Pre-Map Table		Collapsed fields in Post-Map Table Entry
1	Slot A LGhist<23:0>	Slot B LGhist<23:0>	LGhist<24:0>
2	Slot A ShiftDist<2:0>	Slot B ShiftDist<2:0>	ShiftDist<3:0>
3	Slot A Bank<6:5> Slot A Bank_next<6:5> Slot A Bank_next_next<6:5>	Slot B Bank<1:0> Slot B Bank_next<6:5> Slot B Bank_next_next<6:5>	Bank<6:5> Bank_next<6:5> Bank_next_next<6:5> Bank_next_next_next<6:5>
4	Slot A prev_index<6:5> Slot A prev_index_next<6:5>	Slot B prev_index<6:5> Slot B prev_index_next<6:5>	Prev_index<6:5> Prev_index_next<6:5> Prev_index_next_next<6:5>
5	Slot A Nalloc<5:0> Slot A Tos<5:0> Slot A Ptos<5:0>	Slot B Nalloc<5:0> Slot B Tos<5:0> Slot B Ptos<5:0>	Slot A Nalloc<5:0> Slot B Nalloc<5:0> Slot A Tos<5:0> Slot B Tos<5:0> Slot B Ptos<5:0>
6	Slot A noshift	Slot B noshift	Slot A noshift

Table 3–21 shows the fields in the post-map table entry that is maintained for each slot in a map-chunk and is directly transferred from the pre-map table at map time.

Table 3–21 Post-Map Table Entry Fields

ID	Fields in the Post-Map Table entry
7	PC<51:5>, PC<0> (palmode bit), PC+4<15:5>
8	Jump, Push, Pop exit attributes and sequential exit flag
9	Branch prediction entries: G0<7, 0>, G1<7, 0>, CH<7, 0>, BM<7, 0>
10	Conditional branch attributes<7:0>
11	Jghist<35:0>, Jump prediction<51:2>

Table 3–22 Fields that are Available from Collapsing Buffer at Map Time

ID	Fields in the Post-Map Table entry
12	Alternate PC<21:0> (8 in all; 1 for each map-chunk instruction)
13	Store Set ID<4:0> (8 in all; 1 for each map-chunk instruction)
14	Slot Mask<7:0>
15	Map Chunk information: length<2:0>, slot 0 start position<2:0>, slot 1 start position<2:0>, slot 0 length<2:0>

Table 3–23 Fields in Post-Map Table Entry That are Created During Execute (*E*) and Kill Time (*K*)

ID	Fields in the Post-Map Table entry
16	Jump Target<51:0>, Jump Target Valid (*Execute*)
17	Kill location <2:0> Kill Valid (*Kill*)

Notes for Tables 3–20 through 3–23:

For some of the fields mentioned above, we give a brief description that includes details on how information is collapsed before it is written into the post-map table. Note that we use the ID in the above tables to describe the corresponding field.

- Lghist for Slot B can have at most one new bit added to it with the rest of the bits overlapping with that of Slot A. To determine if there was indeed a new bit added to slot B's lghist, we use the newest Shift Distance bit for Slot B. The collapsed Lghist is created as follows:

```
If Slot B ShiftDist<0>
LGhist<24:0> = CONCAT (Slot A LGhist<23:0>, Slot A Lghist<0>)
      Else
Lghist<24:0> = CONCAT (0, Slot A Lghist<23:0>)
```

- The Shift Distance for Slot B has one new bit while the other two bits overlap with that of Slot A.
- ShiftDist<3:0> = CONCAT (Slot A ShiftDist<2:0>, Slot B ShiftDist<0>)
- The two successors to *Slot A Bank* are exactly the same as *Slot B Bank* and its successor. Hence, we need to store only 4 out of the 6 bank identifier fields.
- As with the bank identifiers, the next *prev_index* of Slot A is the same as that of *Slot B prev_index*. So we store only 3 out of the 4 fields.
- Ptos (previous top of stack) is used solely when restarting after an instruction that pops the return stack. If Slot A had such an instruction, Slot B's top of stack would indeed be slot A's Ptos. Hence, there is no need to store the PTos for slot A.

- The no shift bit, which prevents the shift distance bits from being modified more than once for the same fetch slot when it is restarted (on an exception), is relevant only for the first slot (see branch predictor section for more details).
- The low PC bits <4:2> are created only on a need basis for a particular instruction in one of the fetch slots comprising the map chunk. The *PC+4* field is not present in the pre-map table and is created on the fly from the corresponding PC bits. Pre-calculating this field is necessary for restarting the line predictor latches with a new index as fast as possible.
- For conditional branches that are predicted as not taken, we need to store the alternate address (alternate PC) to handle mispredicts. This address would be used on a restart from a mispredicted not-taken branch. Since a branch instruction can occur in any position of the map chunk, provision must be for storing up to 8 alternate addresses.
- As with branches, a load or store instruction may occur in any position in the map-chunk. Hence, we need to provide storage for all instruction positions in the map-chunk.
- The slot mask specifies whether a particular instruction originated from slot A or slot B.
- Slot 0 length is not directly available from the collapsing buffer. It is calculated using the slot mask that is provided by the collapsing buffer.
- The Jump Target Valid bit enables two jump instructions each belonging to slot A and slot B to share the same location for storing the actual target on a jump misprediction. The following section provides more details on the sharing mechanism.
- To ease implementation, both the pre-map and post-map tables are partitioned such that a particular partition resides close to the check-pointed component. For instance, in the partition residing close to the branch predictor, we need to store only those fields that are relevant to the branch predictor such as Lghist, shift distance, no shift, prediction entries etc. while fields such as store set identifiers and Jump predictions need not be.

3.9.1.1 Checkpoint Table Functions

As mentioned earlier, the fields in the checkpoint table are not only written during instruction map time but also during the execution phase as well as when instructions are killed due to an exception.

When the Ebox executes a jump instruction, it forwards the actual target of the jump to the checkpoint table so as to validate the jump prediction. The checkpoint table accesses the corresponding entry in the post-map table using the INum that is provided by the Ebox to access the predicted jump address. If a mismatch occurs between the true target and the predicted address, the checkpoint table signals a jump mispredict to the Ebox. At the same time, it stores the true target into the table. This target value will eventually be used for restarting the pipeline as well as for training the jump predictor. The *jump valid* bit is also set on a jump mispredict when the correct target is stored. Since an earlier exception overrides a younger exception, a mispredicted jump in slot A can always store its true target while a mispredicted jump in slot B may do so only when a jump instruction in slot A has not already mispredicted.

Occasionally, the Ebox requires the return address that needs to be saved in the stack register when executing a subroutine call. The checkpoint table uses the INum provided to find the associated PC of the subroutine call instruction and sends the PC of the subsequent instruction (PC+4) to the Ebox.

When the Ebox eventually executes the "return" instruction in the subroutine, execution is redirected to the address that was provided by the checkpoint table. Note that the return address also needs to be validated. The description given for jumps for signaling mispredicts is also true for "return" instructions. This is because the address predicted by the return and jump predictors share the same field as only a jump or a return instruction can be valid in a fetch slot.

3.9.1.1.1 Restarting on an exception

The checkpoint table is responsible for restarting the pipeline on an exception by providing the *line predictor* and *PC calc* logic with the new address. An exception may occur appear through the exception funnel (E-funnel) from the Pbox or on the fast-path used for early signaling of branch mispredictions.

The E-funnel exceptions take priority over the fast-path exceptions. The information available to the checkpoint table from the E-funnel includes the type of exception and the exception INum. The checkpoint uses the exception type and the INum to access the post-map table to get the appropriate restart address. The slot mask (Table 3–22) lets us determine the slot in which the misprediction occurred. With this information, we can choose the appropriate address from a set of addresses that is stored on a fetch-slot basis (PC). The low bits of the exception INum helps us to choose an address from a set of 8 addresses stored on an instruction basis in the map-chunk (Alternate PC). Remember that the low bits<4:2> of the PC are not stored in the post-map table. However, by using the map chunk information (Table 3–22) and the position of the instruction in the map chunk, the low bits of the restart address can be easily determined.

The restart may cause control to be transferred to PAL code in which case the checkpoint table also needs to provide the address to which control has to resume after return from PAL code. The PAL starting address itself is created by adding the offset provided through the exception funnel to the base address that is read from a PAL base register.

If no exceptions are present in the E-funnel, the fast-path, which is used for early resolution of conditional branch mispredictions, is checked for the presence of an exception. Information on whether the conditional branch instruction was a mispredicted taken or not-taken type as well as its INum is also available on the fast path.

Table 3–24 lists the different restart scenarios that are handled by the checkpoint table. The different types of restart addresses mentioned for the non-PAL exceptions are available in the post-map table. Note that the complete restart address is needed only by the *PC calc* logic while just the low bits of the restart address <14:2> are needed by the line predictor latches but a cycle earlier than *PC calc*. Due to timing constraints in the implementation, the low bits<14:5> of the incremented PC (PC+4) are stored apriori in the post-map table. This would be used whenever the restart address is PC+4 rather than calculating the value at the time of restart.

Table 3–24 Exception Types and Restart Address

Exception	Restart Address	Return Address for PAL
Mispredicted not-taken Conditional branch, IFETCHB	PC+4	N.A
Mispredicted taken Conditional branch	Alternate PC	N.A
Mispredicted jumps	Jump Target	N.A
Replay, Load Store order violation	PC	N.A
DTB Miss	PALbase + offset	PC
Unalign, Write FPCR, Integer/FP Trap	PALbase + offset	PC+4

3.9.1.1.2 Restoring Predictor States

In addition to providing the restart address, the checkpoint table also needs to restore the states of the different predictors in the Ibox namely, the branch, jump and return predictors. Due to the complex nature of the branch history bits, the control for restoring the state is non-trivial. Table 3–25 shows how we create the initial lghist and shift distance from the post-map entry based on whether the restart occurs in slot A or slot B. Table 3–26 details the complete restoration process.

Table 3–25 Creating Slot-Based Predictor States From Mapped Information in the Post-Map Table

	LGHIST	SHIFT DISTANCE

Slot_A	if (MappedShiftDist<0>)
	  Slot_Ghist = MappedGhist<24:1>
	else
	  Slot_Ghist = MappedGhist<23:0>
		Slot_ShiftDist = MappedShiftDist<3:1>

Slot_B	Slot_Ghist = MappedGhist<23:0>	Slot_ShiftDist = MappedShiftDist<2:0>

Table 3–26 shows.....

Table 3–26 Restoring Predictor States on a Restart

Type of Excepting instruction	LGHIST, SHIFT DISTANCE, NOSHIFT	JGHIST	NALLOC/TOS
Conditional Branch (Bxx)	Restart in 1st half of slot? Taken? Ghist = Slot_Ghist,1; ShiftDist =Slot_ShiftDist,1; NoShift = 0 Not taken & No valid Bxx insn after? Ghist = Slot_Ghist,0; ShiftDist =Slot_ShiftDist, 1; NoShift = 1 Restart in 2nd half? Taken? Ghist = Slot_Ghist,0; ShiftDist = Slot_ShiftDist,1; NoShift = 0 Not taken & No valid Bxx insn after? Ghist = Slot_Ghist,1; ShiftDist = Slot_ShiftDist, 1 NoShift = 1 (=0 if slot ends i.e PC_low<4:2> == 0x7)		
PUSH (BSR)	If valid insn before? If restart in 1st half || no valid Bxx insn in 2nd half? Ghist = Slot_Ghist,0; ShiftDist = Slot_ShiftDist, 1; NoShift = 0 else /* restart in 2nd half & valid Bxx insn in 2nd half */ Ghist = Slot_Ghist,1; ShiftDist = Slot_ShiftDist, 1; NoShift = 0		Nalloc = Slot_Nalloc + 1 Tos = Slot_Nalloc
Jump (JMP)	*** Same as for Push(BSR) ***	JGhist = Slot_JGhist<26:0>, (Jtarget<19:11> ^ Jtarget<10:2>)	

Table 3–26 Restoring Predictor States on a Restart

Type of Excepting Instruction	LGHIST, SHIFT DISTANCE, NOSHIFT	JGHIST	NALLOC/TOS
Push + Jump (JSR)	*** Same as for Push(BSR) ***	JGhist = Slot_JGhist<26:0>, (JTarget<19:11> ^ Jtarget<10:2>)	Nalloc = Slot_Nalloc + 1 Tos = Slot_Nalloc

Pop (RET)	*** Same as for Push(BSR) ***		Nalloc = Slot_Nalloc
			if (Slot A restart)
			Tos = Slot B Tos
			else
			Tos = Slot B Ptos

Pop + Push (JSR_COROUTINE)	*** Same as for Push(BSR) ***		Nalloc = Slot_Nalloc + 1 Tos = Slot_Nalloc
Any other instruction (restart would be "at" this instruction)	If no valid Bxx insn after & valid insn before? If restart in 1st half || no valid Bxx insn in 2nd half? Ghist = Slot_Ghist,0; ShiftDist = Slot_ShiftDist, 1; NoShift = 1 else /* restart in 2nd half & valid Bxx insn in 2nd half * Ghist = Slot_Ghist,1; ShiftDist = Slot_ShiftDist, 1; NoShift = 1		

Default State	Ghist = Slot_Ghist
	ShiftDist = Slot_ShiftDist
	if (Slot A restart)
	NoShift = NoShift_old (from post-map table)
	else
	NoShift = 0
		JGhist = Slot_JGhist	Nalloc = Slot_Nalloc Tos = Slot_Tos

3.9.1.1.3 Predictor Training

The checkpoint table is also used for training the branch and jump predictors. The jump predictor is trained only on a misprediction while the branch predictor is trained on both correct and incorrect predictions. The mispredict information is available in the kill field of the post-map table (Table 3–23).

The following state information is provided to the branch predictor for training each slot in the map chunk: lghist, shift distance, bank, previous index and prediction bits (Table 3–20). In addition, using the kill position and the map-chunk information (Table 3–22), the actual instructions retired in each slot are also provided. This includes information on whether there was a mispredict in any of the slot as well as the position in the map-chunk where the mispredict occurred. For more details on how the training is done, please refer to the branch predictor section.

As for the jump predictor training, the checkpoint table provides the true target to the jump predictor. It also uses the slot Jghist to calculate the index into the jump predictor array. The hash function for the index calculation is mentioned in the jump predictor section.

3.10 Ibox Interfaces

3.10.1 Pbox Interface

3.10.2 Qbox Interface

3.10.3 Ebox Interface

3.10.4 Mbox Interface

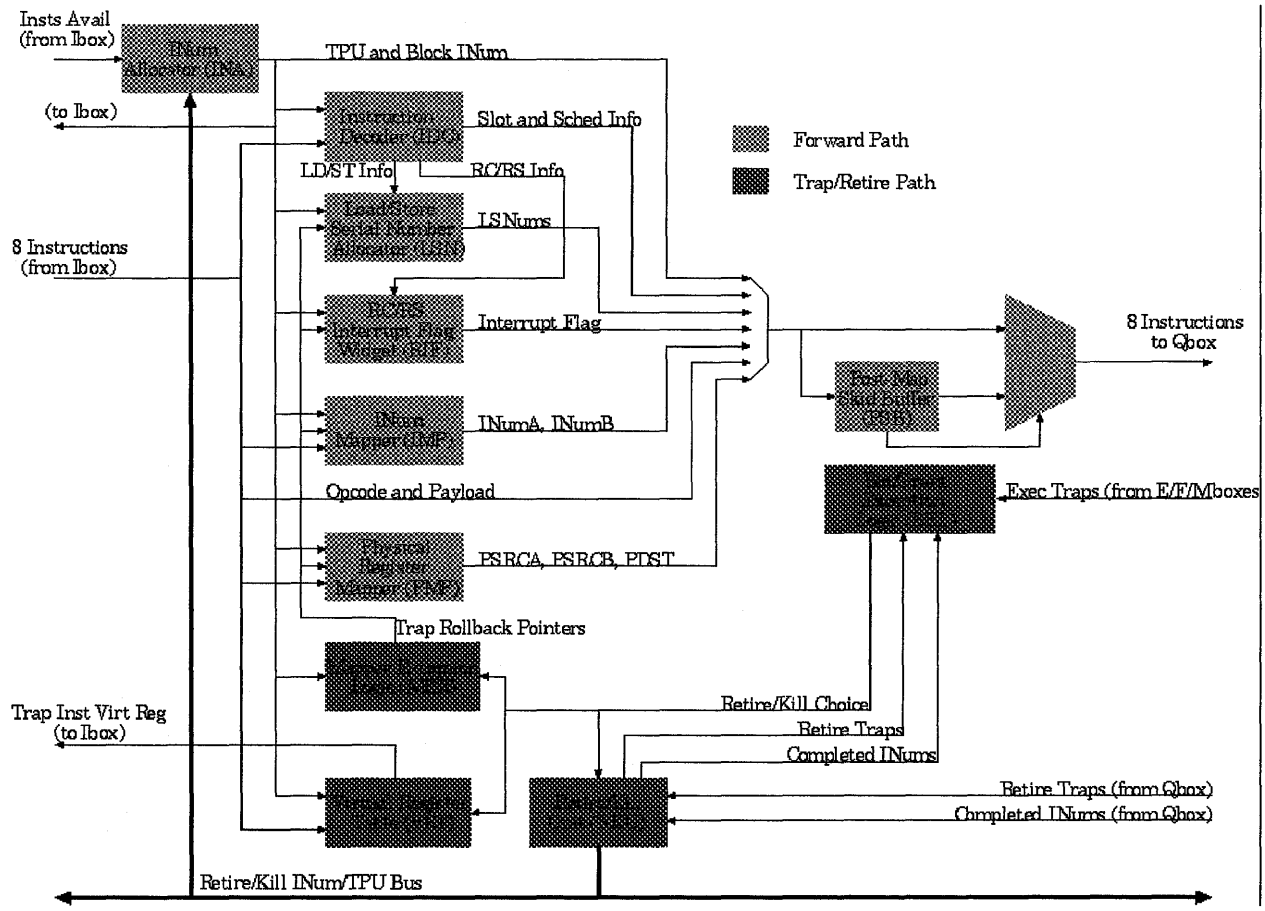
3.10.5 Cbox Interface

4

Dependency Mapper Unit — the Pbox

The Pbox processes instructions that are fetched by the Ibox. The Pbox assigns INums (instruction numbers) to the instructions, analyzes the data dependencies between instructions, and maps their architectural source and destination values into physical registers. The Pbox also maintains data structures that allow recovery of all relevant processor state that corresponds to the architectural state of the machine prior to any unretired instruction. This allows the processor to perform rapid trap recovery in the presence of branch mispredicts or other exception conditions. The Pbox passes the renamed instructions to the Qbox for scheduling and dispatch.

Figure 4–1 Pbox Block Diagram



The Pbox consists of the following components:

Table 4–1 Pbox Components

Name	Mnemonic	Description	Described in Section
Bid/Grant Exception Logic	BEL	Chooses which of the pending kills from all TPUs should be broadcast to the rest of the chip.	4.3.10
Instruction Decoder	IDC	Decodes each of the eight instructions that arrive in a cycle. The decoder is placed early in the pipe to aid slotting decisions and to provide inputs to the load/store flow control mechanisms and to the IPR interlock mechanisms	4.3.6
INum Allocator	INA	Allocates INums to new map blocks sent down by the Ibox. The INA also contains the Map Thread Chooser (see Section 4.3.3.3), which picks the next thread that will map instruction blocks and informs the Ibox	4.3.3
INum Mapper	IMP	Responsible for mapping source operand registers (VReg) into the INum of the last writer for the source operand	4.3.1
Load/Store Serial Number Allocator	LSN	Associates a sequential identifier with each load instruction, and a second identifier with each store instruction. These LNums and SNums are used to prevent deadlock and manage flow control into the Mbox load and store queues	4.3.7
Mapper Exception Logic	MEX	Rolls the IMP, PMP, LSN, and RIF state back to the trap point when the MEX is notified by the BEL of an exception	4.3.4
Memory Queue Allocation	MQA	Governs the allocation and deallocation of load queue (LQ) and store queue (SQ) chunks to memory instructions. Also controls the High-Water Mark (HWM) that is sent to the Qbox to regulate the issuing of loads and stores.	4.3.5
Physical Register Map	PMP	Allocates physical destination registers to each dispatched instruction. This table is also used to map virtual register operands into the corresponding physical registers	4.3.2
Post-Map Skid Buffer	PSB	Holds a silo of the last few map blocks that have passed through the Pbox forward path	4.3.8
RC/RS Interrupt Flag Widget	RIF	Maintains state necessary to implement the RC/RS instructions	4.3.9
Retire/Kill Unit	RKU	Communicates the identity of retired and/or killed instructions to all concerned boxes by way of the Retire/Kill bus	4.3.11

4.1 Dependency Analysis: General Concepts

Previous "out of order" processors detected dependencies between instructions in different ways. The key goal is to recognize real dependencies between instructions (i.e. true read-after-write (RAW) dependencies) while "untangling" dependencies that are an artifact of the processor archi tecture (like write-after-write (WAW) or write-after-read (WAR) dependencies). For example, take a look at the following chunk of C code:

```
a = b + c;
d = a * a;
a = e + f;
```

Note that there is a RAW dependency (a = b + c must be computed before d = a * a), a WAR dependency (d must be computed before the result of a = e + f is written), and an apparent WAW dependency if the compiler chooses to use the same register for the first value of a as for the second. Let's look at the macro for this C code. (Again, all macro programs are stylized and not meant to reflect actual Alpha assembler code.)

```
    ; A is in R1, B in R2...
001 ADDL R2,R3 -> R1
002 MULL R1,R1 -> R4
003 ADDL R5,R6 -> R1
```

As you already know, the key to out of order execution is to recognize that R1 in this case has many different lifetimes in the course of a program. The lifetime of R1 in lines 1 and 2 is separate and distinct from the lifetime of R1 in lines 3 and thereafter. If the processor architecture provided a bazillion registers, the compiler would use a new register for each lifetime of a value. That is, it would create a new name for each lifetime of the variable a. Let's pretend that the C code was compiled into such an instruction set:

```
001 ADDL R2,R3 -> R1
002 MULL R1,R1 -> R4
003 ADDL R5,R6 -> R11
```

In this case the second lifetime of a is stored in R11. This removes the WAW and WAR spurious dependencies. Now a suitably intelligent scheduler can recognize that instruction 003 can be executed in parallel with (or even before!) instruction 001 or 002.

- Alas, we don't have an infinite (or even very large) number of architectural registers, so sooner or later a compiler that creates a new name for every register lifetime would run out of new names. Fortunately, there are lots of ways to create these new lifetime names at execution time in hardware, rather than at compile time. **We believe that execution time mechanisms offer the best opportunities to squeeze the last bit of performance from a program.** The two most frequently encountered renaming approaches are:
  - Rename each destination register (in the architectural register space) into a physical register (in a larger physical register space). If two different instructions that are in flight (that is, they have been fetched and have entered the scheduling unit and have not yet retired), write architectural register R1, then each lifetime of R1 will be assigned to a different physical register. This mechanism removes WAW and WAR dependencies. In some cases it is used to detect RAW dependencies. (The 21264 uses detects RAW dependencies by comparing physical register names.)
  - Rename each destination register into a serial number. Each in-flight instruction has a unique serial number. This instruction serial number (or INum) can be used for RAW dependency detection. Unfortunately, it cannot be used to eliminate WAW or WAR dependencies unless (as in the case of machines using a re-order buffer) the microarchitecture provides a separate architectural register file. (Since the INum space is finite, each INum is reused fairly often. If instruction 51 writes R1 at time t1 and then writes R5 the next time INum 51 is re-allocated at t2, then we have no way of referring to the lifetime of the R1 that was written at t1. If the write at t1 was the last time R1 was written, then

we have lost its state. The solution to this problem is to copy R1's value to the architectural register file sometime before t2. This copy operation is the reason we don't use a classic re-order buffer organization in the 21464 Qbox.)

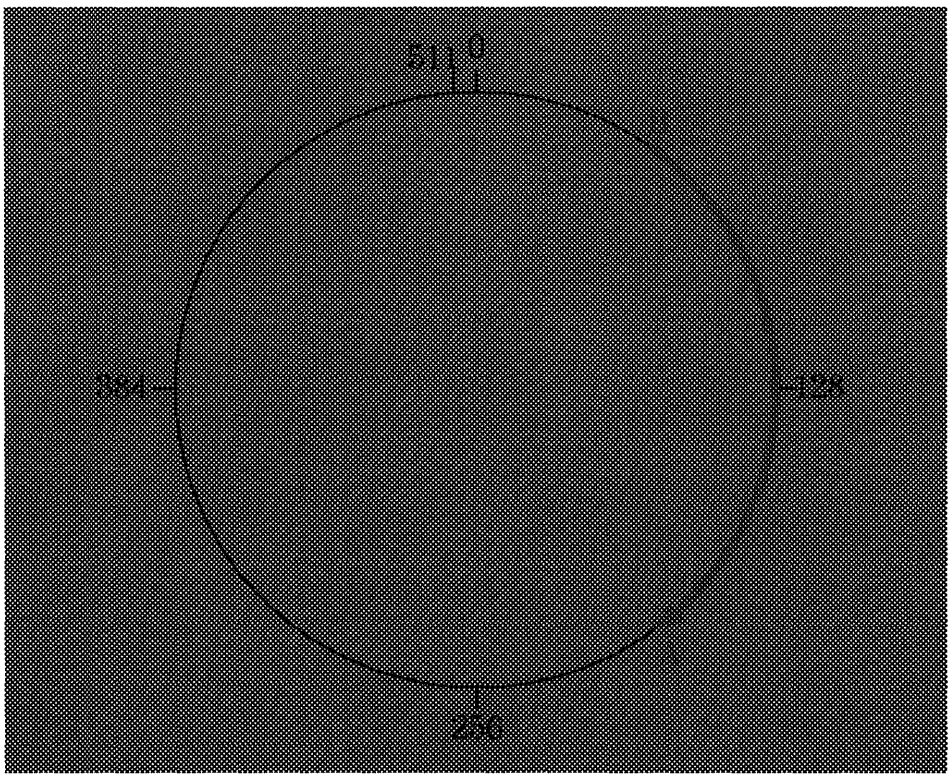
The 21464 Pbox renames incoming register operands into an INum space to facilitate the scheduling decision. We rename incoming register operands into a physical register space to eliminate WAW and WAR dependencies.

4.2 INum Space

Similarly to the 21264, 21464 uses instruction numbers (INums) to uniquely identify in-flight instructions. All TPUs share a single INum space, so INums are unique across TPUs. It is the INum Allocator (see Section 4.3.3) that allocates INums and the Qbox Completion Unit (see Section 5.2.16) that frees them upon retirement.

We use INums in the range 0 to 511. We consider the INum space to be cyclic, so after 511 we wrap back to 0. One can visualize the space as a circle (as in the diagram below) that increases in the clockwise direction, except where we wrap from 511 back to 0. We allocate INums within a TPU in an increasing order (i.e. clockwise). Therefore, within a TPU, younger instructions have larger INums, except in the case of a wrap. In the diagram, INum A is younger than INum B. Imagine that the space between A and B shows the total range of INums in use. Then A represents the insert pointer (the youngest INum is use) while B represents the retire pointer (the next INum to retire).

Figure 4–2 The INum Circle



Why do we have 512 INums? The architecture group did a number of studies and determined that we need to support a scheduling window of 128 entries, and we need to allow at most 256 in-flight instructions at any given time. Therefore, we need to choose

an INum space containing at least 256 INums to uniquely identify all in-flight instructions. In addition to uniquely identifying instructions, we need to be able to compare INums of the same TPU to determine which of two instructions is older. With only 256 INums we cannot accomplish this without additional information, namely which INum represents the youngest in-flight instruction for a given TPU. However, by increasing the INum space to 512 values - tacking a 9th wrap bit onto the lower 8 bits - we can.

4.2.1 INum Age Comparison

A TPU's allocated (i.e. in-flight) INums will never cover more than a contiguous half of the INum circle, 256 of the 512 possible values. This is a very important point; it is this fact that allows us to determine which of two instructions in that TPU is older. In fact there is a simple, robust method for making this determination. First of all, note that we can interpret the INum space as consisting of 9-bit 2's complement signed numbers rather than unsigned values; i.e. the wrap bit becomes a sign bit. The diagram below visualizes the INum circle using this interpretation. Given this fact, the rule for determining the relative age of INums A and B is as follows:

```
if (A - B > 0)
    A is younger than B
else if (A - B < 0)
    A is older than B
```

Where A-B is a 9-bit 2's complement subtraction. Note that the outcome A-B==0 is not possible because of the constraint that in-flight INums cover no more than a contiguous half of the space and are therefore, by definition, unique. To understand why this algorithm works, consider the diagram below. Let A and B be the youngest and oldest in-

flight INums, respectively. The contstraint on the distance between oldest and youngest means that the relative values of A and B break down into four cases, illustrated below. Note again that in every instance, A is younger than B.

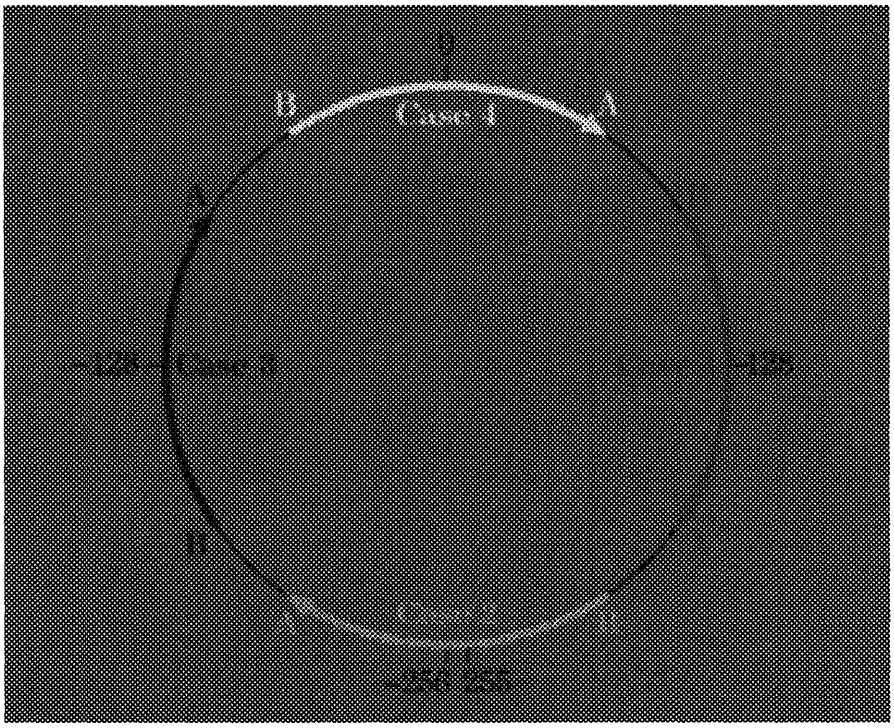


Table 4–2 INum Age Relationship

Case	Sign of A	Sign of B	Magnitude Relationship	Sign of A-B	Sign of B-A
1	+	+	|A| > |B|	+	–
2	–	+	256 < |A| + |B| < 512	+ (overflow)	– (overflow)
3	–	–	|A| < |B|	+	–
4	+	–	0 < |A| + |B| < 256	+	–

Table 4–2 shows the relationship in greater detail. The first four columns merely transcribe what is evident from the illustration, while the last two show that the algorithm gives the correct result for each case. Case 1 is very straightforward; A-B subtracts a positive number from a larger positive one, yielding a positive result. Case 3 is the dual of Case 1, with the signs and relative magnitudes of A and B reversed. In Case 4, A–B subtracts a negative number from a postive one where |A| and |B| add up to a max of 255, so the result is positive and within the range of 9-bit 2's complement representation [-256,255]. For all of these cases, B–A is simply the negation of A–B. Case 2 is a little less intuitive. A–B subtracts a positive number from a negative one, but since |A|+|B| > 256 the result is negative yet out of the range of 9-bit representation — which means that it wraps around to the positive side of the circle. Likewise, B–A subtracts a negative number from a positive one, giving a positive, out-of-range result — which there-

fore maps to a negative value in 9 bits. Notice that |A|+|B| < 512 which means that neither A–B nor B–A can wrap all the way around from positive to positive or negative to negative. Thus 9-bit 2's complement subtraction is sufficient to determine the relative age of any two INums.

In places where we use INums as unique identifiers, and do not need to do age comparisons, we need not store the 9th bit of the INum. Dependency detection is one situtation where uniqueness is sufficient. Therefore, in most places in the Instruction Queue, the 21464 stores only the lower 8 bits of the INum.

4.3 Component Details

4.3.1 INum Mapper (IMP)

4.3.1.1 Design considerations

The central problem in scheduling for out-of-order-issue processors is the identification of dependencies between instructions. Each instruction that reads results from a register file depends on the instruction that last wrote the required result to the register file. Before the issue mechanism can decide that an instruction *X* is ready to issue, it must know what other instructions produce the data that *X* requires. (These instructions are the parents of *X*.)

As an example, consider the following code fragment:

```
I1: LD R3 <- (R4)

I2: CLR R2

I3: ADD R5 <- R3 + R2
```

(All code fragments in this report are stylized and not meant to be in the form of legitimate Alpha assembler notation.)

Assume for the moment that R4 was loaded by an instruction that executed a very long time ago. I1 then is data ready when it is fetched and passed from the Ibox to the Qbox. It has no known parents. Similarly, I2 doesn't read any input operands. It is data ready when it arrives at the Qbox. I3 on the other hand, requires inputs generated by I1 and I2. I3 has two parents, (I1,I2). Until I1 and I2 are issued, I3 is not ready. As it turns out, I1 is a load, so it has a latency of two cycles, thus I3 can't be issued any earlier than two cycles AFTER I1 has issued.

We can determine the dependencies between instructions via several different mechanisms. 21464 has chosen to use INum mapping. In this scheme, a mapper remembers the INum of the last instruction to write each register. At map time, we rename each input register for each instruction from its original virtual register name to the INum of the last writer for that register. This mapping operation maps dependencies from the (limited) virtual register name space with all its spurious write-after-read and write-after-write dependencies into the INum space which is free of these false dependencies.

4.3.1.2 Design Architecture

The INum Mapper (IMP) processes each map chunk (8 instructions) in parallel. It maps the source register specifier for each instruction from the 6 bit virtual register space (31 int registers, 31 floating point register, 2 PAL permanent registers) into INum space (8 bits), each source virtual register being replaced with the INum of the in-flight instruc-

tion that last wrote the virtual register (from the point of view of program order.) Additionally, the IMP remembers which INum last wrote each of the 64 virtual registers in the CMAP (current map) vector. There is a CMAP vector for each of the four hardware threads. The vector is indexed by virtual register number. If the instruction that last wrote a virtual register is not in flight (i.e. the last writer for the register has retired) then the mapper will produce a NULL INum.

Instructions are processed in the order in which they were fetched. Effectively, the read operands of the first instruction are mapped to their producer INums and the write destination register for the first instruction is marked with the first instruction's INum. Then the second instruction is processed, and so on until each of the eight instructions in the fetch block have been remapped.

In fact, the eight instructions in a fetch block are all remapped in parallel. This means that we have two stages of mapping. The first stage maps each of the 16 source operands from VReg space to the INum of its last writer *ignoring other instructions in this fetch block*. This is done by a 16 way parallel lookup into the CMAP for the current thread. The second stage of mapping looks for dependencies within the line and supplies new INum mappings for source operands that are written by instructions in the same fetch block.

The IMP must also maintain a list of "last writers" for each of the registers *for every mapped instruction*. This is called the back map or BMAP. Given that we support up to 256 in-flight instructions, the BMAP could require as much as 256 * 64 bytes or 16Kbytes of storage. A table organized in the obvious way (indexed in one dimension by INum, in the other by VReg) would be very hard to maintain, as we'd have to write up to eight rows per tic into the BMAP.

As it turns out, the BMAP is indexed in one dimension by the fetch block number (this is the INum of the first instruction in the block divided by eight) and in the other dimension by VReg number. Each cell (B,VReg) in the BMAP contains two bytes. The first byte (LAST_WRITER) contains the INum of the last writer of VReg BEFORE fetch block B was processed. The second byte M contains a mask such that if M is set, then INum = B*8 + k wrote register VReg. On a trap, the Mapper Exception logic directs the BMAP to read the map state from the column of cells that corresponding to the trap point and load this state into the CMAP.

This restore operation is done in parallel for each of the 64 entries in the CMAP like this:

```
FOR i = 0 to 64 DO
            IF BMAP(TrapINum<7:3>,i).M<7:0> == 0
            THEN
                CMAP(i) = BMAP(TrapINum<7:3>,i).LAST_WRITER
            ELSE
                CMAP(i) = BMAP(TrapINum<7:3>,i).LAST_WRITER
                FOR j = 0 to TrapINum<2:0> - 1 DO
                    IF BMAP(TrapINum<7:3>,i).M == 1
                    THEN
                        CMAP(i) = 8 * TrapINum<7:0> + j
                    END
```

```
        END
    END
END
```

If an entry BMAP(B,VReg).M<7:0> is zero, then the CMAP(VReg) is loaded with BMAP(B,VReg).LAST_WRITER. Otherwise we scan the writer mask for the backmap cell *up to the trap point* if any instructions that are before the trap point and in this map block have written VReg, then the CMAP is loaded with the INum of the last such instruction. Otherwise it is loaded with the LAST_WRITER INum.

4.3.1.3 Map Predecode Bits from the Ibox

The predecode value bits communicate the type of the source and destination **dependencies** for each instruction category, as shown in the table below.

In the table:

- Source A and Source B correspond to the Ra and Rb fields respectively
- A Null value indicates that the corresponding field does not contain a valid source
- SourceIPRclass and WriterIPRclass refer to Internal Processor Register scoreboarding classes.
- ShadowReg1 is a PAL shadow mode register number 1; its use is implicit in the CALL_PALL instruction opcode.

Note that this table is a somewhat coarse approximation of how opcodes are allocated to predecode categories. The actual assignment for a particular opcode is not always obvious. Consult Section 3.8.2.3.1 for the exact mapping. Table 4–3 lists the predecode value meaning for the predecode bits received from the Ibox.

Table 4–3 Predecode Value Meaning for I%MAP_INST_I4A_H[7:0]<35:32>

Predecode Value[1]	Source A Dependency Type	Source B Dependency Type	Destination Dependency Type	Destination Dependency Field	Instructions
00101	Integer	Null	Integer	Rc	Integer operates with Rb=immediate
01110	Floating-point	Null	Integer	Rc	FtoI*x*
00011	Null	Null	Integer	Ra	Unconditional branch
01001	Null	Null	Null	Null	MB and other special instructions
00111	Integer	Null	Floating-point	Rc	ItoF*x*
01100	Floating-point	Null	Null	Null	Floating-point conditional branch
00010	Null	Null	ShadowReg1	Implicit	CALL_PAL
01000	Integer	Null	Null	Null	Integer conditional branch
00100	Integer	Integer	Integer	Rc	Integer operates and store-conditional

Table 4–3 Predecode Value Meaning for I%MAP_INST_I4A_H[7:0]<35:32> (Continued)

Predecode Value[1]	Source A Dependency Type	Source B Dependency Type	Destination Dependency Type	Destination Dependency Field	Instructions
11110	Floating-point	Floating-point	Floating-point	Rc	Floating-point operates, MF_FPCR, and MT_FPCR
11011	Null	Integer	Integer	Ra	Integer loads
11000	Null	Integer	Floating-point	Ra	Floating-point loads
00110	SourceIPRclass	Null	Integer	Rc	HW_MFPR (See Sec. Section 17.2)
11101	Floating-point	Integer	Null	Null	Floating-point stores
10110	SourceIPRclass	Integer	WriterIPRclass	Rc	HW_MTPR (See Section 17.2)
11111	Integer	Integer	Null	Null	Integer stores

[1] These values do not necessarily represent the types of the operands themselves. The Pbox uses the predecode bits to distinguish between floating-point, integer, IPR, and PAL shadow register dependencies (since the same virtual register bits can have different semantics depending on the instruction type), and to detect the absence of dependencies (i.e. the Null cases). The predecode values allow dependency analysis to proceed without waiting for a full decoding of the instruction opcode and function fields. See also Table A–2 for more mapping information.

4.3.2 Physical Register Map (PMP)

4.3.2.1 Design Considerations

While the IMP has mapped each virtual register operand from its virtual register number into the INum of the last instruction to write the register (or INum = NULL if the last writer has already retired), we still need to find out where to store each destination and where in the physical register file the latest lifetime of each virtual register resides. (That is, the IMP told us who last wrote register *X*, but we also need to know where the writer put the data — which physical register contains the latest lifetime of each virtual register.)

Why did we bother to remap to INum space if we're only going to map again into a physical register space? The whole thing has to do with our nearly paralytic fear of free-lists. In a 21264-like scheme for register mapping, we would need to pick eight good free registers out of a pool of 512. That is perceived as being very hard to do.

And so, you will notice the INum mapper has no giant free list. The next INum is peeled off sequentially. (This is almost true, see Section 4.3.3.) Because of that, the backup map is much simpler than it might otherwise be, and trap recovery is simpler.

Unfortunately it forces us into another remapping, since some input operands were written so long ago that the INum that was formerly associated with them has retired and been re-allocated to a new destination register. (An INum is only a good "rename" for a virtual register while the INum is in flight.) Again, we are in mortal fear of building a "gimme eight good ones" free list mechanism. So, after a whole lot of collaboration, the 21464 team came up with a really neat scheme for mapping from INums to physical registers that doesn't use a free list.

4.3.2.2 Design Architecture

The general scheme that we intend to use for physical register renaming is to translate each virtual register specifier in an instruction into the INum that last wrote it (or NULL if the last writer has retired) and then translate the last writer INum into the physical register that it wrote. If the last writer INum is NULL, then we look up the physical register name in the Architectural State Table. (A register that was last written by a retired instruction is referred to as being part of the architectural or in order state of the machine. We will use the term architectural state here.

The IMP has done the first half of the rename task for us. The second half can best be described with a program chunklet. We need to rename both source register operands and destination register operands. Assume that the source virtual register is SVReg, the INum it was last written by is SLastWriterINum, and we are looking for the register's current physical home SPReg.

```
if (SLastWriterINum == NULL) {
    /* the source register is part of the architectural state */
    SPReg = ArchRegTable[SVReg];
}
else {
    /* the source register was last written or will be written
       by an instruction that is currently in flight.  What is
       that instruction's next destination register? */
    SPReg = NextDest[SLastWriterINum];
}
```

Note that the operation of mapping from source virtual register to source physical register required nothing more complicated than one or two reads from a thing that looks like a register file.

Mapping destination registers is a little more complicated. Assume that the destination virtual register is DVReg, the INum that the destination was last written by was DLastWriterINum, and the physical destination register will be DPReg. The INum of the instruction we are remapping is WriterINum.

```
if (DLastWriterINum == NULL) {
    /* The last writer retired a while ago.  The
       DVReg is currently in architectural state. */
    LastDest[WriterINum] = ArchRegTable[DVReg];
}
else {
    LastDest[WriterINum] = NextDest[DLastWriterINum];
}
DPReg = NextDest[WriterINum];
```

Let's start from the bottom. Note that there is an array called "NextDest" that is indexed by the INum of the instruction we are currently mapping. This array contains the destination register for each in-flight INum. If a physical register number appears once in the NextDest array it won't appear twice. That is, each in-flight instruction has a unique

destination register. (Up to 256 in-flight instructions — 512 registers; not a coincidence.) All we need to do in translating from a virtual destination register to a physical destination register is look the instruction's INum up in the NextDest array. At the same time, we find out where the previous physical home for the DVReg was. We load this physical register number into a second array that contains the physical destination register that WriterINum will use at some point in the future. The idea here is that if an instruction writes R5, then the previous home for R5 will be a "free" register when *this* instruction retires. This is a really knotty point here. Read the paragraph again. It usually takes folks a few times through before they fully appreciate the elegant simplicity of this approach.

Now notice that if we don't retire WriterINum, then the old home for DVReg (that is, physical register DPReg) will not become a free register. (If an instruction doesn't retire, it is as if we never wrote its destination operand.) On the other hand, since WriterINum never retired, we can re-use the same physical register stored in the NextDest array when we re-allocate WriterINum next time. If the instruction does retire, then we need to prepare WriterINum's entry in the NextDest array for the next time around. We can't leave NextDest[WriterINum] unchanged, or we'll over-write what might well be architectural state. So first we need to update the architectural state table:

```
ArchRegTable[DVReg] = NextDest[WriterINum];
```

Then, we simply do this:

```
NextDest[WriterINum] = LastDest[WriterINum];
```

Note also that on a trap (i.e. when we abandon a group of INums) we don't do anything at all in the PMP.

The whole scheme looks pretty good. The downside here is that the obvious implementation requires two read ports into the NextDest array for the read operands, plus one read port for the write operand, plus one read port for writing the last writer from NextDest to LastDest, plus one write port for the LastDest to NextDest update for each of eight instructions. In addition, since we may retire as many as 16 instructions per cycle, we need 16 read ports to copy from the NextDest table to the Architectural state table. That's 48 read ports and one write port into the

NextDest array plus the read and write ports into LastDest and NextDest for instruction retiring and so forth.

The first refinement is to turn the copy of LastDest to NextDest into a lateral copy that doesn't actually use any read ports at all. That makes the retire operation rather inexpensive in the PMP. The second refinement is to use group reads of contiguous entries (i.e. one read port is only needed to read out one block of 8 entries) for the 8 read ports for the write operand and the 16 read ports for updating Architecture State Table. Therefore, we reduce the number of read ports of the NextDest array to 27. One obvious approach to further reduce is to replicate the NextDest and LastDest arrays. The Qbox/Pbox team has developed an approach that replicates the NextDest array by a factor of three, reducing the requirement to a register file like array that is eight bits wide, 256 entries deep, and has nine read ports, and one write port. This appears well within the bounds of practical implementation.

4.3.3 INum Allocator (INA)

4.3.3.1 Design Considerations

Each incoming instruction must be assigned an INum. INums are defined in INum Space (Section 4.2). 21464 has a space of 512 possible INums where only 256 at most are in use at any given time — the same as the maximum number of in-flight instructions.

The INum is used for RAW dependency detection, branch mispredict recovery, and general exception identification. Dependency detection is discussed in Section 4.1. If a branch instruction is executed and the Ebox/Fbox/Qbox determines that the branch was mispredicted, the executing unit must send the branch instruction's INum to the Ibox. The Ibox uses the INum as an index into a table containing the alternate branch address. The INum is also used to establish priority of exceptions. If two exceptions are signalled to the Ibox at the same time, the Ibox will compare the INums responsible for the exceptions and chose the earliest INum.

To address dependency detection, the assigned INums need only be "unique", that is two in-flight instructions can't ever have the same INum. The requirements imposed by exception processing are more stringent, however. In order to identify the "oldest" of a pair of in-flight instructions their INums have to be assigned such that we can always determine whether instruction A is older than instruction B or not. This issue is described in Section 4.2.1. But more interesting problem for the INum Allocator is deciding how to divide INums amongst different threads.

The INum Allocator also subsumes the related but distinct functionality of the Map Thread Chooser (Section 4.3.3.3), which decides from cycle to cycle which TPU should map and pass an instruction block along to the Qbox.

4.3.3.2 Design Architecture

In single thread mode, INum allocation is simple; all we need to do is to allocate INums in the range 0 to 255 and toggle a "wrap bit" that becomes INum<8> each time we pass through 0. Further, we need only make sure that all the in-flight INums are on the same half of the 512 entry circle.

Multithread mode is a little more complicated. In this case, we need to maintain the ordering relationship between INums within a thread only. (A comparison between INum A from thread 0 and INum B from thread 1 is meaningless, or rather, doesn't need to have any significance.Exception priority resolution — a major reason for having INums — is done independently in each thread.) This ordering can be maintained by ensuring, as in the single thread case, that the insert pointer and retire pointer stay in the same 255 value range.

The problem with INum allocation in multithread mode is ensuring an optimal allocation of INums to active threads. Perfect allocation requires knowledge of what each thread will do in the future. We don't have that knowledge, so we have to settle for "good enough" allocation.

We considered dividing the space from 0 to 255 into four equal sized chunks and allocating each chunk to one and only one of the four threads. We called this scheme "hard partitioning". This approach has one really big problem: it implies that when we only have a single thread that is active, that thread can only have 64 instructions in flight. In this case, 21464 takes a 20% performance hit. This is unacceptable for two reasons.

First, we want multithreading to be modeless: when there is only one active thread we are in single thread mode, when there is more than one active thread we are in multi-thread mode. The transition from one to the other should not require great honking masses of machinery to reconfigure themselves. Second, there is some suspicion that many applications pass through serial sections of code where all the child threads are waiting on a single parent thread to do a particular task. In the hard partitioned scheme, that parent has no access to the idle resources for which the children have no need. (The children are quiescent — asleep — waiting for the parent to do its thing.) This approach was too slow.

At the other end of the solution space, we considered completely free allocation of INums from a pool. We hate the idea of free-list choosers. This approach was too hard.

The approach we have chosen is a compromise. We divide the range 0 to 255 into four chunks. Thread 0 is given INum blocks 0, 4, 8, 12 and so on to block 28. Thread 1 is given INum blocks 1, 5, 9, 13 and so on. Each block contains 8 INums. When a thread needs a new INum block, it looks forward from its insert pointer over the next four blocks for the first free block. When a thread is quiescent, it returns all of its "own" blocks to a "shared pool" as each block retires. (If some of a quiescent thread's blocks are not in flight, they enter the shared pool when the thread quiesces.) When a thread wakes up, it claims all of its own blocks from the "shared pool". When a thread *X* goes looking for a new block, it looks in the shared pool, and in the list of idle (not in flight) blocks belonging to thread *X*. This has the advantage of making idle resources available to active threads, while avoiding complex free-list schemes. This approach was just right.

4.3.3.3 Map Thread Chooser (MTC)

Apart from being responsible for INum allocation proper, the INum allocator also contains the Map Thread Chooser (MTC), which is responsible for picking a valid thread to map each cycle. A valid thread is one that is not quiesced, and either has instructions in the Ibox collapsing buffer or has its fetch valid bit set. The latter signal indicates that this thread's instructions are currently being fetched and thus can bypass the instruction buffer. No threads can or will map if the post-map skid buffer gets full or the INA runs out of INums. Sometimes one or both of these events occurs - or the MTC chooses a thread which turns out to be invalid - while a fetch block is en route from the Ibox to the Pbox. To enable recovery in these events, the Ibox retains the last fetch block sent until the Pbox indicates that the collapsing buffer can update itself. The MTC will assert the update signal any cycle it has a valid map thread choice and has INums available (since this implies the last block was successfully mapped).

The MTC will choose the thread with the fewest number of consumed INum chunks that have valid instructions to map. In the event that two or more valid threads have the same least number of consumed INum chunks, the tie is broken using a round-robin algorithm. In order to monitor the number of INum chunks in flight, the MTC maintains four counters, one per thread. A thread's counter is incremented when one of its fetch blocks is passed to the Pbox from the Ibox's collapsing buffer and successfully mapped, and decremented when one of its map blocks is retired by the Qbox. The MTC determines which threads have valid instructions to map by monitoring the number of fetch blocks in flight that could be mapped by the time the map choice is accepted by the collapsing buffer stage. After making a map choice, the MTC forwards its selection to the Ibox.

The MTC may choose a thread that cannot actually map because it had a line mispredict, set mispredict, Icache miss, ITB miss, etc. If any of these Ibox mishaps should occur, the Ibox will send a "slot 0 invalid" signal to the MTC. If the MTC sees this signal, it knows that the chosen thread cannot map this cycle. Therefore will not update its counters but restart its map choice on the next cycle. It will also clear all of the bits in the Map TPU signal, indicating that the current map thread choice is invalid.

4.3.4 Mapper Exception Logic (MEX)

4.3.4.1 Design Considerations

When the Ibox signals that it has fielded an exception, the IMP, PMP, LSN, and RIF, must restore their state to the time the trapping instruction was mapped. The INum allocator insert pointer is unchanged, but all INums between the trap point and the current insert point must be abandoned.

4.3.4.2 Design Architecture

The Mapper Exception logic (MEX) takes as inputs the thread id and the exception INum of an exception (from the BEL) and accesses the appropriate column in the backmap (BMAP) of each affected section (IMP, PMP, LSN, and RIF) to roll the thread's context back to the trap point. In addition, the MEX clears all relevant BMAP state between the trap and insert points. (The "writers" mask in each cell of the BMAP between the trap and insert point is cleared if the corresponding INum has been abandoned. This way, when the state of the mapper is restored, all registers whose last writer is either retired or abandoned will be marked as being "ready to read".

4.3.5 Memory Queue Allocation Unit (MQA)

The Memory Queue Allocation Unit (MQA) governs the allocation and deallocation of load queue (LQ) and store queue (SQ) chunks to memory instructions. The Mbox accepts and implements the allocation decisions of the MQA. However, the Mbox initiates the deallocation process, with the exception of killed instructions, whose LQ/SQ chunks are aggressively reclaimed by the MQA. The MQA also controls the High-Water Mark (HWM) that is sent to the Qbox to regulate the issuing of loads and stores.

Note: We will attempt to explain the operation of the MQA in terms applicable to both load and store queue functionality. However, we will default to using the store queue operation when a description in generic terms becomes too cumbersome. Differences between the load and store queue allocation logic will be noted as necessary.

4.3.5.1 Allocation

Allocation is based on the per-TPU demand for load/store queue chunks (LSChunks). The MQA assigns LSChunks only to active (i.e. not quiescent) TPUs which exhibit demand by mapping load and store instructions - with one twist.

There is a delay of several cycles between when the HWM is elevated and the Qbox recognizes that an instruction that was above the HWM is now below it and may issue. To mitigate this latency, the MQA artificially inflates demand for each active TPU, so that even without mapping any memory instructions, a TPU has demand for some number of LSChunks. This number is currently thought to be 2, but that is the subject of ongoing performance model experiments. This inflated demand effectively results in

the preallocation of LSChunks to a TPU, and a corresponding elevation of the HWM, such that data-ready memory instructions have a better chance of issuing immediately upon entering the instruction queue (IQ).

4.3.5.2 Background and Terminology

To make it easier to discuss the MQA algorithm, we define the following terms:

YLSNum	The Youngest LSNum allocated to mapped instructions by the LSN
ADLSNum	The Artificial Demand LSNum - i.e. YLSNum plus artificial demand; provided to the MQA by the LSN

In addition, there are a few important background facts to keep in mind when considering MQA operation:

- LSChunks contain groups of 4 LSNums.
- The HWM is maintained on an LSChunk granularity (i.e. it does NOT change on the granularity of individual LSNums).
- ADLSNum is also maintained on an LSChunk granularity.
- Only instructions with LSNums strictly below the HWM may issue; it follows that the HWM is always immediately above the last allocated LSChunk.
- Each of the two load queues and one store. queue have completely distinct, independently managed LSNum/HWM spaces; so do the TPUs within a given load/store queue.
- LSNums, unlike INums, are allocated continuously; there is no necessary connection between the boundaries of INum blocks and LSChunks. This fact comes into play most significantly in dealing with kills and retires.

4.3.5.3 Basic Allocation Loop

We will use store queue operation to illustrate the basic allocation loop. On every cycle, each TPU compares its store ADLSNum (from the LSN) to the current store HWM. If ADLSNum >= HWM, this TPU has demand for store chunks and submits a bid to TPU arbitration.

If there are one or more free store chunks available, the TPU Arbitration unit selects a winner from the active TPUs which have demand. The winner is the TPU from the set of bidders which was least recently allocated an LSChunk; the TPU which wins in this cycle goes to the back of the line. Arbitration declares no winner if there are no active TPUs with demand and/or no free chunks.

The MQA allocates an LSChunk to a TPU which wins arbitration in a given cycle. Conceptually, when an LSChunk is allocated, it is set aside for the stores whose SNums are in the range of the ADLSNum value for which the bid was generated. For example, if the bidder's ADLNum was 80, the allocated LSChunk will contain the stores with SNums 80, 81, 82, and 83.

A successful arbitration leads to a number of events, both internal and external. Internally to the MQA, the winning TPU updates its store HWM by adding one LSChunk. The Allocation Picker chooses which of the free LSChunks will be allocated, and the ADLSNum value is written into that LSChunk's entry in the Tag Array. The match

enable (MATCH_EN) bit for that entry is also set. Finally, the allocated chunk is removed from the Available vector and assigned to the Inflight Vector of the winning TPU.

As for externally visible events, the MQA sends the winning TPU ID, ADLSNum, encoded LSChunk ID, and a valid bit to the Mbox, which writes the TPU and ADLSNum into the appropriate SQ entry. When stores with LSNums in the range of this ADLSNum issue, their state will be assigned to this entry. In addition, the MQA sends the updated HWM to the Qbox.

Going back to the beginning of the process, if on a given cycle ADLSNum < HWM for a given TPU, then it has no additional demand for store chunks. It does not submit a bid to arbitration nor update its HWM.

4.3.5.4 Reset

During reset, the HWM for each TPU is set to 0, and LSNum allocation also starts from 0. However, the LSN initializes the ADLSNum for each TPU to a positive value, ensuring that all TPUs come out of reset with demand. At their first opportunity, the TPUs will bid and arbitrate to raise their HWMs to the point where ADLSNum < HWM - possibly before any memory instructions have been mapped or even fetched.

4.3.5.5 Deallocation

The following sections cover how the MQA responds to kill and retire events. However, at this point it is important to address the non-obvious role of the Mbox in deallocation. It is easy to grasp that LSChunks mapping to instructions in the shadow of a kill can deallocate, although there are some subtleties we will discuss shortly. One would also tend to think that LSChunks in the shadow of a retire can also be immediately reclaimed, but this is not the case.

Both stores and loads may surrender their INums before they are ready to actually leave the SQ or LQ. Stores maintain state in the SQ until they are copied out of the merge buffer, which may not only happen long after they become retireable, but out of INum order. In the LDQ, prefetches never lead to a retry and thus retire early, possibly long before they have executed. For these reasons, the MQA may only deallocate LSChunks for retired instructions once the Mbox says it is safe to do so. This is achieved via a fully-decoded deallocation signal per LQ/SQ. The MQA response to this signal is to remove the designated LSChunks from the Inflight Vectors and place them in the Available Vector.

4.3.5.6 Kills

When a kill occurs, the MQA has to reclaim any LSChunks in the shadow of the kill. In the interest of performance, the MQA tries to reclaim any LSChunks that map onto killed instructions. This is achieved by comparing the ADLSNum corresponding to the killed instruction with all LSChunk tags for the kill TPU. All tags where LSChunk >= kill ADLSNum and the MATCH_EN bit is set generate a match signal, which leads to their removal from the TPU's Inflight Vector and addition to the Available Vector. The MATCH_EN bit is also cleared for all matching tags, making them ineligible for future comparisons - this avoids aliasing problems when the LSNum space wraps.

Note that we kill from the ADLSNum corresponding to the kill INum - a value obtained from the LSN Backmap - not the YLSNum which maps to the kill INum. This means that the blocks between the YLSNum and ADLSNum - i.e. most of the blocks allocated

due to artificial demand - remain allocated to the TPU after a kill, a handy optimization. This also has the side effect of solving the problem of partially-killed LSChunks. The effective kill point is always at the beginning of some LSChunk after the true kill point, so we don't need to worry if the kill occurs in the middle of a chunk or not.

Also note that the mapping from a kill INum to the corresponding YLSNum (and ADLSNum) is subtle. If the kill INum is that of a store instruction, for example, then the corresponding kill YLSNum is the SNum of that store. But if the kill is for some other type of instruction, the kill YLSNum is the SNum of the last store allocated prior to this instruction.

The aggressive reclamation of killed LSChunks by the MQA has an important implication for the Mbox. To avoid a hazard, the Mbox must not send a deallocation signal for LSChunks that are completely in the shadow of a kill. Otherwise, the MQA could receive spurious deallocation signals for LSChunks that it has just reclaimed and then allocated.

Making killed LSChunks available for reallocation is only part of the task. The MQA also needs to check the relative position of the kill and the current HWM. If kill ADLSNum >= HWM, the HWM has not yet caught up with or is just equal to demand, and there is no problem. However, if HWM > kill ADLSNum, then the High-Water Mark is above the point where we have allocated LSChunks for this TPU and must be lowered. In this case, the MQA sets HWM = kill ADLSNum. Note how this means that the TPU comes out of a kill with a positive demand for LSChunks.

As a final note, the Mbox must make sure on kills that it is truly good and done with LQ/SQ entries before the MQA has a chance to reallocate them. The pipeline would appear to provide more than enough time for this, but we need to make sure.

4.3.5.7 Retires

We have already discussed how retirement of LSChunks is decoupled from deallocation, due to the tendency of stores and prefetches to linger in their queues past retirement. The immediate action that the MQA must take on retirement is to disable the retired instructions from matching against future kills (and retires). The mechanism for handling this operates as follows: on a Retire Block event, the LSN reads its Backmap and supplies the YLSNum corresponding to the retiring block. Note that this must be the YLSNum, not the ADLSNum, since the latter corresponds to instructions still in flight! The LSChunk tags compare against the retire YLSNum, and all LSChunks for the TPU that are in the shadow of the retire (i.e. older and with MATCH_EN bits set) clear their MATCH_EN bits. There are actually two different cases:

1.If retire YLSNum == ...11 then clear MATCH_EN for all tags where LSChunk <= retire YLSNum 2.If retire YLSNum != ...11 then clear MATCH_EN for all tags where LSChunk < retire YLSNum

In other words, if the retire YLSNum corresponds to the youngest entry in an LSChunk, the chunk is fully retired and we may clear MATCH_EN for it and all older chunks. If the retire YLSNum is not the youngest one in a chunk, then the chunk is only partially retired, and only the MATCH_EN for the older chunks may be cleared.

LSChunks will typically retire well before they are deallocated. However, the MQA retires on a granularity of Retire Block events (i.e. INum blocks), whereas the Mbox retires at the resolution of Next-to-Retire events (i.e. individual instructions). For this

reason, the MQA may see a block deallocate before it has retired - for instance, if a Next-to-Retire of a store allows the Mbox to release a SQ chunk before the INum block containing that store can retire. This means that the block in question will reside in the Available Vector and no longer in the TPU's Inflight Vector, but still have its MATCH_EN set. This is not a problem, since the fact that the chunk doesn't belong to any TPU means that its tag won't match on anything; the tag state will be overwritten when the chunk reallocates.

4.3.5.8 Quiesce

[NOTE: This is just a sketch - we'll need to decide if things actually work this way or not - Peter].

In the interest of performance, a TPU going into Quiesce needs to release all of its LSChunks to the free pool. The Mbox signals Quiesce to the MQA only after the Quiesce trap has been taken, and after all LQ and SQ instructions for the TPU have completed. When the MQA sees the Quiesce signal, it removes all allocated LSChunks from the Inflight Vector of the TPU and places them into the Available Vector. This avoids forcing the Mbox to send out deallocate signals for any partially-utilized LSChunks belonging to the TPU going into Quiesce. The MQA knows that when it sees the Quiesce signal, it is safe to reclaim all chunks allocated to the TPU.

Coming out of Quiesce is relatively straightforward; for the TPU in question, it looks very much like coming out of reset. LSNum allocation restarts at 0, and the HWM also starts at 0, arbitrating its way up to the level called for by the ADLSNum in the first few cycles after waking up.

4.3.5.9 Merge Buffer Purging

[This is a placeholder for whatever policy we decide to implement. - Peter]

4.3.6 Instruction Decoder (IDC)

4.3.6.1 Design Considerations

In the traditional microprocessors of yore, there was an instruction decoder (just one). It sat at the front end of the processor, parsing instructions as they came into the machine and telling the functional unit(s) exactly what to do for each operand; which registers or memory to get data from, how to operate on the data, where to put it, etc. 21464 has many more registers, funtional units, and operations than that Jurassic CPU, and by virtue of this additional complexity - and size - is far less centrally controlled. There is also a complex register renaming process and out-of-order scheduling operation between the processor front end and the functional units, so this classical model of operations is not practical. Still, there are certain things we would like to know about the instructions within the Pbox and Qbox before we ship them off to their ultimate destinations.

4.3.6.2 Design Architecture

The Pbox Instruction Decoder (IDC) acts similarly to the instruction decoder in a traditional textbook microprocessor, with a few important distinctions. First of all, the IDC operates on up to 8 instructions in parallel. Secondly, the output of the IDC does not directly or exclusively drive the functional units which execute the instructions - the Ebox, Fbox, and Mbox, which also see the opcode and function bits, perform local decoding to determine (for the most part) how to execute a given instruction. Rather,

the IDC outputs are signals which highlight particular properties of the instructions, in the manner of scoreboarding, mode, predecode, or valid signals, for example. Some of the signals go to the Load Store Serial Number Allocator or the RC/RS Interrupt Flag Widget to condition their behavior. The remainder go through the Post-Map Skid Buffer to the Mbox, or to the Qbox where they either influence scheduling decisions and/or get cached in the Payload Arrays for distribution at issue time. One of the most important specific functions of the IDC is to provide slotting information to the Instruction Queue.

4.3.7 Load/Store Serial Number Allocator (LSN)

4.3.7.1 Design Considerations

21464 issues loads and stores out-of-order. This can lead to deadlock situations in the Mbox if things aren't managed properly. For instance, imagine we executed the following chunk of code:

```
001:    ST      R3->(R5)
002:    LD      (R5)->R2
003:    LD      (R9)->R2
```

Note that these three instructions must execute in order if we are to get the ``right" answer. Now imagine that we have an Mbox with a combined load/store queue. That is, all stores and loads enter a single queue. Further, suppose that the queue had just two entries. Instructions enter the queue at issue time, and leave the queue when it is known that they will retire. Suppose R3 does not become data ready until well after R5 is data ready. In that case, the two load instructions will issue before the store instruction. They will consume both entries in the LD/ST queue. Neither however, can leave the queue since we don't know whether either will be able to retire until instruction 001 issues. (Note that we don't need to wait until they retire to eject them, we just need to make sure they won't need to be replayed. Without seeing all ``earlier" stores, we can't make that decision. The LD/ST queue is now deadlocked.

Yes, this example is contrived — we won't build a combined queue of just two entries. However, the deadlock behavior is inherent in the design and is inevitable if the number of LD/ST queue entries is less than the in-flight instruction limit. (Or perhaps the limit is the size of the instruction window — it depends on when instructions are allowed to leave the IQ window.)

4.3.7.2 Design Architecture

To solve this problem, we assign an ascending serial number to each load and store instruction. (Each load will get a LNum and each store will get a SNum.) The Mbox has separate load and store queues. Each has sixty four entries. We will allocate LNums and SNums in the range 0 to 255, which accommodates the pathological case where all 128 instructions in the queue are loads. Each TPU has its own independent LNum and SNum space. Memory barrier instructions will get an SNum. (A memory barrier instruction will never be dispatched to a functional unit - it is a NOP. But a marker for the MB will be sent to the Mbox when the MB passes through the Post-Map Skid Buffer. The Mbox can always deduce the next LNum or SNum to be allocated and the position in the load and store queues of each decoded barrier instruction.

As the Mbox processes entries in the LSqueues, it frees up space in the queue. On each cycle the Mbox will send two values per TPU to a box in the IQ. This box is called the Load/Store Number High-water Marker (HWM). Each load/store/barrier instruction in

the IQ stores its L/S number in the HWM. The Mbox sends MAXLNum<8:0> and MAXSNum<8:0> to the HWM. Each load instruction compares its LNum to MAXL-Num. If it is "less than" MAXLNum, then the load instruction may issue as soon as it is data ready. Otherwise the load instruction must wait. Stores behave similarly. Note that the comparison is actually "less than" but within the same half of the 512 point LSNum "circle".

As the Mbox processes entries in the LSqueues, it frees up space in the queue. On each cycle the Mbox will send two values per TPU - the load and store high-water marks - to a box in the IQ. This box is called the Load/Store Number High-water Marker (HWM). Each load instruction compares its LNum to current load high-water mark for its TPU. If it is "less than" (or "below") the high-water mark, then the load instruction may issue as soon as it is data ready. Otherwise the load instruction must wait. Stores behave similarly. Note that the comparison is actually "less than" but within the same half of the 512 point LSNum "circle".

Finally, note that the LNums and SNums must be reclaimed when instructions are abandoned due to a trap. This behavior differs from the IMP recovery mechanism. However, the LSN is almost identical to the IMP. In this case, the backup map array for the LSN is indexed as LSN_BMAP(B,ISload), that is there is a column for each block of INums, and one row for loads and one row for stores. The LSN_CMAP is a pair of counters that are incremented each cycle by the number of loads/stores that were allocated in that cycle. On recovery, the LSN_CMAP is restored like this:

```
FOR i = LOAD to STORE DO
          IF LSN_BMAP(TrapINum<7:3>,i).M<7:0> == 0
          THEN
              LSN_CMAP(i) = LSN_BMAP(TrapINum<7:3>,i).LAST_NUM
          ELSE
              LSN_CMAP(i) = LSN_BMAP(TrapINum<7:3>,i).LAST_NUM
              FOR j = 0 to TrapINum<2:0> - 1 DO
                  IF LSN_BMAP(TrapINum<7:3>,i).M == 1
                  THEN
                      LSN_CMAP(i) = LSN_CMAP(i) + 1;
                  END
              END
          END
      END
```

That is, the LSN_CMAP entry for loads is set to the last LNum allocated before the trap point. If the trap point is in the middle of a block, we check to see if there were loads before the trap point but within the trap block. If so, we increment the LAST_NUM by the number of loads we find. Stores are treated in the same way.

4.3.8 Post-Map Skid Buffer (PSB)

4.3.8.1 Design Considerations

Because there are fewer instruction queue entries available than the in-flight instruction limit, the IQ sometimes fills. Because of pipeline delays, the Pbox may not realize that IQ is full until well after it has filled, and several instructions have been dropped on the floor. In this circumstance, we don't want to signal a trap all the way back to the Ibox, becaise this wastes time and (perhaps more importantly) wastes INums. (We do not reclaim bad path INums when a trap occurs, only after their retirement.) It is much better to buffer up the excess instructions until the Ibox can be informed to stop fetching in a tidy mannerDesign Architecture

4.3.8.2 Design Architecture

As each instruction passes out of the main Pbox forward path components (INA, IMP, PMP, IDC, LSN, and RIF) on its way to the Qbox, it is copied into the post-map skid buffer circular queue. When an IQ full stall is signalled, the skid buffer sets the front pointer of the queue to point to the instruction block after the last one that was successfully allocated into IQ. Which one this is can be determined in advance from the pipeline timing. When the IQ full condition is cleared, the skid buffer replays the stored map blocks down to the IQ.

Control of the Ibox is achieved by sending an INHIBIT_ISTREAM signal to the INA, which incorporates it into the ``INums available" information that it sends to the Ibox, thus signalling that there are NO available INums. Rather than simply telling the Ibox to send new instructions every time the Qbox signals that the IQ is not full, we prefer to empty out the PSB so that we can bypass the new data coming down the pipe to the Qbox. To do this, we make sure that the PSB is empty enough that by the time the new instructions get to the PSB, we are able to bypass the data to the Qbox and not write it into the PSB only to be read out sometime later.

The skid buffer contains all the muxing logic necessary to select between the silo outputs and the unbuffered forward path.

The post map skid buffer (PSB) is approximately 3 entries deep, the actual depth is dependent on the control pipeline timing from when the Qbox signals that the IQ is full to when we can tell the Ibox to stop sending the Pbox new instructions, and the data pipeline timing from the Ibox collapsing buffer to the IQ. Each entry stores the TPU, the instruction valid vector, and all the data that goes to the Qbox for a given instruction. The PSB is built in 4 separate storage arrays - one for data produced in P1A, another for P3A data, and two for P2A data since these signals span a layout partition boundary. The P2A and P3A array uses the control piped from the P1A array.

The control logic itself uses no state machines apart from the read and write pointers, which are one-hot and either advance or maintain their current position on every cycle. The decisions of whether or not move the pointers, and whether to bypass incoming blocks or read out of the buffer are based on four binary variables: the adjacency (or not) of the pointers, the (in)validity of the incoming block, the current state of the output mux, and the fullness (or not) of the IQ.

On a kill, the trap TPU needs to be compared to the stored TPUs. If there is a match, then the instruction valid vector needs to be set to all zeros. If there are any valid entries in the PSB after a trap, (eg. there are multiple threads running, at least two threads have

data in the PSB and one of the threads traps) the Pbox will send the invalidated data to the Qbox in subsequent cycles when there is room in the IQ to accept instructions. In other words, we won't collapse out invalid instructions if there are any valid instructions in the PSB.

We do cause a performance loss by sending invalidated map blocks to the Qbox in the case when a trap occurred to a thread that had blocks in the PSB. In the case when the machine is running a single thread, we would not need to send the invalidated map blocks since we could tell that the skid buffer is empty. To determine that it is empty, we need to OR across the instruction valid vectors. If all instruction-valid vectors are 0 then the PSB is empty!

4.3.9 RC/RS Interrupt Flag Widget (RIF)

4.3.9.1 Design Considerations

The Alpha SRM specifies two instructions that are to be used by code that translates or emulates VAX instructions. The two instructions are RC (for Read and Clear interrupt flag) and RS (for Read and Set interrupt flag). In real life on an in-order machine, RS Rx sets a bit called the interrupt flag and writes the previous state of the bit to register Rx. (RC Rx, as you might guess, clears the interrupt flag and saves the previous state to register Rx.) The SRM further says that this bit is cleared whenever a PALCALL_REI instruction is executed. PALCALL_REI is the PAL instruction sequence that is called whenever an interrupt service routine decides to return to the process that got the interrupt. Imagine the following code sequence:

```
Retry:  RS      R9
        do something that is probably wrong if an interrupt occurs
End:    RC      R9
        BLBC    R9,Retry
```

If an interrupt occurs and is serviced between Retry and End, then the interrupt flag will get cleared by the PALCALL_REI routine that returns back to our program segment.

This is not really all that hard on an in-order machine. 21464 is not an in-order machine. The interrupt flag must be maintained as in-order state.

21464 only knows about program order up to the point at which instructions are sent to the Qbox. (And then again, in the completion unit: we map dependencies in order, and retire instructions in order — everything else is higgledy-piggledy.) So it seems natural that we'd try to do the interrupt flag processing in the Pbox at map time (or soon thereafter). If this paragraph looks familiar, it is because I copied it from the description of the Store/Conditional Failure Widget (which no longer exists).

4.3.9.2 Design Architecture

The interrupt flag (we'll call it the INTerrupt Flag or INTF) is computed based on the speculated Istream. For this reason, the state of the flag must be "rolled back" when the Ibox erroneously predicts a path between an RS and an RC instruction (or, for that matter, between two RS's or two RC's).

Let INTF_CURRENT represent the current value of the interrupt flag. This value must be maintained on a per-thread basis. We also maintain an interrupt flag table, INTF_T, with an entry for each INum. INTF_T can be shared among threads, since the INum space is partitioned between threads in a non-overlapping manner.

Here are the rules for generating and maintaining the flag. Assume for now that we process instructions coming from the Ibox one at a time.

- At reset, clear INTF_CURRENT[3..0] (i.e. the current value for every thread).
- If instruction *X* in TPU *Y* is an RS, copy INTF_CURRENT[Y] to the INTF bit in the RS's payload. Set INTF_T[X] and INTF_CURRENT[Y].
- If instruction *X* is a RC, copy INTF_CURRENT[Y] to the INTF bit in the RC's payload. Clear INTF_T[X] and INTF_CURRENT[Y].
- For all other instructions *X*, copy INTF_CURRENT[Y] to INTF_T[X].
- On a trap where INum T in TPU Z is the trap point, copy INTF_T[T] to INTF_CURRENT[Z].

You will notice that we didn't mention PALCALL_REI. That's because PALCALL_REI is implemented as a stream of PAL instructions. Before the PALCALL_REI actually returns to the interrupted program, it will perform an RC R31 to clear the INTF bit as per the SRM.

4.3.10 Bid/Grant Exception Logic (BEL)

4.3.10.1 Design Considerations

The Pbox shares the responsibility with the Ibox for deciding which of all pending disruptions (exceptions, traps, interrupts, etc.) should be fielded. The function of the Bid/Grant Exception Logic (BEL) is to choose one execution-time or retire-time disruption per cycle out of the pool to trigger a chip-wide kill and send its INum to the Retire/Kill Unit (discussed below) which controls the Retire/Kill Bus (RK Bus). The BEL must also inform the Ibox so that it can make the final decision as to whether the kill will actually occur. Keeping track of execution-time disruptions for every in-flight instruction, and arbitrating between disruptions from different TPUs are among the challenges in implementing the BEL.

This discussion is fairly localized to the Pbox and may not shed much light on the chip-wide framework and arbitration mechanism for disruptions. For a better understanding of this context, see Section 22.1.

4.3.10.2 Design Architecture

The work of the BEL is simplified by the fact that the Qbox Completion Unit (CU) maintains the information associated with retire-time disruptions. The CU passes along no more than one candidate - the next instruction eligible to retire, if it has any associated disruptions - at a time. The BEL must keep track of the execution time disruptions for very in-flight instruction, and decide on a per-TPU basis which one is the oldest. Only the oldest in each TPU matters, since any younger disruptions are by definition on the bad path and will be killed off. This sorting by age is done in a decoded INum space. Disruptions on one TPU have no effect on another, since they represent independently executing programs at the hardware level. Nevertheless, the 21464 can only broadcast one kill at a time, which requires an arbitration mechanism. The BEL there-

fore hops in a round-robin fashion between those TPUs which are signalling any execution-time disruptions. If the CU is signalling a retire-time disruption on a given cycle, that disruption trumps the execution-time ones since it is, by definition, the oldest disruption in the machine. The winner of this arbitration process has its INum and TPU passed on to the Ibox and the Retire/Kill Unit. There is an additional dimension to kills, namely, whether the kill affects only the instructions younger than the one corresponding to the disruption INum, or includes that particular instruction as well. This ``kill at''/ ``kill after'' distinction is also passed on by the BEL.

4.3.11 Retire/Kill Unit (RKU)

4.3.11.1 Design Considerations

The Pbox owns the responsibility of driving the highest priority Retire INum or Kill INum to all the boxes on the chip that have state affected by instructions being killed and/or retired. The Retire/Kill Unit (RKU) controls the Retire/Kill Bus (RK Bus), which is the medium for communication of these events for the entire CPU. The structure of the RK Bus is such that only one kill or retire INum may be broadcast per cycle, which requires some means of arbitration. Also, since the Ibox is involved in the arbitration process, there are delays between when the BEL informs the Ibox of its choice for a kill and when the kill can be driven onto the RK Bus.

4.3.11.2 Design Architecture

Since the 21464 retires all instructions in order, there is only one possible candidate for retirement at any given point in time, winnowing down the choices for what to broadcast on the RK Bus. Further, as described above, the BEL funnels all pending kills down to one candidate per cycle. As the final arbitration mechanism, the RKU prioritizes kills over retires. If there is a valid kill and a valid retire in the same cycle, the kill will win and be broadcasted on the RK Bus, causing the retire to be stalled and to try again the next cycle.

The RKU has a stall pipeline that queues retire requests from the Qbox Completion Unit (CU). A Kill only becomes valid if the Ibox signals to the RKU that it may proceed. A pipeline in the RKU manages the latency between a kill being passed from the BEL and the valid/invalid indication returning from the Ibox. Retire-time exceptions (RTEs) are handled in a special way. The RKU first broadcasts them as a next-to-retire INum event, after which they are passed along to the BEL to be re-broadcast in the form of a kill.

The RK Bus sends the retire or kill INum and TPU, and whether the event is a kill or retire. The RK Bus also broadcasts whether a kill should occur at or after the retire/kill INum, or, in the case of a retire, whether it applies just to the INum in question or the entire INum block. For a detailed description of the RK Bus signalling protocol, please consult *Driving the Retire/Kill Bus (RK Bus)*.

5

Instruction Issue and Retire Unit — the Qbox

The Qbox processes instructions that are renamed by the Pbox and determines an appropriate schedule for those instructions. When all input operands for an instruction *x* have been produced or will be produced by an instruction *y*, already in the execution pipeline, we say that instruction *x* is "data ready". The Qbox selects the eight best "data ready" instructions for execution in eight integer pipeline units and four floating-point pipeline units. In addition, the Qbox selects up to four data-ready branch instructions for resolution in each cycle. It also retires all eligible instructions, committing them to architectural state.

The Qbox consists of the following components:

Table 5–1 Qbox Component Summary

Name	Mnemonic	Description	Described in Section
Queue Chunk Allocator/Deallocator	ALC	Manages the 32 instruction queue allocation chunks. Picks the two chunks to be allocated to the next group of eight instructions.	5.2.14
Instruction Queue	IQ	The queue from which instructions are picked for execution.	5.2.1
Queue Entry Table	QET	Translates INum dependencies delivered from the Pbox IMP stage into queue entry number dependencies. It also sets the No Live Dependency (NLD) bits which are set, for instance, when an instruction enters the queue data ready.	5.2.2
Dependency Arrays	DAs	Holds an identifier for the producer of each operand for each instruction in the queue.	5.2.3
Picker Arrays	PKs	On each cycle, chooses the oldest data ready instruction for each execution pipeline.	5.2.4
Bid Enable Logic	BID	Prevents otherwise ready instructions from bidding in pipes that cannot service them, either because of a slotting decision or because of non-data related resource conflict.	5.2.5
Completion Unit	CMP	Keeps track of which instructions have issued, which have passed their trap points, which are I/O instructions, and which have retired.	5.2.16
Destination Register Number Array	DRN	Holds the destination register specifiers for each instruction. This array are separately located from the SRN because it is not on any performance-critical paths.	5.2.9
Exception Kill Logic	EKC	Is repsonsible for removing from the Instruction Queue any instructions that have been killed due to an exception.	5.2.18

Table 5–1 Qbox Component Summary

Name	Mnemonic	Description	Described in Section
In-Flight Table	IFx	Keeps track of instructions that have issued and feeds INums which have passed their trap points to the Completion unit.	5.2.15
Load/Poison Re-arm Widget	LPR	Handles notification of load/miss events from the Mbox and ensures that all instructions that depend on a missed load will replay at some later time. The LPR also determines when individual instructions are eligible to be deallocated.	5.2.11
Load/Store Number High-Water Marker	HWM	Disables load and store instructions whose LSNums indicate that there may not be space available for them in the Mbox load/store queues. Also contains the logic for preserving the consistency of the DTB on misses.	5.2.10
Oldest CBR Selector	OCS	Identifies the oldest conditional branch issuing in the current cycle (that is, the one most likely to cause a misprediction).	5.2.13
Payload Array	PAY	Contains all the instructions and the register file addresses of all operands.	5.2.17
Post-Issue Logic	PIL	Gathers bubble requests and routes them to the appropriate pipelines. The PIL is also responsible for sequencing completion signals for the floating-point pipelines.	5.2.12
FPCR Control	FCR	Controls the update of the FPCR in the Fbox.	5.2.6
Profile-Me Data Collection	PFM	Collects the following instruction-time-oriented performance data for the two in-flight profile-me instructions: data ready, bid, issue, deallocation, and queue chunk deallocation.	5.2.7
Source Registers Number Arrays	SRNs	Contain the indices of the physical registers assigned to each source operand of each instruction. These arrays (there are two) are kept close to the dependence/bid/grant logic as the launch of the input physical register specifiers may be a critical path.	5.2.8

5.1 Scheduling Decisions — General Concepts

Our goal in the Qbox is to choose the ``best" 8 instructions to execute for each tic of the clock. The Qbox chooses these instructions from a ``window" of 128 candidates. Each of the eight scheduling pipelines can handle a subset of the 128 candidate instructions. Alas, the subset can contain (in some cases) up to half of the instructions in the window. So, the Qbox includes "pickers" that choose the best instruction out of a set of 64 candidates. Scheduling is a four step process:

1. Identify all data ready instructions.
2. For each pipe, select the ``oldest" data-ready instruction enabled for execution in that pipe.
3. Assert the result-ready signal that corresponds to each selected instruction, so that all instructions that are stored in the queue can see that the chosen instructions have been issued.
4. For each instruction in the queue, test the result-ready signal for each operand for each instruction in the queue. (The IMP in the Pbox has renamed each source virtual retister into the INum of its last writer. The QET renames these dependencies from INum space into queue entry space.

We can identify a data ready instruction by checking to see that both of its parent entries have asserted their result-ready signals. This scheme is called a "decoded-space" dependence array. Earlier plans called for an encoded-space scheme, but though such schemes are more scalable than decoded-space schemes, this comes at a cost in cycle time and complexity.The following sections describe the individual blocks that implement the general solution that has been described so far.

5.2 Component Details

5.2.1 Instruction Queue (IQ) Generalities

5.2.1.1 Design Considerations

The goal is to find every little bit of instruction level parallelism (ILP) in a program. (In multi-thread mode, we want to find all the ILP and all the parallelism between threads.) In the Pbox, we remove as many spurious dependencies as we can from the instruction stream by renaming the virtual registers specified in each instruction into a physical register space. This removes WAW (write-after-write) and WAR (write-after-read) dependencies. The task then is to pick from the collection of instructions that are in flight but have not yet issued, the best eight instructions to issue on the next cycle based on their actual RAW (read-after-write).

Picking eight instructions out of a pool of any reasonable size is a tough proposition no matter how you do it. The problem gets more difficult as the number of candidates that need to be examined increases. Back when dinosaurs roamed the earth we did a bunch of studies that showed (for single threaded applications) the point of diminishing returns (for performance vs. scheduling window size) seemed to occur around 96 to 128 entries in the window. So, 128 sounded like a good number. (More recent studies show that, when we consider all the other nitty gritty details, the point of diminishing returns is a little larger than 128.) The Qbox chooses 8 instructions on every tick from a pool of 128 instructions in the scheduling window.

The problem is complicated by the fact that the functional units of the Ebox and Fbox are divided into multiple clusters. Results from Ebox cluster 0 incur a one tic delay before they are available to clusters 1, 2, 3, 5, 6, and 7. (The results are available immediately in clusters 0 and 4.) The scheduling unit must take this into account without imposing "spurious" delays. If instruction *Y* depends on *X*, and both can execute in pipe 0, then *X* and *Y* should execute back-to-back in the absence of contention that might otherwise delay *Y*.

5.2.1.2 Design Architecture

This section has an implied familiarity with the Overview and Scheduling Decisions sections.

The 21464's solution to the scheduling problem did not spring forth all at once. It evolved over a few years of tinkering, experimenting, and brainstorming. Our solution features at is core a "persistent decoded space dependence array". The dependence array is the widget that keeps trace of RAW dependencies for the instructions in the queue. No matter how you cut it, this is a CAM. Each non-ready read operand for each instruction must CAM against the "result ready" signals for all of the issued instructions that are still in the instruction queue. When both operands for an instruction have seen a CAM match, the instruction will send its "bid" for a picker grant slot to the

"picker" units. In the 21464's decoded-space CAM one result-ready wire is associated with each instruction in the queue. When an instruction is granted (issues), it asserts its result-ready signal. In the 21464's persistent array, this signal stays asserted until the relevant instruction has left the queue.

Such decoded space arrangements normally consume lots of wire tracks. Our scheme makes use of some clever encoding tricks to reduce the width of the dependence array to a manageable size. A description of the tricks is beyond the scope here, but they are well documented in the "q_dax_arx" RTL code and in the detailed dependence array block diagrams. (See the Qbox implementation leader.)

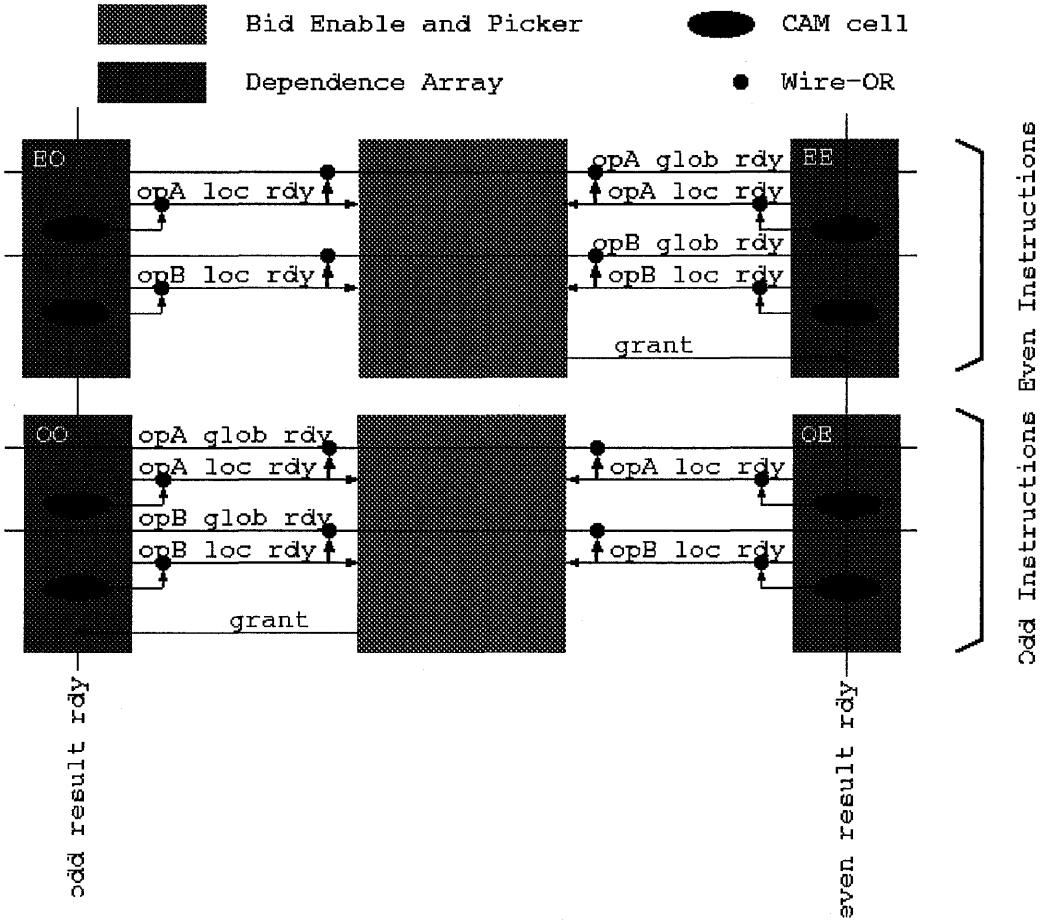
Our initial suspicion (which has been borne out in all of our subsequent circuit feasibility studies) was that the CAM-bid-grant (or more concisely, "bid-grant") loop was going to be very tight. This meant that we had to keep the width of the dependence array as small as possible.

The first insight was to split the logic that looks at all of the "bids" from the data ready instructions and generates the eight "grants" to the issued instructions into eight pieces. This reduces the problem from a "pick 8 of 128" to 8 problems of "pick 1 of 128". This looked like a good idea. But a pick 1 of 128 picker is much slower than a pick 1 of 64 relative to our cycle time. (A 10% hit in cycle time is not offset by the difference in performance between 8 pick 1 of 128 pickers and our scheme.) So, we divided the queue into two halves. The west half of the queue contains all the in-flight instructions with even INums, the east half of the queue contains the odd instructions. Each half of the queue picks instructions for four pipes. Each pipe has its own picker. So there are eight pickers, each picking the oldest instruction out of 64 in its half of the queue. Each picker picks instructions for just one functional unit.

Because of the one-to-one association with pickers and functional units, each picker considers bids from just those instructions that may execute in the corresponding pipe, but each picker sees CAM results (data ready signals) for all 64 instructions in its half of the queue. However, the picker's outputs -- the result-ready wires, are routed to the dependence arrays in both halves of the queue.

Figure 5–1 shows a simplified view of one half of the instruction queue. The top picker and dependence arrays are in the "west" or even half of the queue, the bottom picker and dependence arrays are in the "east" or odd half. The CAM entry for an instruction in the dependence array is loaded when the instruction is allocated into the queue. When the CAM entry detects that both of the instruction's operands have matched against the result ready wires, the entry sends a bid to the attached picker.

Figure 5–1 Simplified View of One-Half of the Instruction Queue



For now, let's assume that the 21464 only has two pipelines. All the odd instructions will be sent to the bottom or "odd" half of the IQ, and all the even instructions will be sent to the top half. (This picture is rotated relative to the actual chip layout -- we often refer to the even half as the "west" half and the odd half as the "east" half.) Assume that we have the following code segment:

INum	EntryNum	OpCode	Operands
80	20	SUB	R3,#5 -> R5
81	21	ADD	R5,R5 -> R9
82	22	ST	R5,(R9)

Note that we re-map from the INum space to Entry number space to make the comparison logic smaller. The SUB instruction is in the even half, the ADD in the odd half. The ADD instruction must wait until it sees that the instruction at entry number 20 has been issued.

So, let's assume that the SUB instruction is data ready. It sends its data ready signals to the bid-enable/picker logic. SUB instructions always bid as soon as they are ready, so the picker scans all outstanding bids and picks the bid from the "oldest" instruction. Eventually the SUB instruction wins the bid. The picker asserts the "grant" signal for

entry 20, which asserts the "even result ready" signal for entry 20. This result ready indication stays asserted until the SUB instruction is released from the queue. The indication is sent to the dependence array blocks labeled EE and OE in the diagram.

In the odd half of the queue, the ADD instruction has been waiting for entry number 20 to signal that it has been granted. Entry 21 in Array OE was loaded (when the ADD instruction entered the queue) with a 128 bit mask. For each of the instruction's two input operands. One and only one bit in each mask is set, corresponding to the entry number of the "parent" instruction for that operand. When the result of "ANDing" the mask with all the "result ready" signals is non-zero, the operand is data ready. Dependence array OE is responsible for the even bits in the mask for each odd numbered instruction. (OO holds the odd bits for odd instructions, and so on.) One cycle after the SUB instruction was granted (or issued) the ADD instruction will become data ready, bid, and be granted if it is the oldest bidding instruction from the odd half of the queue.

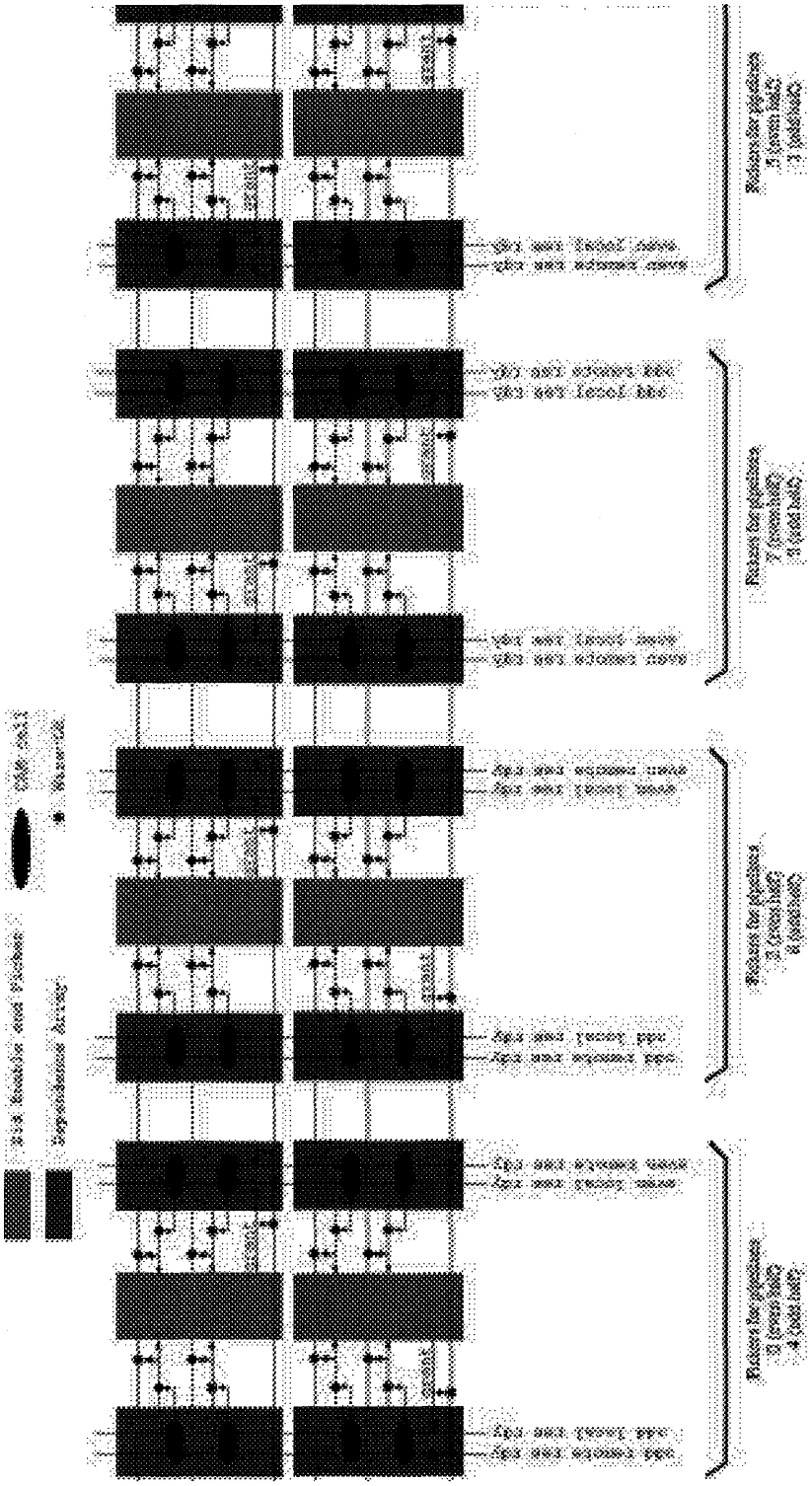
Finally, the Store instruction has been waiting for both the ADD and the SUB to issue. Its A operand became data ready at the start of cycle 2 when the SUB was issued. The EE dependence array noted that the SUB had issued, and sent the operand A data ready signal into the picker. The B operand became data ready in cycle 3 when the ADD was issued. This was noted by the EO dependence array. The STore then issues in cycle 3 when it is picked by the even picker.

The 21464, of course, has eight pipelines. For the moment, assume that each instruction is assigned to just one pipeline (and, thus, one picker) by the Pbox instruction decode logic. The figure below shows all eight pickers and the sixteen dependence arrays that make up the core of the 21464's instruction queue.

To illustrate, consider our SUB instruction 20 that has been granted by a picker and has a consumer -- ADD instruction 21. When 20 issues, every instruction in the queue tests to see if its operands have become data ready. As a result of this test, the entry containing the ADD will notice that both its operands will be data ready as a result of the SUB.

So, returning to our example, instruction 20 might be picked by the picker for pipe 7, while instruction 21 can only execute in pipe 1. How does the grant information from picker 7 get to picker 1? Via a global result ready signal, of course. When the SUB instruction issues, it will assert its "global result ready" signal one cycle after it asserts its "local result ready" signal.

Figure 5–2 Simplified View of Full Instruction Queue



Note that because the SUB and ADD are executing in different pipelines we incur a one cycle penalty (similar to the EV-6 "cross cluster" penalty). In this scheme, the ADD won't issue until cycle 3 even though the result from the SUB may be available early enough to allow the ADD to issue in cycle 2. This penalty was seen as a big deal. So we explored some other ideas.

One way to mitigate this problem is to group pairs of pickers together. This way, each picker in the pair sees the CAM results from each entry's match against its own granted INum and the INum granted by the other picker in the pair. This works pretty nicely. Now instead of having eight opportunities to incur the cross-picker penalty, we only have 4 opportunities. If the SUB is "slotted" to picker 7 and the ADD is slotted for picker 3, we would encounter no "cross-cluster" penalty.

A second way to mitigate the problem got the name "follow me" picking. To illustrate: imagine that *Y* depends on *X*, that *X* must execute in pipeline 7, and that *Y* could execute in either pipeline 7 or pipeline 1. We need to be careful about where *Y* executes however. Imagine it became data ready in pickers 1 and 7 at the same time. Then it might issue from both at the same time. We'd probably get the correct answer, but at the very least, we'd waste an issue cycle in one of the pipelines. So, our instruction slotter picks a pipeline in which we'd prefer to execute *Y*. Assume for now that the slotter picked pipeline 1. With this arrangement we'd still incur the cross-picker penalty. But notice that *Y* becomes data ready in picker 7 one full tick before it becomes data ready in picker 1. If we allow *Y* to issue from picker 7 ONLY in the first tick after its operand became data ready from an instruction issued by picker 7, then there is no chance that it would be issued by pickers 7 and 1 at the same time. (Since picker 1 won't know about *Y* being data ready until after picker 1 samples the global data ready signal.)

Up until now, our discussion has ignored the fact that load instructions have a three cycle latency -- that is, their result data is not available until the start of the third cycle after the start of the load's Execute (E) cycle. This means that if a LD instruction is issued in cycle 0, its dependents can't be issued until cycle 3. For this reason, when a LOAD instruction is issued, it does not assert its "result ready" wire until three ticks have passed. It will assert the "global result ready" wire at the same time it asserts the "local result ready" wire. Why? Well, we know that the load data arrives at the inputs to all of the functional units in the Fbox and Ebox at the same time. Therefore, by asserting global data ready one tick earlier than it normally would be asserted, the other functional units can grab load data as soon as it is ready.

We should note that the Mbox supports three load ports. Because of our odd/even distribution of instructions into the pickers, we need to find some way of allocating the extra load port. On even ticks, load instructions in some even positions in the map block may issue to this "weak" load port. On odd ticks, loads in certain odd positions may issue to the "weak" port.

Entries in the queue are allocated in groups of 4 instructions called "queue chunks". The IQ is full if either the even half or the odd half of the queue has no available queue chunks. An instruction stays in the IQ until we are certain that no instructions in its chunk will need to be replayed as the result of a load-miss or other mishap.

5.2.2 Queue Entry Table (QET) and Reallocation Logic (RAL)

5.2.2.1 Design Considerations

The Pbox produces dependence information based on the INum of the instruction that produced each input operand for a mapped instruction. That is, an instruction like ADDQ R3,R2,R1 will pass through the Ibox and have its input registers (R1 and R2) remapped into INum space. With our decoded space dependence array, it would be both un-necessary and slow to represent the entire INum range in the instruction queue itself. After all, only 128 instructions can be in the queue at the same time, why do we need to reflect completion for the entire 256 (or 512) instruction range?

So, we need to transform INum based dependencies into a more compact form, as the speed of a decoded space scheme is dependent on keeping the dependency checking logic as small as possible.

5.2.2.2 Design Architecture

5.2.2.2.1 Algorithm

The Queue Entry Table transforms INum dependencies into EntryNum dependencies. When an instruction passes through the QET, the INum of each operand's parent is used to index a table that indicates the position of the parent instruction in the instruction queue. If the parent instruction is no longer in the instruction queue, we know that the associated operand is now data-ready.

The lookup is most easily described as

```
for(i = 0; i < 8; i++) {
    inst[i].src_a_entry_num<6:0> = EntryTable[ inst[i].src_a_inum];
    inst[i].src_b_entry_num<6:0> = EntryTable[ inst[i].src_b_inum];
}
```

As it turns out, the table does not need to be quite so large. We know the low three bits of the entry num for any parent INum: they are identical.

```
Enum<2:0> = INum<2:0>
```

Further, we want the entry number in decoded form to make things convenient for the IQ core. Finally, we need to send all the ODD parent dependencies to the ODD dependency arrays, while the EVEN dependencies go to the EVEN arrays. (We split the QET and Dependency arrays into EVEN and ODD parent dependencies to speed up the bid/grant loop.) So, we're really just translating the INum BLOCK bits from INum space to entry number.

```
for(i = 0; i < 8; i++) {
    if(inst[i].src_a_inum<0>) {
            inst[i].odd_src_a_entry_msk<15:0> =
                    OddEntryTable[ inst[i].src_a_inum<7:3>];
            inst[i].even_src_a_entry_msk<15:0> = 0;
    }
    else {
            inst[i].odd_src_a_entry_msk<15:0> = 0;
            inst[i].even_src_a_entry_msk<15:0> =
```

```
                    EvenEntryTable[ inst[i].src_a_inum<7:3>];
    }
    if(inst[i].src_b_inum<0>) {
            inst[i].odd_src_b_entry_msk<15:0> =
                    OddEntryTable[ inst[i].src_b_inum<7:3>];
            inst[i].even_src_b_entry_msk<15:0> = 0;
    }
    else {
            inst[i].odd_src_b_entry_msk<15:0> = 0;
            inst[i].even_src_b_entry_msk<15:0> =
                    EvenEntryTable[ inst[i].src_b_inum<7:3>];
    }
}
```

For the most part, this is a rather simple operation: a RAM lookup. But there are two problems that must be addressed.

1. What if instruction 51 depends on instruction 48? They are both in the same map block, so when 51 arrives, neither it nor 48 have yet been entered into the Queue?
2. What if instruction 51 depends on instruction 2 which is leaving the queue just as instruction 51 arrives at the ET?

We solve the first problem by updating the QET map (the table that maps INums to ENums) during cycle Q0, while we don't actually translate the instructions that will be mapped until cycle Q1.

The solution second problem is not so simple. We solve it by adding an extra bit of information to the parent information for each operand. This extra bit is called the "No Live Dependency" or NLD bit. Each "chunk" of INums in the map (4 entries comprise a chunk in the IQ) has a stored NLD bit. This is read each time we translate an INum parent into an entry num parent. SRCx_NLD is set for the x (x is A or B) operand if its parent is no longer in the instruction queue.

Note that each entry in the entry tables has an associated NLD bit that indicates that all INums in this chunk have already issued and left the queue. The NLD bit is set for INum<7:3,0> when the entry for that INum chunk in the IQ is re-allocated to a new chunk of instructions. (So we don't actually set the NLD bit when an instruction is "done", but rather when we need the space that it formerly occupied.) We clear the NLD bit for a chunk of INums when that chunk is loaded into the IQ.

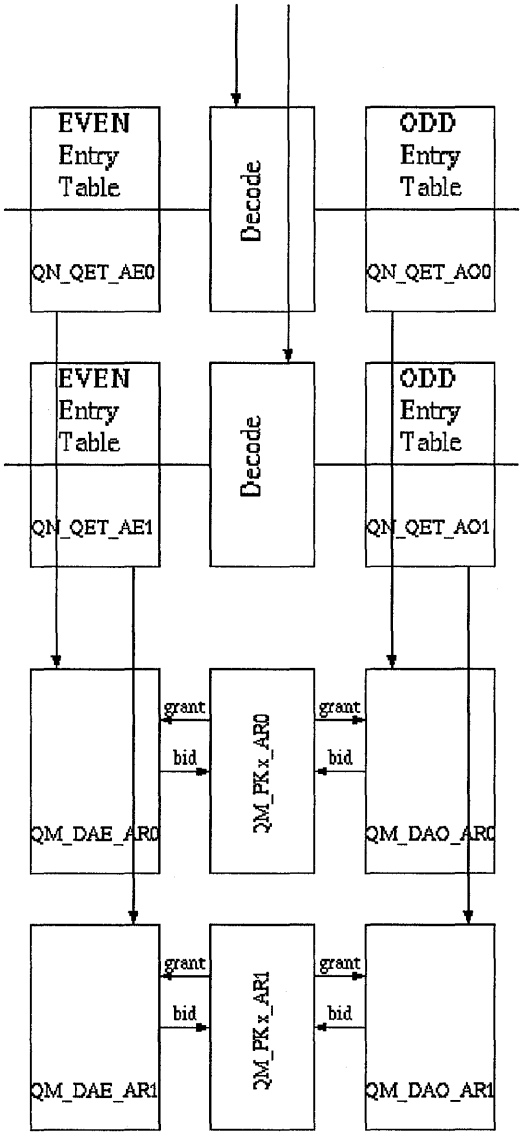
Deriving the signals necessary to clear the NLD bit on re-allocation is entwined in the details of queue chunk allocation and deallocation. When a queue chunk is re-allocated (that is when the "write_chunk" signal is asserted for the chunk), the RAL section in the IQ core sends the INum block number associated with the re-allocated chunk back to the QET. The QET then sets the NLD bit associated with that INum block.

5.2.2.3 Physical Organization

The QET is built from a block that is replicated eight times. Each block decodes parent INums for HALF of the instructions entering the queue. (The west blocks decode parent INums for even instructions, and the east blocks decode parent INums for odd instructions.) Each block corresponds to one issue pipeline in the instruction queue core.

There are four instances of the QET schematic section. Each section contains two blocks (even and odd) and translates instructions bound for an associated Qx_DAE and Qx_DAO sections as shown in the next figure. All four instances are identical with identical inputs and outputs. The replication is an aid to routing and is required to limit the transit time through the QET stage.

Figure 5–3 Simplified Diagram of QET and Pickers for Two Pipelines



The RAL (reallocation logic) is contained in the IQ core itself.

5.2.3 Dependency Arrays (DAs)

5.2.3.1 Design Considerations

Once an instruction has entered the IQ it must wait until both of its operands have become data ready. We need to detect that an operand's parent has issued and, to support the followme scheduling technique, we need to know whether the parent issued in a local pipeline or in a different execution cluster. The detection scheme must also take into account the different instruction latencies for loads, integer operations, and floating point operations.

5.2.3.2 Design Architecture

Our strategy is to represent the dependence of an operand on a parent instruction as a 128 bit vector. Bit *X* in the vector is set if and only if the operand is produced by the instruction at entry number *X* in the instruction queue. When an instruction is issued we assert a "result_ready" wire corresponding to the instruction's entry position in the IQ. There are actually two result_ready signals for each instruction in each dependence array. One result_ready wire (local_result_ready) indicates that entry *X* has issued and dependents may now issue in the same Ebox/Fbox cluster that *X* issued to. The second result_ready wire (global_result_ready) indicates that dependents on *X* may issue in any cluster (i.e. the cross cluster penalty has been paid).

The algorithm can be depicted as

```
for(e = 0; e < 128; e++) {
    entry[e].srca_lcl_rdy =
            (local_result_ready<127:0> & entry[e].srca_entry_mask<127:0> != 0);
    entry[e].srcb_lcl_rdy =
            (local_result_ready<127:0> & entry[e].srcb_entry_mask<127:0> != 0);
     entry[e].srca_glb_rdy =
            (global_result_ready<127:0> & entry[e].srca_entry_mask<127:0> != 0);
    entry[e].srcb_glb_rdy =
            (global_result_ready<127:0> & entry[e].srcb_entry_mask<127:0> != 0);
}
```

But again, we use the fact that INum<2:0> is the same as ENum<2:0> to save some wires when we load the dependence array, so that we need not store a 128 bit entry mask. Further, we divide the dependence array entry for a given position in the queue into two halves. The EVEN dependence array (shown on the floor plan as Qx_DAE) checks for dependencies on instructions in the EVEN half of the IQ core. The ODD array checks for dependencies on ODD instructions. This arrangement mirrors the division of the QET tables into even/odd halves. So, the actual algorithm looks like this:

```
for(e = 0; e < 128; e++) {
  entry[e].srca_lcl_rdy_odd =
          (local_result_ready<127:1:2> & entry[e].odd_srca_entry_mask<63:0> != 0);
  entry[e].srcb_lcl_rdy_odd =
          (local_result_ready<127:1:2> & entry[e].odd_srcb_entry_mask<63:0> != 0);


  entry[e].srca_glb_rdy_odd =
```

```
        (global_result_ready<127:1:2> & entry[e].odd_srca_entry_mask<63:0> != 0);
    entry[e].srcb_glb_rdy_odd =
        (global_result_ready<127:1:2> & entry[e].odd_srcb_entry_mask<63:0> != 0);

    entry[e].srca_lcl_rdy_even =
        (local_result_ready<126:0:2> & entry[e].even_srca_entry_mask<63:0> != 0);
    entry[e].srcb_lcl_rdy_even =
        (local_result_ready<126:0:2> & entry[e].even_srcb_entry_mask<63:0> != 0);

    entry[e].srca_glb_rdy_even =
        (global_result_ready<126:0:2> & entry[e].even_srca_entry_mask<63:0> != 0);
    entry[e].srcb_glb_rdy_even =
        (global_result_ready<126:0:2> & entry[e].even_srcb_entry_mask<63:0> != 0);
}
```

5.2.3.3 Physical Organization

The dependence arrays are built from a block that is replicated sixteen times. Each execution pipeline picker is connected to an EVEN half dependence array and and ODD half dependence array. (See the floor plan).

5.2.4 Picker Arrays (PKs)

5.2.4.1 Design Considerations

Given that we've found a set of instructions that are ready to issue, we need to choose the "best" from the set to send to an execution unit. As described in the overview section, we divide the instruction queue into eight blocks. Each block keeps track of instruction dependencies for all instructions in the queue and chooses one data ready instruction on each tic to send to a particular execution pipeline. The core of this decision process is called the "bid/grant loop". The bid/grant loop is implemented, for the most part, in the dependence arrays and the picker arrays.

Choosing the "best" instruction is an optimization problem. It is most likely not computable in bounded time. For this reason, we adopt a simple heuristic: we choose the oldest data ready instruction for a pipeline on each tic. Choosing the oldest is a simple algorithm, and we've got a simple implementation.

5.2.4.2 Design Architecture

As an instruction Z enters the instruction queue it is given a bidding token. The token is sixty-four bits wide and has all its bits set to 1 except for the bits corresponding to the Z's entry number and all the instructions in Z's chunk that are before instruction Z. Each time a new chunk *Y* is written into the queue, every instruction will clear bits <4*Y+3:4*Y> in its bidding token.

When an instruction Z bids, it performs a bitwise AND between Z's bidding token and all other bids in this picker. If the result is zero, then Z wins this round of bidding.

This mechanism guarantees that the oldest bidding instruction (age being determined by time-of-entry into the instruction queue) will win any bid.

5.2.5 Bid Enable Logic (BID)

5.2.5.1 Design Considerations

Even though both of an instruction's operands are data ready, the instruction may not be able to bid for a given pipeline. First, the resource required to serve the instruction may not be available. (For example, floating point square root and divide operations are not pipelined. Additionally, resources in the Mbox are limited, so some load and store instructions may be prevented from bidding until they fall under the "high water mark" -- see the "High Water Mark" widget.)

Further, an instruction that bids in one of its "followme" pickers, may, unfortunately, bid in its preferred picker on the very next cycle. The solution to this particular problem is still under discussion.

5.2.5.2 Design Architecture

The bid enable logic keeps track of issue preconditions that are separate from the data-readiness of an instruction's operands. In particular, the bid-enable logic monitors the results of high-water-mark comparisons, the occupancy of the non-pipelined floating point pipes, and, for resources that "ping-pong" between halves of the issue queue (e.g. the weak load pipe is shared between two pickers -- on "odd" cycles it can be used by a picker that chooses among odd instructions, on "even" cycles it can be used by an even instruction picker) such as loads, JSR, and some MTPR operations.

5.2.5.3 Physical Organization

The bid enable logic is replicated in two copies. One serves the four pipelines in the south of the instruction queue, the other serves the north pickers. The logic is identical and replicated for electrical reasons.

5.2.6 FPCR Control Unit (FCR)

The floating-point control register (FPCR) control unit controls the update of the FPCR in the Fbox. FPCR is implemented as a speculative-committed pair of registers. First, the FCR ensures that only the oldest in-flight MT_FPCR instruction in a TPU will update the speculative FPCR. Second, the FCR updates the committed FPCR from the speculative FPCR when the oldest in-flight MT_FPCR instruction in a TPU becomes retirable. This mechanism, along with the native mode FPCR trap and PAL mode fetch barrier, guarantees the correct architecture (in-order) behavior of writing and reading the FPCR register.

5.2.7 Profile-Me Data Collection (PRM)

The IQ Profile-me data collection unit collects performance data for the two in-flight profile-me instructions. The data collected in this section include instruction data ready time, instruction bid time, instruction issue time, instruction de-allocation time and the instruction queue chunk de-allocation time. To be more specific, the real data collection storage for the data collected in this section is in the Ibox. PFM is mainly responsible for generating the control signals for the Ibox to capture the cycle time information so that the profile-me software can calculate all those data mentioned above.

5.2.8 Source Register Number Arrays (SRNs)

5.2.8.1 Design Considerations

The 21464 is blessed with a very large Register File which has a non-trivial access time. When an instruction issues, its source register IDs need to be sent to the register file to begin looking up the values as early as possible. The Register File also runs extremely hot; anything that can be done to save power by averting unnecessary lookups is a boon to the chip.

5.2.8.2 Design Architecture

The Source Register Number Arrays (SRNs) store the renamed source physical register (PReg) IDs for each instruction. There are actually two SRN sections; one covers the PRegA and PRegB values for instructions in even map block positions, and the other covers the ones in odd map block positions.

Associated with each PReg ID is a valid bit. Register numbers with their valid bit deasserted do not cause a lookup in the Register File (or the Ebox register cache), thus saving power. These bits are deasserted in cases where there is no valid instruction in that issue block position, the source does not exist for that instruction (e.g. LDQ has no valid PRegA), or the source value is zero (e.g. Ra == R31). Since the Ebox does its own instruction decoding, it knows the difference between these cases - i.e. when to simply ignore the value returned by the Register File and when to substitute in a vector of zeroes. The values of these bits are determined by the Pbox.

Because of their critical timing, the SRNs sit in the IQ core itself rather than in the "late" IQ, and send out their data to the Register File and Ebox as soon after instruction grant as is physically possible, before any of the other information associated with an instruction leaves the box.

5.2.9 Destination Register Number Array (DRN)

5.2.9.1 Design Considerations

The register file needs physical register indices for instruction destinations, not just sources (see the Source Register Number Arrays description), in order to store operation results. The timing is somewhat more relaxed as result writeback happens later in the pipeline than source reads. Similar power issues apply, but the urgency is reduced by the fact that each instruction has a maximum of two source operands but at most one destination. A more pressing problem is the fact that superfluous reads are merely wasteful, but spurious writes are destructive.

5.2.9.2 Design Architecture

There is a single Destination Register Number Array (DRN) section which stores the renamed destination physical register (PRegD) IDs for all instructions in the IQ. It sits in the "late" IQ partition, to the side of the IQ core, as its timing is aligned with the bulk of the information going from the Qbox to the execution boxes. When an instruction issues, its PRegD value is forwarded to the Register File and Ebox to address the register file and register cache, respectively.

Each PRegD ID has an accompanying valid bit. If this bit is deasserted, it means either that the instruction has no destination register (as in the case of an ordinary store), or that the destination register is R31. In either case, the Register File and Ebox will not write the register file/cache for that instruction. They rely entirely on the Pbox to correctly assert these bits to avoid spurious or dropped writes.

5.2.10 Load/Store Number High-Water Marker (HWM)

5.2.10.1 Design Considerations

As documented in the description of the Pbox Load/Store Serial Number Allocator, the aggressive, out-of-order execution of the Mbox, combined with a finite number of load and store queue entries, can lead to deadlock, unless there is some way to guarantee dependencies are resolved before the queues fill up. The Pbox addresses this by assigning a 8-bit serial number — an LSNum — to all memory operations. Each operation type has its own serial number class: loads get LNums, and stores get SNums. Each TPU also has its own LNum and SNum spaces.

The Mbox keeps track of the fullness of its load and store queues on every cycle, and sends the Qbox a per-TPU, per type (load or store) "high-water mark" value. The Load/Store Number High-Water Marker (HWM) must insure that only memory operations below the applicable high-water mark can issue.

On a different but related subject, there are also a number of challenges associated with maintaining the consistency of virtual to physical memory mappings while a DTB miss is in the midst of being processed. This problem and our general policy for solving it are at length in How the 21464 Does DTB Fills. To summarize the important points, DTB misses lead to a PAL flow which services the miss. The flow contains a so-called "DTB writer block", a group of instructions which put the new translation into the "speculative" DTB entry (there is one per TPU). Each TPU sees its own speculative entry as well as the common, committed DTB state. When the DTB writer block retires, the contents of its TPU's speculative entry are written into the committed state.

While this new translation is being written into the DTB, any memory operations which depend on this new translation must not be allowed to issue since they will lead to more misses, or, even worse, potentially erroneous behavior. This process is further complicated by the fact that there are actually two copies of the DTB - one accessed through each of the two Mbox strong load ports - which must be kept coherent. The DTB writer logic in the HWM must prevent memory instructions which are "DTB-dependent" from issuing, and handle the situation when bad path code generates spurious DTB writer blocks.

5.2.10.2 Design Architecture

The HWM affects the issue behavior of memory instructions through the "load/store bid enable" (LDST_BIDEN) signals - one per instruction in the queue - which it passes along to the Bid Enable Logic. Queue entries with their LDST_BIDEN signals asserted are free to bid, provided the other conditions enforced by the Bid Enable Logic (e.g. data readiness) are satisfied. Entries with deasserted values may not bid, with the exception of their very first cycle in the queue. Computing the correct LDST_BIDEN values takes a cycle. Since most queue entries are not memory ops, and since it turns out that the majority of incoming memory ops are eligible to bid on entry into the queue, the Bid Enable Logic speculatively assumes that the LDST_BIDEN signal for

each entry is true on allocation. If that assumption should prove false, and the instruction gets granted in the meantime, there is enough time to shoot it down before it leaves the IQ.

The LDST_BIDEN signals are always asserted for instructions which are neither loads nor stores; the HWM knows which these are by virtue of ISLOAD and ISSTORE signals conveyed by the Pbox. These are maintained on a per-entry basis, while the TPU ID is stored on a per-chunk basis. Note that ISLOAD and ISSTORE are asserted for any instruction that executes in an Mbox load queue or store queue, respectively, and has a valid LSNum. Not all memory operations are loads or stores in the conventional sense (e.g. WH64).

LDST_BIDEN is deasserted for loads and stores which are below their high-water mark, as determined by a comparison of the the LSNum of the individual memory operation in a given entry and the high-water mark for its chunk TPU. The HWM receives a high-water mark update from the Mbox for each TPU on every cycle. Once a memory operation falls below its high-water mark, its LDST_BIDEN signal remains asserted until that chunk is reallocated.

Finally, LDST_BIDEN is deasserted for any memory operation which is DTB-dependent. For implementation purposes, we use a somewhat coarse definition of this concept: any memory operation which is allocated after a valid, active DTB writer block in the same TPU is considered DTB-dependent on that block. This includes, unfortunately, operations which do not actually use the translation being modified by the DTB writer block. But in the average case, the memory operations that are mapped after a DTB miss will be dependent on same page as the reference that missed.

When it is known, in the due course of time, that every instruction in the entire DTB writer block - in both halves of the IQ - is not poisoned (i.e. not the victim of a missed load), the DTB-dependent instructions are free to bid and be granted. The Load/Poison Re-arm Widget (LPR) indicates via its NO_REISSUE signals that a DTB writer instruction has passed its poison point. (For a general understanding of the concept of "poison", consult An Overview of the Poison Mechanism in the 21464.) The dependent instructions will read from the speculative entry for their TPU until such time as the DTB writer block retires and is written to the committed state of the DTB. Throughout this entire process, memory operations in other TPUs are free to bid and issue, as are memory ops in the same TPU which are older than the DTB writer. Non-memory operations are completely unaffected.

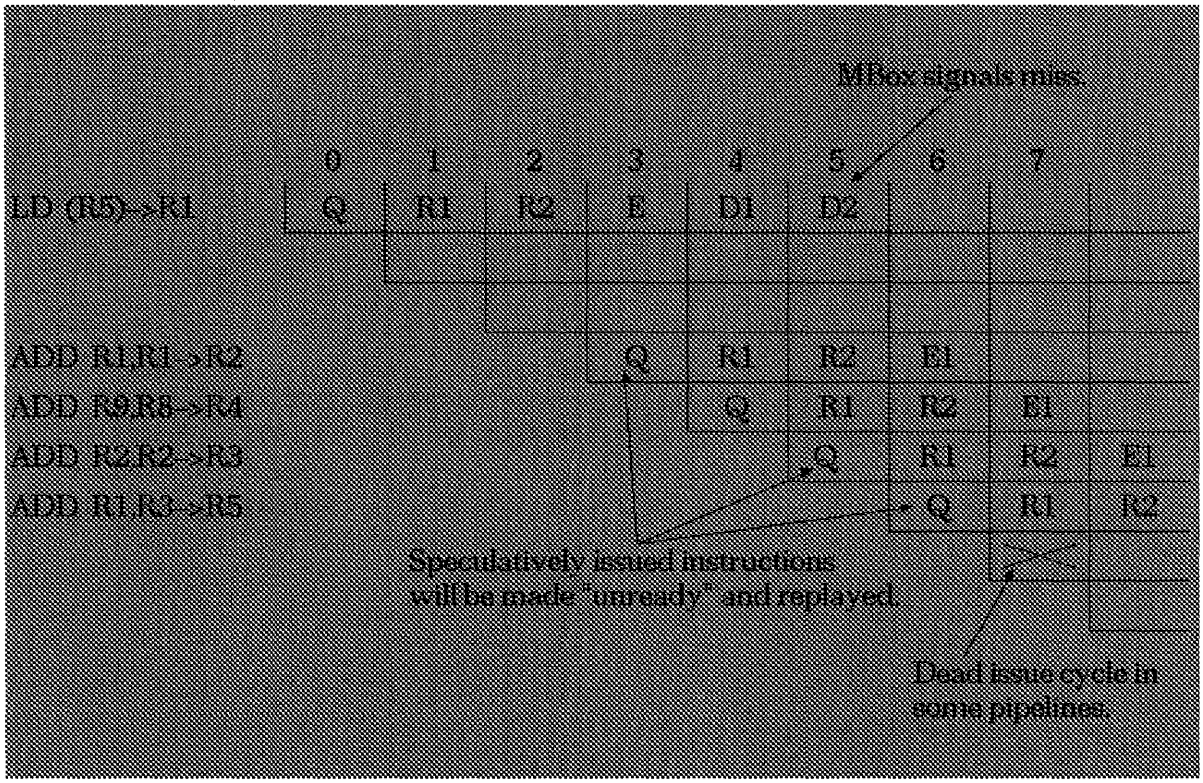
The DTB logic in the HWM tracks DTB dependencies throughout the IQ in each TPU, and deasserts the LDST_BIDEN signal of every DTB-dependent entry until it sees every applicable NO_REISSUE signal. To make this task easier, it receives "DTB writer" (DTBWRT) flags from the Pbox indicating which instructions are members of a DTB writer block. It also handles the important and dangerous scenario of spurious DTB writers entering the queue when a valid one is already active. There is only one signal per TPU indicating that there is a valid DTB writer in the queue. If a spurious DTB writer were to come in after a valid one and seize control of that signal, it is possible that older instructions could become dependent on this younger DTB writer, leading to deadlock. To avoid this situation, the HWM DTB logic ignores DTB writers entering the queue while there is already an active one in the same TPU, since these are known *á prioi* to be on a bad path.

5.2.11 Load/Poison Re-Arm Widget (LPR)

5.2.11.1 Design Considerations

Consider Figure 5–4.

Figure 5–4 Tracking Data-Ready Instructions



Remember the fundamental rule for hyper-complex super-scalar deep-pipelined hell-bent-for-leather microprocessor design: if you aren't sure, guess. So, when the Qbox issues a load instruction we always guess that the load will hit in the first level cache. This is a pretty good guess, as it is correct far more often than it is wrong. But if we're going to guess, we need some way of backing out of incorrect guesses. The Load/Poison Re-arm Widget (LPR) keeps track of all instructions that became data ready because we guessed that a load would hit. In Figure 5–4, instruction 1 is a load. The Qbox scheduled its first dependent, instruction 2, in tic 3 assuming that the load would return data at the beginning of instruction 2's execute phase. Instruction 2 has a latency of 1 cycle and caused instruction 4 to become data ready in cycle 3, though it didn't issue until cycle 5. When instruction 4 issued it caused instruction 5 to become data ready in cycle 5 and then issue (broadcast its INum) in cycle 6. When instruction 1 missed in the cache, all the instructions that depended on the load (or whose ancestors depended on the load) operated on "bad" data.

Fortunately, because of register renaming and all kinds of neat stuff that happens in the Mbox, we don't need to worry a whole lot about the bad data that the load shadow instructions write. It will be overwritten later or it will be ignored completely. We do need to replay the instructions however.

In order to replay the instructions in the load miss shadow we need to identify them. The 21264 declared that all instructions that issued in the shadow of a load miss would be "replayed". Using this approach, instructions 2, 3, 4, and 5 would all be re-issued at the appropriate time. This works well for the 21264 becaue they don't issue a whole lot of instructions in the shadow of a load. But notice that instruction 3 didn't depend on the missed load at all; the 21264 would replay this instruction unnecessarily. But for a short load shadow the cost is relatively low.

The 21464, on the other hand, could issue lots of instructions in the shadow of a load, many of which aren't related to the load. (Some might not even be in the same TPU). Replaying all of them would simply be too expensive. Instead, we replay only those instructions that are actual dependents of a missed load.

In order to be replayed, instructions either have to stay in the queue up to the point they are re-armed or, if they are allowed to deallocate on issue, be injected back into the queue. The latter alternative is essentially tantamount to a trap, which is too expensive an event for something that happens as relatively frequently as a load miss. But not only is queue space is limited, but it is allocated and deallocated on a chunk granularity, and our performance is very sensitive to how long a chunk stays in the IQ. Therefore, we need to remove instructions from the IQ as soon as we know that they may no longer be victimized by a missed load. The Load/Poison Re-arm Widget (LPR) tells the IQ core which entries contain instructions that are victims of missed loads and need to be re-armed, and which entries are free to be reallocated.

5.2.11.2 Design Architecture

The problem of how to identify which instructions are actual victims of a missed load is solved by "poison". To learn more about poison across the chip, consult An Overview of the Poison Mechanism in the 21464. The basic idea is that the property of a load missing in the cache is propagated from the load to all instructions that issue and consume its data, to all of their issuing descendents, and so on, and so forth - just like the data itself. Instructions that are poisoned need to be re-armed and replayed. This is achieved by resetting their data readiness to its original state and then reissuing the culprit load via a bubble. When the load's result becomes ready once again, its dependents will become data ready, bid, and eventually issue, and so on down the line.

Note that this cycle may repeat several times - for instance, if the load misses in both the first-level and second-level caches. Each failed attempt returns a poisoned value, which leads to a chain of poisonings, re-arming, a new bubble, and so on, until the load finally comes back poison-free.

The Load/Poison Re-arm Widget receives the poison information from the Ebox and passes it to the Dependency Arrays in the form of REARM_A and REARM_B signals. These indicate if an instruction's A or B operand, respectively, has been poisoned. The LPR additionally sends out a "reset result ready" (RESET_RES_RDY) signal to each poisoned instruction, indicating that it should clear the "result ready" (RES_RDY) state signaling to the rest of the IQ that it has produced a valid result. The RES_RDY bits will be set again as a consequence of instruction replay.

The Mbox also communicates with the LPR, indicating when a load has missed via a WILL_RETRY signal. The LPR asserts the RESET_RES_RDY signal for load which misses, poisoned or not. Missed loads will set their RES_RDY state again when they are reissued via a bubble, triggering a chain of reissues.

The LPR is additionally responsible for telling the Queue Chunk Allocator/Deallocator (ALC) when individual instructions are eligible for deallocation. This is a function of both poison and instruction type. Single-cycle operations, for example, can deallocate immediately after their "poison point" - i.e. the point in the pipeline where the Ebox returns poison information. Loads, by contrast, must wait until the Mbox has indicated whether they have hit or missed in the cache. Long-latency operations must wait until either their poison point or the time they bubble, whichever comes later. Multicycle operations must linger in the queue an extra L-1 cycles (where L is instruction latency) after producing a result - enough time for any poisoned descendents to be replayed and see the new RES_RDY signal. The ALC asserts an "okay to deallocate" (OK_DEALC) signal to the ALC for each instruction that may leave the queue.

Finally, the DTB logic in the Load/Store Number High-Water Marker (HWM) needs to know when a DTB writer block had passed its poison point in order to free any DTB-dependent chunks (see the HWM description for more information). The LPR sends NO_REISSUE signals to the HWM to indicate instructions that have passed their poison points and thus will not reissue; the HWM checks this information against its record of which instructions are elements of a DTB writer block to determine when to release dependent memory operations.

5.2.12 Post-Issue Logic (PIL)

5.2.12.1 Design Considerations

So what happens after a load has missed and the Mbox eventually gets the correct data from the second level cache or the system? How does the Qbox re-fire all of the load's dependent instructions? (They were re-armed by the Load/Poison Re-Arm Widget.)

For that matter, how do we handle long latency operations like square root or divide? These operations produce results many cycles after other concurrently issuing instructions are ready to write theirs back to the register file. We need to let fast operations deliver their results quickly - it would be absurd to stage out the results of 1-cycle integer adds to line up with those of 14-cycle floating-point divides. But when long-latency ops eventually do complete, they need to have exclusive access to a result bus and register file port. (Building separate buses and ports just for long-latency ops is far too costly.) This means that the issue slot for the instruction that would otherwise consume those resources at that point in time must be empty. That is, there needs to be a "bubble" in the pipeline into which the completing long-latency operation can slip its result.

5.2.12.2 Design Architecture

In both cases the IQ provides a "bubble request widget", which sits in the Post-Issue Logic (PIL) for operations that need to signal late completion. In the case of load misses, bubble requests are fielded by the strong load pick rs in the same bank of the queue where the instruction originated. The bubble request will cause the load to set its "result ready" (RES_RDY) bit and allow all its dependents to become data ready. Note that load miss bubble requests can still result in a cache miss, so the Load/Poison Re-arm Widget (LPR) must be prepared to kill dependents of a load bubble request.

The Mbox first gives early warning of an impending bubble by asserting the WILL_RETRY signal to the issuing picker, causing the LPR to signal to the Dependency Arrays that the corresponding RES_RDY state must be cleared. Later, the Mbox sends along the actual bubble request in the form of the IQ entry number of the load

which needs to bubble. This goes to the strong load picker in that bank of the IQ. The Mbox always stashes away the queue entry number of loads on issue in the event of a bubble - this saves the Qbox the time and logic needed to do an INum-to-queue-entry lookup.

Note that two loads may be issued concurrently in the same bank - one in the strong load pipe and one in the weak load pipe - and may both miss and need to bubble. But bubbled loads always reissue on the strong load pickers only. So while the WILL_RETRY values are always asserted at a fixed time relative to issue, the bubble requests themselves may be queued up and are therefore accompanied by a "retry valid" bit.

In the case of long latency Fbox ops, bubble requests are routed to the picker that originated the operation. When the instruction first issues, all "operand ready" matches against its INum are discarded. (The dependent instructions knew that they were dependent on a long latency op - they ignored the non-bubble requested broadcast of their parent's INum.) When the bubble request is honored, the INum is rebroadcast and all dependents then become data ready.

Note that the Fbox does not generate its own bubbles; counters in the PIL keep track of when floating-point divide and square root operations are due to complete and send the requests to the appropriate picker. The PIL is also responsible for telling the Fbox when a floating-point instruction has been killed off as the result of a disruption, since the Fbox does not monitor the Retire/Kill Bus. Should something untoward happen to a long-latency Fbox op (such as a division by zero), the PIL will find out only after the retire-time exception trickles through the disruption logic and appears as a kill on the RK Bus.

5.2.13 Oldest CBR Selector (OCS)

5.2.13.1 Design Considerations

Mispredicted conditional branches need to be resolved as quickly as possible to avoid a major performance penalty. When a branch misprediction does occur, the Ibox needs to know the INum of the culprit in the shortest possible time so that it can re-fetch it and start the replay process. If more than one branch mispredicts, we want to handle the oldest one first, since any younger instructions in the thread are on a bad path and are wasting processor resources. This is especially important for multiple mispredicting branches in the same thread - younger branches are rendered moot by the older branches' misprediction, so handling the oldest branch first minimizes the chances of replaying bad path code.

The problem is that quickly sorting out which branch instruction caused the mispredict — and if there is more than one mispredicting branch, which one is oldest — requires an exacting amount of logic. Certain cases, such as multiple mispredicting branches in one cycle, are also very uncommon. This puts the necessity of building the vast infrastructure required to find the correct answer immediately into question.

5.2.13.2 Design Architecture

What if we were to always assume initially — in keeping with the speculative philosphy of 21464 — that the oldest conditonal branch to issue in a given cycle is also the oldest mispredicting branch? It turns out that most of the time, according to benchmark simulations, this is a very astute guess. So this is what the OCS does: it identifies the

oldest conditional branch instruction to issue in a given tic and forwards its INum, TPU, Pipe ID, and Predicted Taken/Not Taken Bit to the Ibox. The Ibox uses this data to query the altenate PC table in preparation for a replay. If it turns out that no branches mispredicted, the Ibox ignores this information.

If we discover upon Ebox branch resolution that some other branch (not the oldest) mispredicted, we undo the mispredict trap, and add the INum(s) of the actual culprit(s) to the trap pool. Between the Pipe ID and the misprediction signals from the Ebox, the Ibox infers that the INum we gave them was for the wrong branch without any need for a special signal from the Qbox (unlike what was orginally thought). Finally, if the branch we selected did mispredict but was the victim of a load miss, the instruction needs to be replayed after the data comes back from main memory. In these two latter cases, the Ibox fakes a line mispredict, falling back on pre-existing error-handling mechanisms, which keeps it from causing any further mischief. For more details on the branch resolution process, take a look at the document Branches, and How To Resolve Them.

The OCS maintains internal state which records which of the instructions in the IQ are CBRs, and the Predicted Taken/Not Taken bits for each one, information it obtains from the Pbox instruction decoder. INums are obtained from the Exception Kill Logic since the OCS does not store INums.

Note that the implementation of this widget is in many respects similar to that of a picker. However, since it does not acutally influence the issuing of instructions from the IQ — and also operates one cycle behind the pickers — calling it a picker (as was originally proposed) would be somewhat misleading.

Note also that this mechanism only applies to integer conditional branch instructions. Floating-point CBRs go into the standard trap pool and are processed like generic disruptions.

5.2.14 Queue Chunk Allocator/Deallocator (ALC)

5.2.14.1 Design Considerations

The instruction queue is divided up into two banks, west and east. Each bank is further divided into sixteen chunks of four entries each. When a map block arrives at the IQ it is written into one chunk in each half of the IQ. We need to allocate these chunks efficiently, since if we run out of entries in the IQ, we need to stall at the Pbox Post-Map Skid Buffer (PSB) and signal back to the Ibox to stop sending map blocks. Needless to say, we don't want to do this very often.

Along with allocation comes the complementary problem of deallocation. Our queue isn't big enough to hold all in-flight instructions, so we need to re-use chunks for new instructions, evicting old ones from the IQ as soon as they are able to leave. It turns out that our performance is very sensitive to how long chunks stay in the queue, so this is a significant issue.

5.2.14.2 Design Architecture

The Qbox does not cause IQ full traps. Ever. Not ever. We don't do that sort of thing; in fact, we don't even support it. Instead, we do the next worst thing, we stall.

A queue chunk is free only if all of its instruction are eligible to be re-allocated. This is an important point; because of our chunk granularity, 16 "problem" instructions could conceivably fill one bank and thus the entire IQ. The exact conditions under which a given instruction may be deallocated are a function of instruction type and poison status, and are described in more detail in the Load/Poison Re-arm Widget (LPR) description. The LPR tells the ALC when a given instruction may be deallocated. In the simplest, most common case, valid single-cycle instructions may be deallocated as soon as they are known to not be the victims of a missed load.

The queue chunk allocator keeps track of which chunks are free. It then selects one chunk from each of the two queue banks to be written on the next cycle. If one or both banks are completely full, then the allocator signals an IQ full condition to the PSB and to the Map Thread Chooser in the INum Allocator. The Thread Chooser tells the Ibox that no threads can accept instructions until the IQ full condition is resolved. This is timed in such a way that the first freshly fetched and mapped blocks will arrive in the IQ immediately after the last buffered block is passed along from the PSB.

5.2.15 in-Flight Table (IFx)

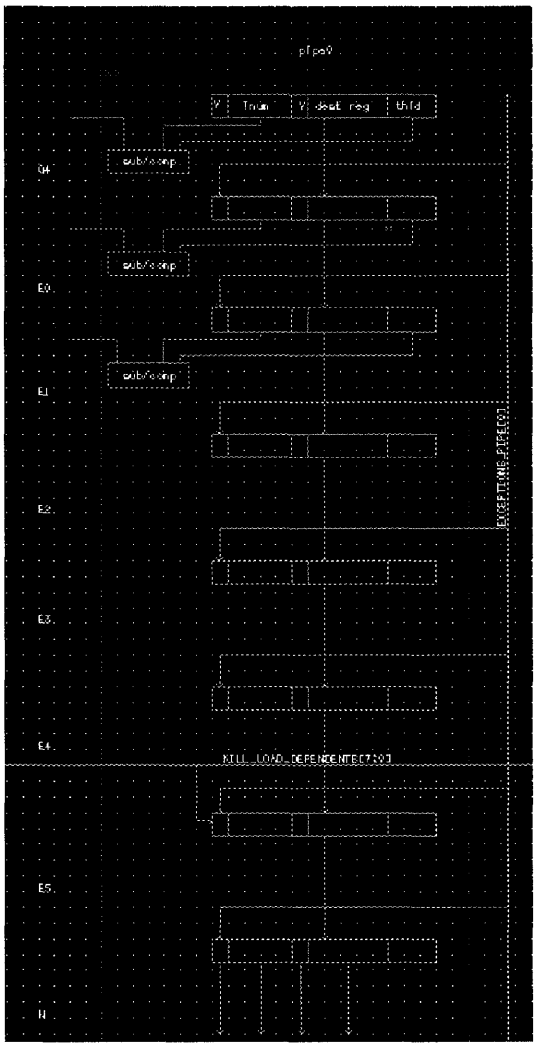
5.2.15.1 Design Considerations

The issue queue does not keep information about an instruction that has issued. Several cycles after an instruction issues, its entry in the issue queue is marked as free and can be reused. This reduces the occupancy of the queue and reduces IQ full stalls. However, certain information - such as the issued instruction's INum and destination register - are needed for later stages in the instruction's life (completion and writeback). The In-Flight Table is responsible for maintaining this state. Most importantly, the In-Flight Table checks - for every issued but not completed instruction — whether the instruction has raised an exception. If an instruction has raised an exception, its INum is marked accordingly in the In-Flight Table, and this information follows it to the Completion Unit. This is done to prevent the completion unit from thinking that the excepting instruction can retire.

5.2.15.2 Design Architecture

Basically, the In-Flight Table is a bunch of registers which mirror the 21464 pipeline from issue to writeback. Logically, the In-Flight Table sits between the IQ and the Completion Unit. Every cycle, the (up to) 8 issued INums enter 8 different staging pipelines — one per picker — in the In-Flight Table. Each cycle, the 8 staging pipelines "shift down."

The staging pipe for pipeline 0 is shown below.



5.2.16 Completion Unit (CMP)

5.2.16.1 Design Considerations

The 21464, like other out-of-order processors, issues instructions out-of-order but commits (retires) them to architectural state in program order. Here, architectural state refers to software-visible state: registers, memory, and IPRs. The completion unit is responsible for this reordering. Logically, the completion unit determines which INum is the "next to retire" for each TPU. This INum is driven to the Pbox RKU, where it is eventually driven out on the RK bus. It is important to retire instructions as early as possible, because many critical resources (INums, physical registers, load/store queue entries) are freed at retire time.

5.2.16.2 Design Architecture

The CMP maintains a vector ranging from INum 0 to INum 255. The state associated with each entry (INum) is as follows:

State	Description
C bit	INum is past the point at which it itself may raise an exception
IO bit	INum has been tagged by the Mbox as an IO load instruction
RTE bit	INum has a retire time exception associated with it
K bit	INum has been killed
Etype<5:0>	Retire time exception code

The following sections describe completion, killing, retirement, and Mbox processing in turn.

5.2.16.2.1 Completion

Instructions are issued from the instruction queue out of program order. They enter the In-Flight Table, and are checked against all possible exceptions. If the instruction makes it through the In-Flight Table without getting an exception, it sets the C bit for its INum. If the instruction gets hit with a retire-time exception while in the In-Flight Table, it sets the RTE bit in the completion unit and stores the exception type in the corresponding Etype field. If the instruction gets hit with an execution-time exception, such as a branch mispredict, the INum is removed from the In-Flight Table, and the CMP state is unaltered.

5.2.16.2.2 Kills

The 21464 indicates kills by posting a kill INum on the Retire/Kill bus and asserting the Kill signal. The completion unit receives this kill INum and simply marks all instructions younger than the kill INum (and in the same TPU) as complete, by setting the C bit. Also, for all killed instruction, we set the K bit to indicate that this instruction was completed due to a kill.

When an instruction associated with an RTE is the "next-to-retire" instruction, the CMP first sends its INum and TPU ID to the Pbox Retire/Kill Unit (RKU) as a "next-to-retire INum". The CMP then passes the INum, TPU ID, and RTE type information to the RKU again as a kill. Please consult Driving the Retire/Kill Bus (RK Bus) for more detailed information.

5.2.16.2.3 Retirement

To retire an instruction *Y* in TPU *X*, instruction *Y* must not cause an exception, and all older instructions for TPU *X* must also not raise an exception. Said another way, *Y* must be complete and all older instructions must also be complete. The CMP works by finding the oldest uncompleted instruction A. INum A is then driven out of the CMP to the Pbox as the next-to-retire INum. This indicates that all INums older than A can retire. The CMP then waits for INum A to complete. Until it does, TPU *X* has nothing to retire, since we must retire in order. Finally, when A completes, we search for the oldest uncompleted instruction in block B. If all instructions in block B are complete, the CMP indicates that the entire block B can retire, and we now advance to the next INum block for TPU *X* and search for the first uncompleted instruction.

The 21464 retires from only 1 TPU per cycle. Therefore, we need an arbitration mechanism to decide which TPU to retire from in a given cycle. We use a retire chooser, which simply chooses round-robin among those TPU that actually have something new ready to retire.

The 21464 uses a shared bus for driving retires and kills. Therefore, it is possible that in a given cycle, both a retire and a kill request access to the shared RK bus. The policy for resolving this conflict is that kills always take precedence over retires. Therefore, the CMP supports a stall mechanism to maintain state in the presence of simultaneous kills and retires.

5.2.16.2.4 Mbox Interface

To facilitate the processing of certain memory operations, there are special hooks from the CMP to the Mbox. The Mbox can signal that the next-to-retire INum is an I/O load operation. The CMP sets the corresponding IO bit in the completion vector, and advances the retire pointer. This allows merging of I/O operations, an important performance enhancement. In a similar vein, the Mbox can force the completion of the next-to-retire INum. This is used to complete memory barrier instructions.

Load/store ordering presents another interesting problem for the Mbox and CMP. A load instruction may complete normally, but still cause an exception. This exception is triggered when an older store issues and writes data to the same address as the load. To handle this, the Mbox has a "stall retire and zap" interface to the CMP. The stall retire interface freezes the CMP's retire pointer at the current instruction. It can not advance. The zap interface in essence "uncompletes" an instruction. The C bit of the violating instruction is forced to 0, thereby ensuring that the retire pointer can not advance past the trapping instruction.

5.2.17 Payload Array (PAY)

5.2.17.1 Design Considerations

From the Qbox core's point of view, the only significant features of an instruction are its dependencies, its latency, and anything else that might complicate scheduling, like being part of a DTB writer block. Certain other generic instruction attributes, namely INums and TPU IDs, are stored in the Exception Kill Logic, and the physical register numbers have structures devoted to them. But what about all of the other minor details that make up an instruction, such as its opcode? Where do they go?

5.2.17.2 Design Architecture

The Payload Array (PAY) contains all of the information about an instruction in which the Qbox has no direct interest, including opcode and function/displacement fields, and other flags and attributes derived by the Pbox (e.g. LSNums, the ISJUMP bits which flag indirect jumps) or passed through from the Ibox (e.g. the MAP_PAL_MODE bit which flags PALcode blocks). All of this type of information associated with a given instruction pops out of the PAY upon issue and is forwarded to the executing box. The PAY sits physically in the so-called "late" portion of the Qbox, off to the side of the IQ core with its sensitive timing paths.

5.2.18 Exception Kill Logic (EKC)

5.2.18.1 Design Considerations

When an exception occurs, there may be instructions on that code path which have been allocated space in the instruction queue. These instructions must be removed from the queue, in the interests of correct program execution, and the conservation of scarce queue space and other processor resources.

5.2.18.2 Design Architecture

The Exception Kill Logic eliminates from the queue any younger instructions with the same TPU as the excepting instruction. It also has the incidental function of storing the INums and TPU IDs of every instruction in the queue and passing them along to the Ebox and Mbox at issue time.

6
Integer Execution Unit — the Ebox

The 21464 microprocessor is organized into several major processing sections called boxes. The Ibox handles instruction fetching and program flow prediction, the Qbox schedules, often out of order the instructions fetched by the Ibox and the Ebox executes most of the non-floating point Alpha instructions scheduled by the Qbox. The Ebox contains multiple copies of its various processing elements so the Qbox can schedule as many instructions per cycle as possible. The upper limit is eight instructions issued simultaneously.

Structurally, the Ebox processing elements are organized into twelve functional units, but not all units are alike. In an attempt to keep the functional units small, fast and tightly coupled to each other, each unit executes a predefined subset of the instruction set. For example, of the eight integer-units, only two can execute store instructions, and only two units can handle any multimedia instruction. See the instruction breakdown table for a complete list.

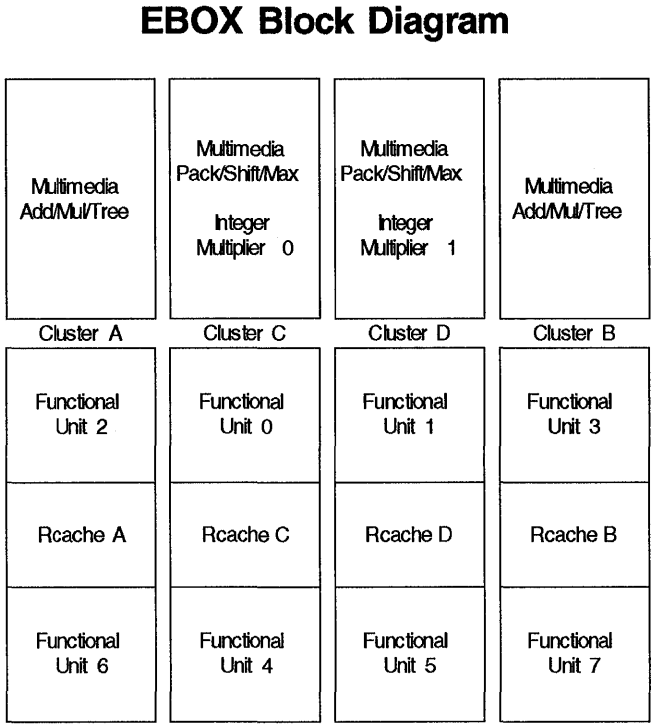
6.1 Major Components

The Major Ebox components are:

Table 6–1 Ebox Major Component Summary

Component	Description
Integer Units (8)	The integer functional units execute the traditional integer arithmetic and logical instructions as well as performing the address generation and data formatting of memory instructions.
Multimedia Units (4)	The multimedia units execute the newer integer instructions targeted at accelerating multimedia operations and also perform integer multiplication.
Register Caches (4)	The register caches store recently written register values allowing dependent instructions to issue before the register file is updated.

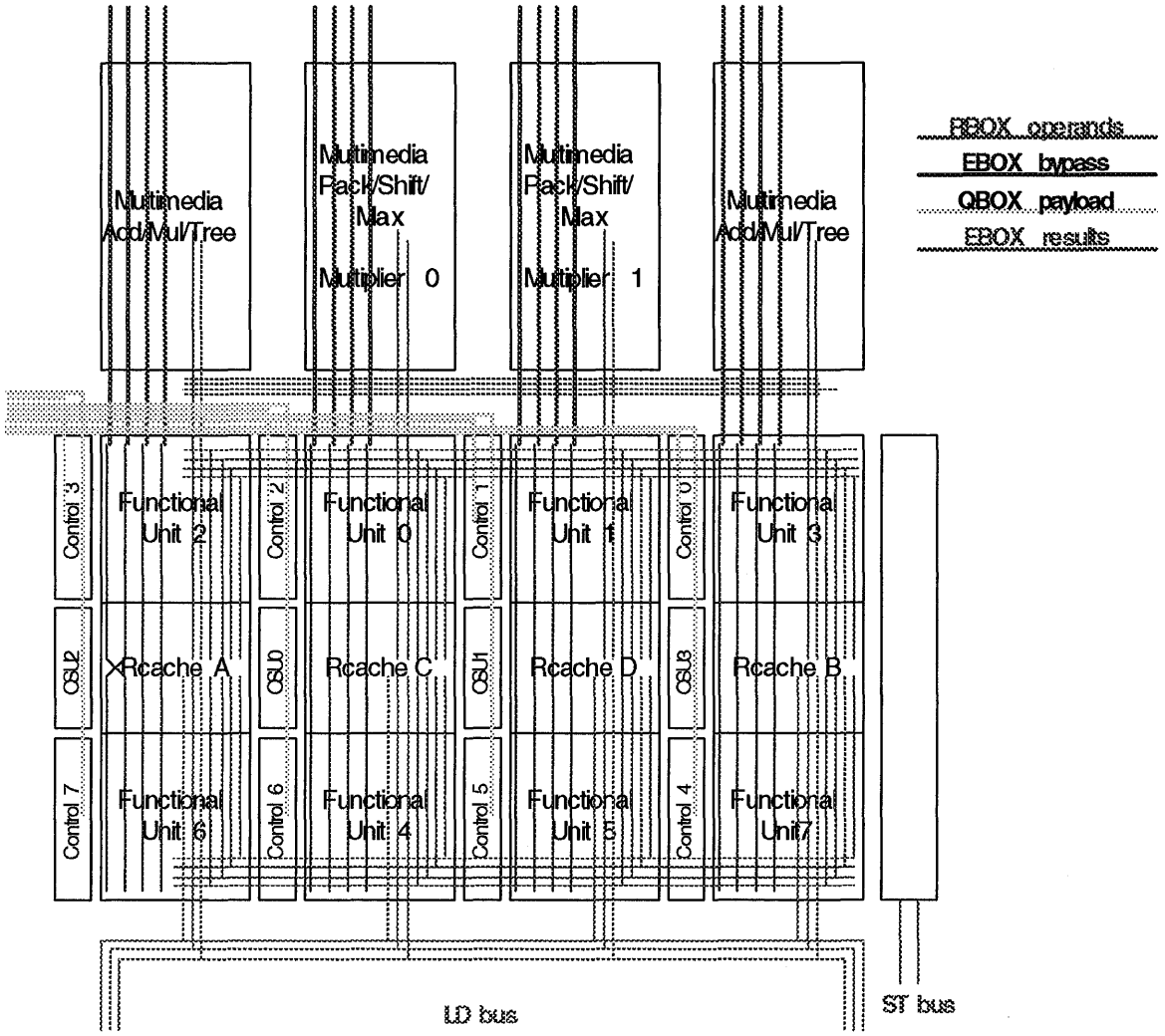
Figure 6–1 Ebox Block Diagram



6.1.1 Datapath

The key to understanding the architecture of the Ebox is an understanding of how operands and results flow through the Ebox. Shown below are the major datapaths and a rough representation of their layout. The rest of this document will mostly discuss the details of the elements within the Ebox and how they interact with these datapaths or the rest of the 21464. Detailed descriptions of the high-speed circuits used to implement the adders, shifters or multipliers can be found in the implementation documents.

Figure 6–2 Ebox Datapath Block Diagram



The Ebox bypass busses supply the Ra and Rb input operands to each functional unit. Values are driven onto the bypass busses from the register file, the register caches or directly from the functional units based on controls from the operand steering unit. The Ebox generates three types of results; the adders, shifters and logic units produce results in a single cycle, load instructions need three cycles to produce a result and multimedia instructions take five cycles to produce a result. To be available as operands to future instructions, all results are distributed throughout the Ebox and written into the register caches.

6.1.2 Timing

Cycle mnemonics are used throughout this document to identify the relative timing of signals. Table 6–2 identifies the cycle relationships assumed by this document.

Table 6–2 Interbox Timing Relationships

Qbox	Q1	Q2	Q3	Q4	Q5	Q6	Q7								
Reg. File				R0	R1	R2	R3						Rw	R1	R2
Ebox								E0	E1	E2	E3	E4	E5		
Fbox								F0	F1	F2	F3	F4	F5		
Mbox									M0	M1	M2				

Each cycle is further subdivided into two phases, the first half of a cycle is the 'A' phase, the second half of a cycle is the 'B' phase. A timing specification E0A refers to the first phase of cycle E0.

All timing references in this spec refer to the latch that launches the data, when significant transit time may be involved, that time or an expected arrival time will be separately stated.

6.2 Integer Clusters

An integer functional unit is a logical collection of processing elements that collectively execute a specific set of Alpha instructions. The 21464 has eight integer functional units organized as four clusters of two units each. The cluster grouping is significant because single-cycle results from previously executed instructions can be utilized immediately only by elements within the same cluster, but require a cycle propagation delay before they are available for use as operands to instructions executing in other clusters. The eight units are not identical but contain a predefined mix of processing elements. To ease implementation, the four clusters will be implemented as identical copies. Functions not needed in a cluster will be left unconnected. Table 6–3 identifies the major sections within an integer cluster and which functional units contain copies of those elements.

Table 6–3 Integer Cluster Sections

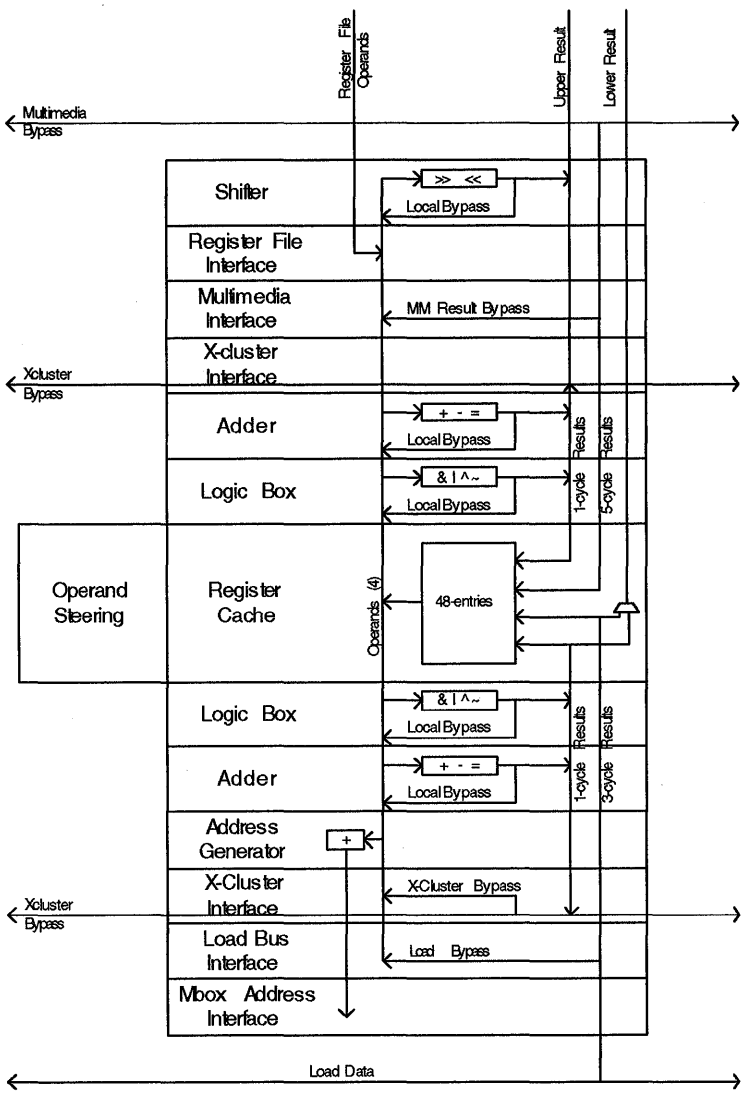
Section Name	Mnemonic	In Units	Description
Adder	Ep_ADx	0-7	A full 64-bit signed integer adder that produces a complete result each cycle.
Shifter	Ep_SHx	0-3	A full 64-bit shifter that produces a complete result each cycle.
Logic Box	Ep_LGx	0-7	Performs logical and arithmetic operations
Register File Operand Interface	Ep_RFx	0-7	Interfaces the operands from the register file to the Ebox opbusses. Also bypasses literals onto the opbusses.
Virtual Address Generator	Ep_VAx	4-7	Computes the 16-bit displacement add and factors the big/little endian control to form a correct virtual memory address.
Load Data Interface	Ep_LDx	4-7	Interfaces the data returned from the Mbox to the functional units and register caches.

Table 6–3 Integer Cluster Sections

Section Name	Mnemonic	In Units	Description
Multimedia Operand Interface	Ep_MOx	4-7	Forwards the instruction operands from the corresponding integer functional unit to the multimedia clusters. Each multimedia cluster is associated with the lower integer functional unit in a cluster and derives its operands from that functional unit.
Register File Result Pipe	Ep_RPx	0-3	Handles staging of different result latencies, floating load format conversion and forwarding of results to the register file.
Cross Cluster Result Interface	Ep_XCx	0-7	Receives one-cycle results from the other functional units, bypass the data onto the operand busses if needed immediately and latches the data for writing into the local register cache.
Global Control	Ep_GCx	0-7	Decodes the instruction information sent by the Qbox and coordinates the various processing elements within a functional unit.
Store Data Interface	Ep_STI	—	Interfaces to the Store Data buses to the Mbox. This unit is not actually part of the integer clusters but resides in a separate partition to the right of the integer clusters

Figure 6–3 shows how the sections are organized within a cluster. Elements that consume operands are generally positioned together in either the upper or lower part of the cluster. This also maps to physical position with the exception of the register file and multimedia interfaces. The operand busses span the full length of the cluster to allow elements in one half of the cluster to bypass results directly to any other unit in the cluster.

Figure 6–3 Cluster Section Organization



6.2.1 Adder

The adder unit completes a full 64-bit signed add/subtract operation in a single cycle. The inputs to the adder are adjusted swapped, complemented sign-extended, or shifted as necessary to allow the core adder to handle the various combinations of add and subtract operations. The instructions serviced by the adder are:

Table 6–4 Instructions Serviced by the Ebox Addr Unit

Type	Instructions
Add	ADDL, ADDL/V, ADDQ, ADDQ/V, S4ADDL, S8ADDL, S4ADDQ, S8ADDQ

Table 6–4 Instructions Serviced by the Ebox Addr Unit

Type	Instructions
Sub	SUBL, SUBL/V, SUBQ, SUBQ/V, S4SUBL, S8SUBL, S4SUBQ, S8SUBQ
Compare	CMPBGE, CMPULT, CMPEQ, CMPULE, CMPLT, CMPLE
Other	LDAH, LDA, RS, RC

Subtractions are handled by twos-complementing the Rb operand and setting the carry-in bit to the adder.

The S4 and S8 variants simply require the Ra operand to be shifted left by two or three bits.

For LDA and LDAH the register file interface has placed the displacement value onto the A operand bus. For RS and RC instructions, the register file interface placed the intr_flag passed by the Qbox in INST_INFO<0> onto the B operand bus. For all these instructions, the adder unit simply performs the equivalent of an ADDQ instruction.

The overflow trap signal is computed during E0A and passed to the EQ partition where it is latched and driven to the Qbox from an E1A latch.

The Adder section also allows direct bypassing of its result onto any or all of the four source-operand busses in the cluster. This allows dependent instructions to execute the following cycle. The operand steering unit detects the local bypass cases and drives the enable mask to the adder. If the Adder is active, it bypasses the result to all enabled busses.

6.2.2 Shifter

The shifter unit handles the arithmetic and logical shift instructions as well as the byte insert, extract, mask and zap instructions. All operations complete in a single cycle. The instructions serviced by the shifter are:

Table 6–5 Instructions Serviced by the Ebox Shifter Unit

Type	Instructions
Shift	SRL, SLL, SRA
Mask	MSKBL, MSKWL, MSKLL, MSKQL, MSKWH, MSKLH, MSKQH
Extract	EXTBL, EXTWL, EXTLL, EXTQL, EXTWH, EXTLH, EXTQH
Insert	INSBL, INSWL, INSLL, INSQL, INSWH, INSLH, INSQH
Zap	ZAP, ZAPNOT

The shifter receives the opcode and other instruction information from an EYA latch in the GCx section. The shifter decodes the opcode/function and if one of the above instructions is detected, controls and clocks are sent to the datapath to enable execution. When no match is detected, suppression of the clocks prevents any further action by the shifter.

When active, the shifter latches the operands at E0A and signals the opbus precharge logic. Results are computed in E0 and can be directly bypassed onto any or all of the four local operand busses in the cluster for use the next cycle. The results are also latched at E1A and driven onto a shared result bus to both the register cache and cross-cluster interfaces.

For big-endian threads, the shifter reverses the byte mask when computing MSKxxx, EXTxxx and INXxxx instructions.

The Shifter also forwards the operands to the multimedia cluster whenever an instruction handled by either the multimedia cluster or the store interface unit is issued.

Multimedia instructions require both Ra and Rb, store instructions only use Ra. The Shifter must not latch Rb for store instructions because the virtual address unit will be using Rb to compute the target VA.

Due to size and wiring constraints, only four instances of the shifter are implemented in the 21464. One in the upper pipe of each integer cluster.

6.2.3 Logic Box

The logic box handles the logical and conditional instructions producing all results including the conditional branch mispredict flag in a single phase. The instructions serviced by the logic box are:

Table 6–6 Instructions Serviced by the Ebox Logic Box Unit

Type	Instructions
Cmove	CMOVLBS, CMOVLBC, CMOVNE, CMOVLT, CMOVGE, CMOVLE, CMOVGT
Branch	BLBC, BEQ, BLT, BLE, BLBS, BNE, BGE, BGT
Logical	AND, BIC, BIS, ORNOT, XOR, EQV
Special	AMASK, IMPVER, SEXTB, SEXTW

For conditional branch instructions, the result is compared to the result predicted by the Ibox. Mispredicts are execution time traps which are reported directly to the Qbox and Pbox for corrective action. To minimize the penalty of a mispredicted branch, the Qbox has identified the oldest CBR issued this cycle and has prepared the Ibox for quick recovery. The logic box separately signals the Ibox if the CBR instruction mispredicted and was also the oldest executing this cycle.

6.2.4 Register File Operand Interface

The register file operand interface places operands supplied by the register file onto the Ebox operand busses. The register file supplies operands whenever the parent instructions results are not in bypasses or the register caches (ie. Parent issued more than eight cycles before the child).

In the case of integer operate instructions that use a literal field as the Rb operand, the operand will be marked as invalid by the Qbox and the OSU will default to enable the register file interface as the supplier. The register file interface detects integer operate instructions that use a literal and drives the literal value onto the bottom byte of the opbusses. The literal is zero-extended in the register file interface datapath.

For LDA and LDAH instructions, a 16-bit displacement is forced onto the Ra operand by the register file interface. To support these instructions, 16-bits are extracted from the instruction word and driven across the datapath. Additional multiplexing in the datapath places the 16-bit displacement on Ra<31:16> for LDAH or Ra<15:0> for LDA. The displacement is sign-extended in the register file interface datapath.

For RS and RC instructions, the flag passed by the Qbox is the instruction result. The flag bit is zero-extended onto the literal field and forced onto Rb. Since Ra must be invalid (ie. Forced to zero), almost any functional unit could drive the result. The current plan is to execute an RS or RC instruction as an ADDQ.

The other special case instructions are AMASK and IMPLVER. For these instructions a CPU specific constant is needed. The AMASK constant is driven onto Ra using the 16-bits needed by LDA/LDAH. The IMPLVER constant is driven onto Rb as a literal. For both instructions, the logic box performs the operation (Rc = Rb & !Ra) operation.

To correctly handle the propagation of poison even for Fbox instructions and to service FTOI and Fstore instructions that will receive their Fa operand from the register file, the register file interface will drive the opbusses for any active (TPU != 0) instruction whose operands did not hit in the OSU even if it is not handled by the Ebox.

6.2.5 Virtual Address Generator

The virtual address generator is a specialized adder that computes the virtual address for instructions that reference memory. Virtual address generation involves adding a signed displacement to Rb and adjusting the low bits to account for endian and alignment constraints. Although the main adder could have been extended to handled this function as was done in previous Alpha chips, feasibility studies showed that a combined adder with the additional input multiplexing and output control was too slow for the 21464.

Because the Mbox acts as the conduit for addresses passed to the Ibox. This unit also decodes JMP instructions and passes the target PC to the Ibox via the Mbox.

There are three basic equations:

va = Rb + SEXT(disp<15,0>)	LDx, STx
va = Rb + SEXT(disp<10,0>)	HW_LD, HW_ST
va = Rb	JMP, ECB, FETCHx, WH64, HW_MTPR, QUIESCE

The instructions serviced by the VAx section are:

Table 6–7 Instructions Serviced by the Ebox Virtual Address Generator Unit

Type	Instructions
Load	LDL, LDQ, LDQ_U, LDL_L, LDQ_L, LDBU, LDWU, LDG, LDS, LDT, LDF
Store	STL, STQ, STQ_U, STL_C, STQ_C, STB, STW, STG, STS, STT, STF
Jump	JMP, JSR, RET, JSR_COROUTINE
Special	TRAPB, EXCB, MB, WMB, ECB, FETCH, FETCH_M, WH64, HW_LD, HW_ST, HW_MTPR, LDx_ARM, QUIESCE

The virtual address generator is implemented identically in each integer cluster but not all of the above instructions can be issued to all clusters. The Mbox has a limit of three load instructions, two store instructions or a combined maximum of four per cycle. The Ibox can only accept one jump per cycle. Slotting restrictions in the Qbox will guarantee that the instructions issue to the correct pipelines.

Alignment and overflow/underflow errors are detected by the generator and reported to the Mbox. These errors are retire-time traps that the Mbox prioritizes with other traps before reporting to the Qbox.

The address generator receives the opcode and other instruction information from an EYA latch in the GCx section. The opcode/function is decoded and if one of the above instructions is detected, controls and clocks are sent to the datapath to enable execution. When no match is detected, suppression of the clocks prevents any further action.

When active, the Rb operand is latched at E0A and the opbus precharge logic is signaled. For non-store instructions, the address generator also activates the opbus precharge drivers for Ra. For store instructions, the SHx section captures the store data and handles Ra precharging and only Rb is latched and precharged by this section.

The address generator does not produce a result value that must be stored in the register caches or register file. The virtual address is sent directly to the Mbox from an early E0B latch and exceptions flags and poison status flags follow from and E1A latch. For load operations, the LDx section will eventually receive the load data and handle forwarding it to the register cache and register file.

The address and exception information is not passed directly to the Mbox from the VA1 section, but goes through the VA2 interface block at the bottom of the Ebox where the single-ended ADDR and the differential INDX busses are formed. For clusters EC and ED, the addresses are alternately driven onto the weak-load (P2) and store-only (P3) busses to the Mbox. To keep the ability to replicate the cluster, the VA units in the EA and EB clusters will also contain the logic to drive one of two busses to the Mbox, but connections will only be made to a single bus.

6.2.6 Load Data Interface

The instructions serviced by the load data interface unit are:

Table 6–8 Instructions Serviced by the Ebox Load Data Interface Unit

Type	Instructions
Load	LDL, LDQ, LDQ_U, LDL_L, LDQ_L, LDBU, LDWU, LDG, LDS, LDT, LDF
Special	HW_LD, STx_C

6.2.7 Multimedia Interface

The multimedia operand interface forwards instruction decode and payload information to both the MM cluster and store interface. Since the MM unit only handles instructions from opcodes 13 (MULx), 14 (I2F), and 1C (multimedia), and the operand interface needed to perform the opcode decode to correctly latch the operands; pre-decodes are forwarded to the MM unit instead of the opcode and TPU values. The function code (inst_info<6:0>) is forwarded so the MM unit can complete the specific instruction decoding.

To avoid the need to pass the opcode, TPU or inst_info fields over to the store interface block, the multimedia operand interface unit generates the specific control signals needed by the store interface to control latching, muxing an format conversion.

For floating store or FtoI instructions that source their operands from the register file, the Ebox provides the conduit to the store interface unit through the multimedia operand interface unit.

The instructions serviced by the multimedia unit or store interface unit are:

Table 6–9 Instructions Serviced by the Ebox Multimedia Interface Unit

Type	Instructions
Multiply	MULL, MULL/V, MULQ, MULQ/V, UMULH
Multimedia	Opcode 1C.*, except SEXTB, SEXTW
Store	STL, STQ, STQ_U, STL_C, STQ_C, STB, STW, STG, STS, STT,STF
Special	ITOFF, ITOFS, ITOFT, HW_ST

6.2.8 Global Control

The global control section decodes the instruction issued to the pipeline and detects valid single-cycle instructions as well as all illegal instructions. The OP_1CYCLE status flag is used by the cross-cluster interfaces and register caches to control distribution and updating of results. The illegal instruction decode is combined with overflow information produced by the adder and multiplier to produce the exception status vector sent to the Qbox.

6.2.9 Store Data Interface

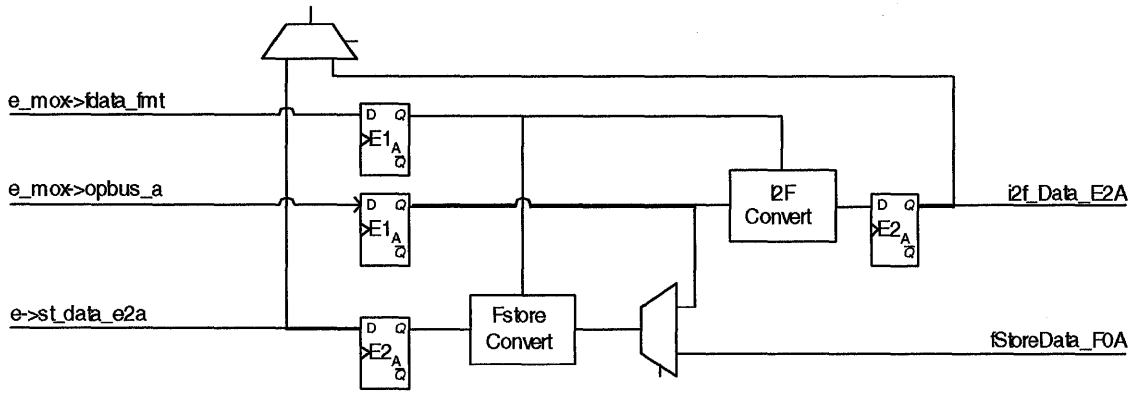
The instructions serviced by the Store Data Interface unit are:

Table 6–10 Instructions Serviced by the Ebox Store Data Interface Unit

Type	Instructions
Store	STL, STQ, STQ_U, STL_C, STQ_C, STB, STW, STG, STS, STT, STF
Special	ITOFS, ITOFF, ITOFT, FTOIS, FTOIT

Figure 6–4 shows the ITOFx and FTOIx instruction store data paths.

Figure 6–4 Ebox ITOFx and FTOIx Floating-Point Store Data Paths



I2F, F2I and Floating Store data paths

6.3 Operand Steering

The operand steering unit tracks the physical register numbers of all instructions that have issued in the past eight cycles and performs compares against the physical register numbers of the four source operands issued to each cluster each cycle. The destination physical register numbers are staged to match the result staging and the write pointer into the OSU CAM is identical to the write pointer used to write the register cache. This structure generates match lines that directly equate bypass enable signals to the interface units and register cache.

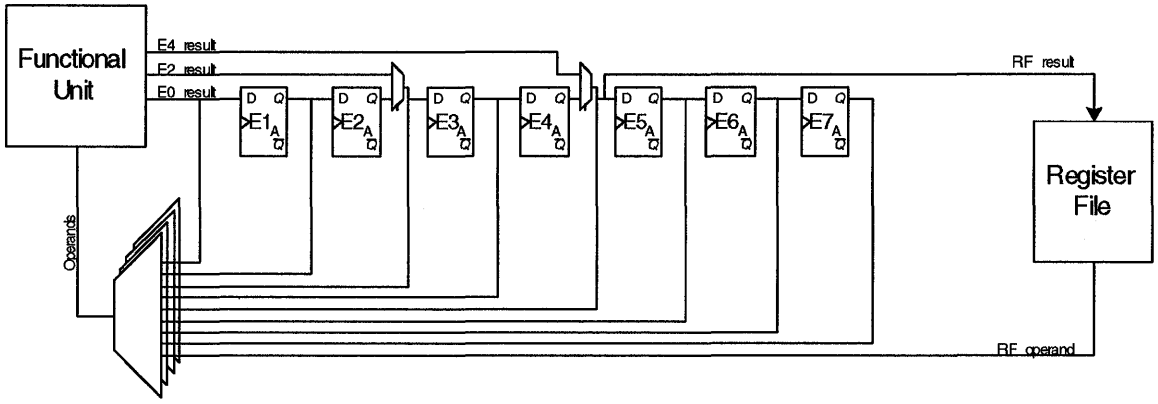
6.4 Register Caches

The register caches locally store copies of recently generated results allowing instructions which depend on these results to execute sooner than if the results needed to be written back to the main register file and subsequently re-read. Without the register caches, the parent to child issue delay on the 21464 would have been at least three cycles longer.

The register caches also equalize the issue to result latency and therefore eliminate the contention for register file write ports the varying E-box and F-box instruction latencies would have created.

Logically, the register cache can be thought of as a shift register. Results are entered based on their execution latency, shift out to the register file in E4, and finally out of the register cache in E7 after which the register file will source the value.

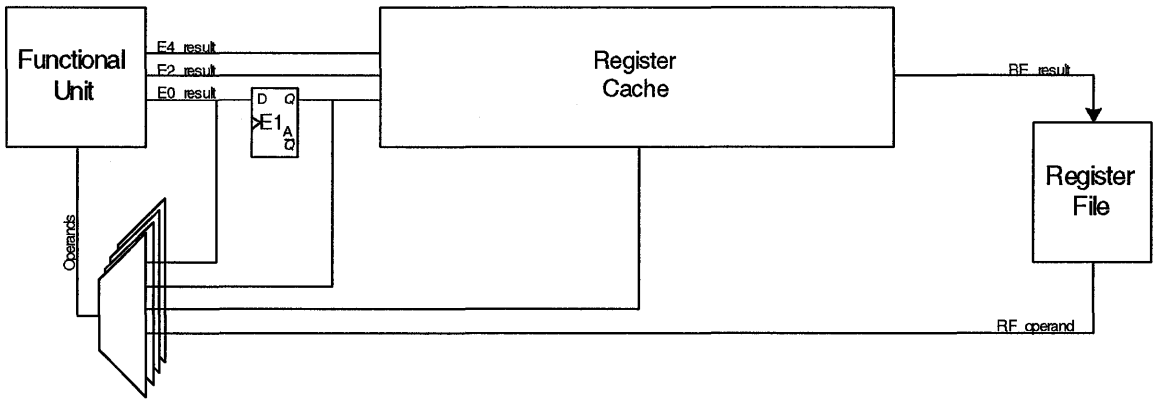
Figure 6–5 Ebox Register Cache Block Diagram



The logical representation above only shows how a single functional unit can access its own results, in reality the result multiplexing is much more complex and allows a functional unit to use any result produced by any other functional unit. Drawing a picture to represent that level of multiplexing is an exercise left for the reader.

Although easy to conceptualize, physically building a register cache out of latches as diagrammed above would waste both area and power. The Ebox register cache is built with multi-port static ram cells. Instead of moving the data through a fixed fifo, the ram version keeps the data in place and moves the read and write pointers.

Figure 6–6 Ebox Register Cache Multiport Static RAM Block Diagram



To provide the necessary locality to meet timing goals, each integer cluster contains a private copy of the register cache. The copies are identical, each containing the full set of available results from every instruction the Ebox executed in the past seven cycles.

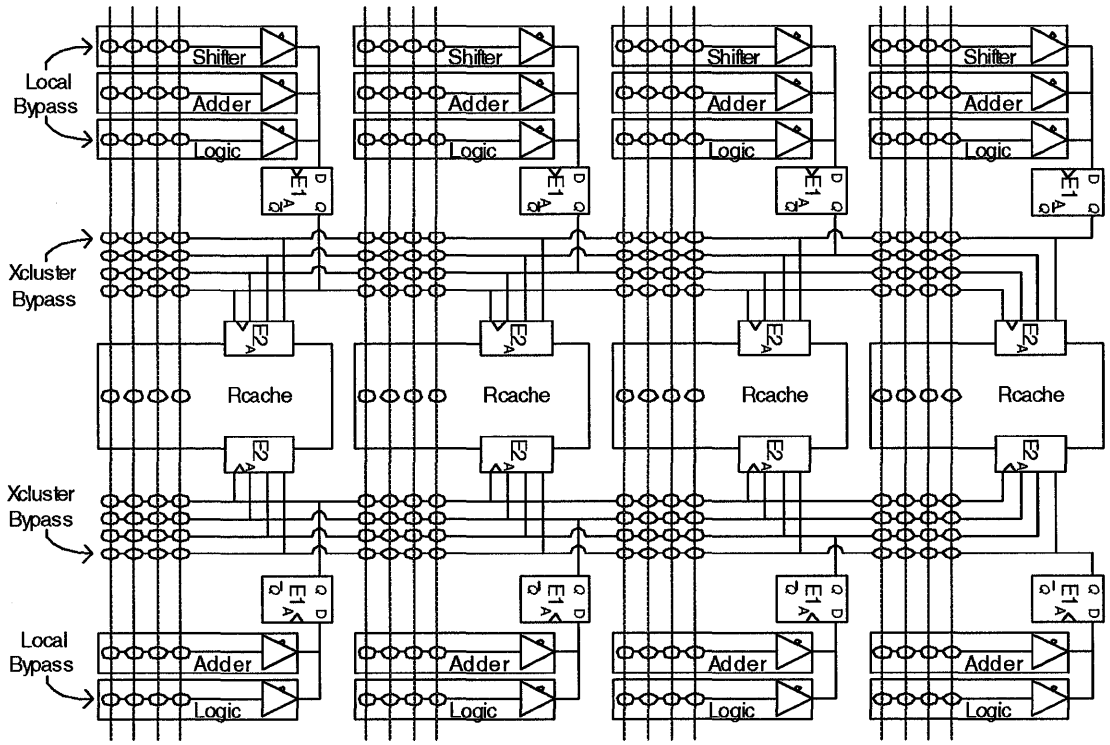
The term available is actually a key point. Single cycle instruction results are not stable until the late in the cycle. The results can be locally driven onto the opbus wires, but there is insufficient time to write the register cache or send the results to any of the other clusters. The E1 cycle is used to transport results to the other clusters, but the

transport delay also consumes much of a cycle leaving only enough time to bypass onto the remote opbusses. The register caches are actually written the second cycle after the results are produced. Table 6–11 and Figure 6–7 show the single-cycle result flow.

Table 6–11 Ebox Register Cache Single-Cycle Result Flow

	E0		E1		E2		E3	
	A	B	A	B	A	B	A	B
Ebox	Execute	Local Bypass	Transmit Xcluster	Xcluster Bypass	Write Rcache	Read Rcache		

Figure 6–7 Ebox Register Cache Single-Cycle Result Flow

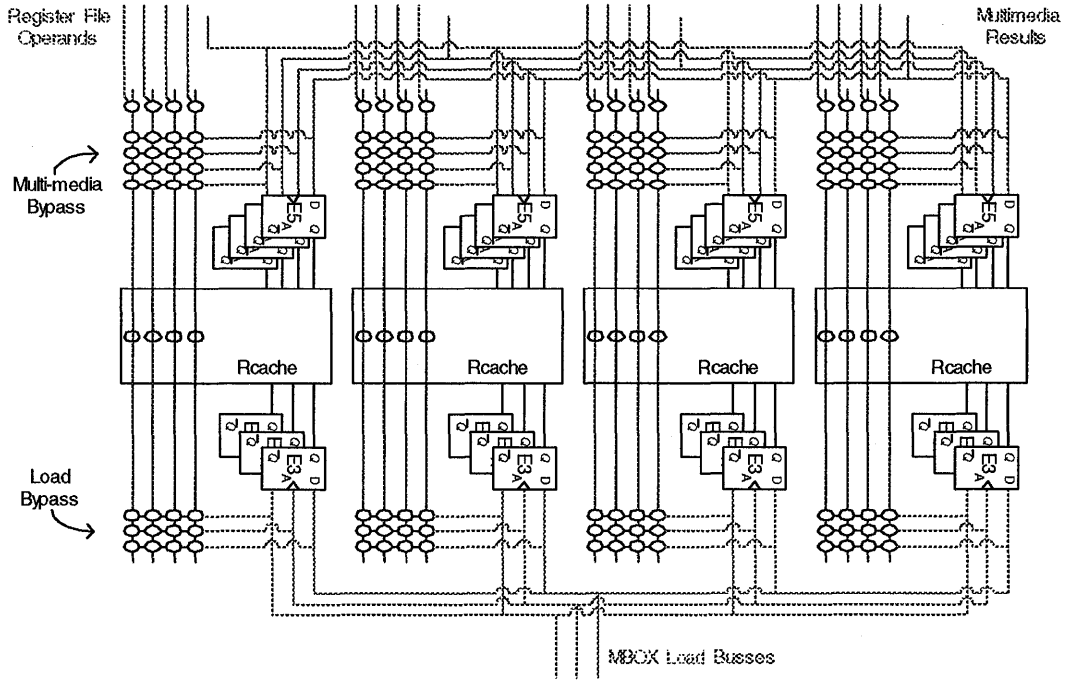


Multi-cycle instruction results are produced outside the integer clusters and broadcast to all clusters simultaneously. All multi-cycle instructions have either a three cycle latency (Loads) or a five cycle latency (multimedia, FtoI, Jumps, IPR reads). Each cluster independently bypasses these results if needed immediately then writes the register cache the following cycle.

Table 6–12 Ebox Register Cache Multi-Cycle Result Flow

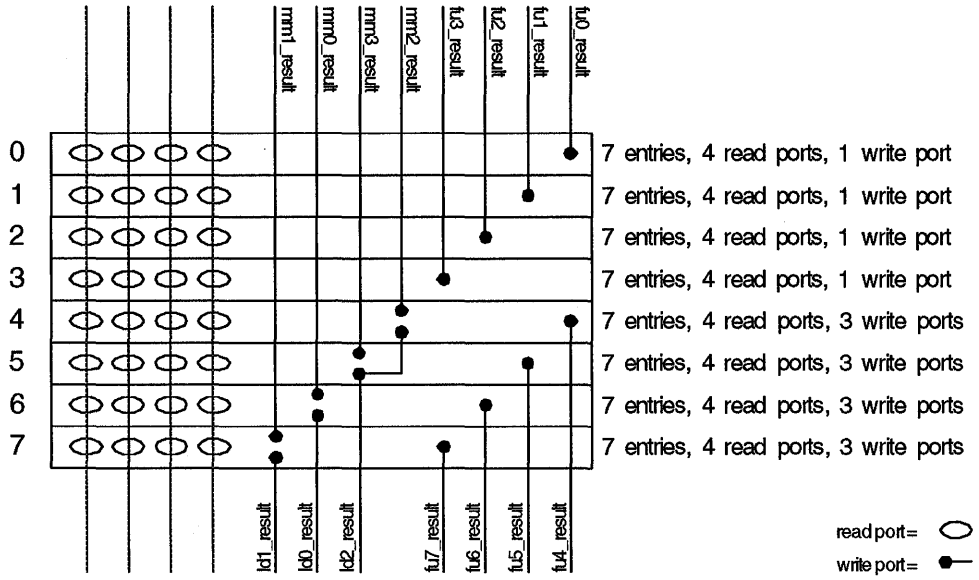
	E2		E3		E4		E5	
	A	B	A	B	A	B	A	B
Ebox 3cycle	...Finish Execute	Bypass Load	Write Rcache	Read Rcache				
Ebox 5cycle				... Finish m-media	Drive result	Bypass m-media	Write Rcache	Read Rcache

Figure 6–8 Ebox Register Cache Multi-Cycle Result Flow



Combining the single and multi-cycle cases shows each register cache receiving up to 15 results per cycle. Eight single cycle latency results from each functional unit, three three-cycle latency results from the memory load interfaces coupled to functional units 4, 5, 6 and 7, and four five cycle latency results from the multimedia clusters also associated with functional units 4, 5, 6 and 7.

Each register cache will source up to four operands, two to each of the functional units in the cluster. Because a result can be used as either or both inputs to any number of future instructions, every register cache entry can drive all four of the operand busses within the cluster.



6.4.1 Writing the Rcache

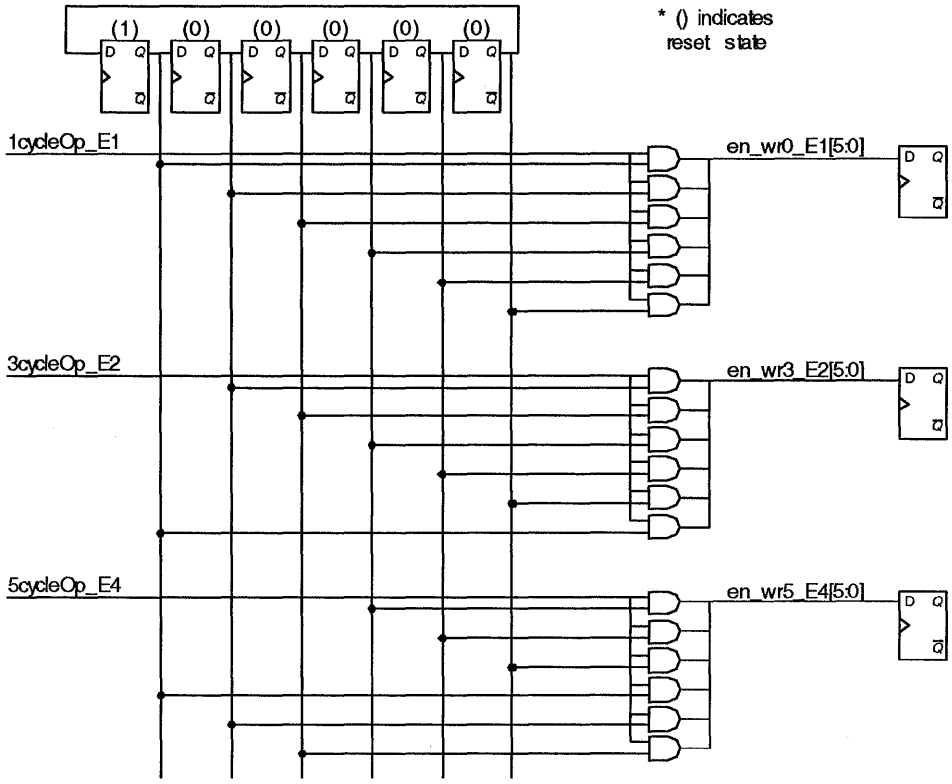
The fixed interval between instruction issue and register file update eliminates the need for any form of busy or free status to be associated with register cache entries. Entries are assigned in a round-robin fashion and are guaranteed to be free for reallocation on the next pass. The assignment sequence is a simple modulo "cache depth" counter implemented as a 1-bit, one-hot shift register.

The single bit allocation pointer is directly combined with the instruction latency information to produce the enable signals for each of the write ports.

```
en_wr0_E1[n] = 1cycle_OP_E1 && ptr[n];
en_wr1_E2[n] = 3cycle_OP_E2 && ptr[(n+1)%RCACHE_ENTRIES];
en_wr2_E4[n] = 5cycle_OP_E4 && ptr[(n+3)%RCACHE_ENTRIES];
```

Remember, single-cycle results are not written until E2, where three-cycle results are written in E3 and five-cycle results are written in E5, and because the allocation pointer is just a shift register, skewing the indices is equivalent to a time delay.

Figure 6–9 Writing Entries in the Ebox Register Cache



6.4.2 Reading the Rcache

Each register cache entry has four read ports, one to each opbus in the cluster. Read control information for each opbus is driven from the operand steering Unit. OSU CAM matches to upper pipe results can be used directly as read enable signals to the register cache. Matches to lower pipe results are more complex because of the ambiguity between a load result and a single-cycle operation result that occurred two cycles earlier. To resolve the ambiguity, the OSU also drives a set of bypass active signals for each of the four lower pipes. If a load or multi-media bypass is active, the register cache should not be read.

The cycle timing of the operand control information is shown below. The OSU performs the CAM operation in cycle EYB, the match lines are distributed as bypass enables in EZA and the register cache is read in EZB.

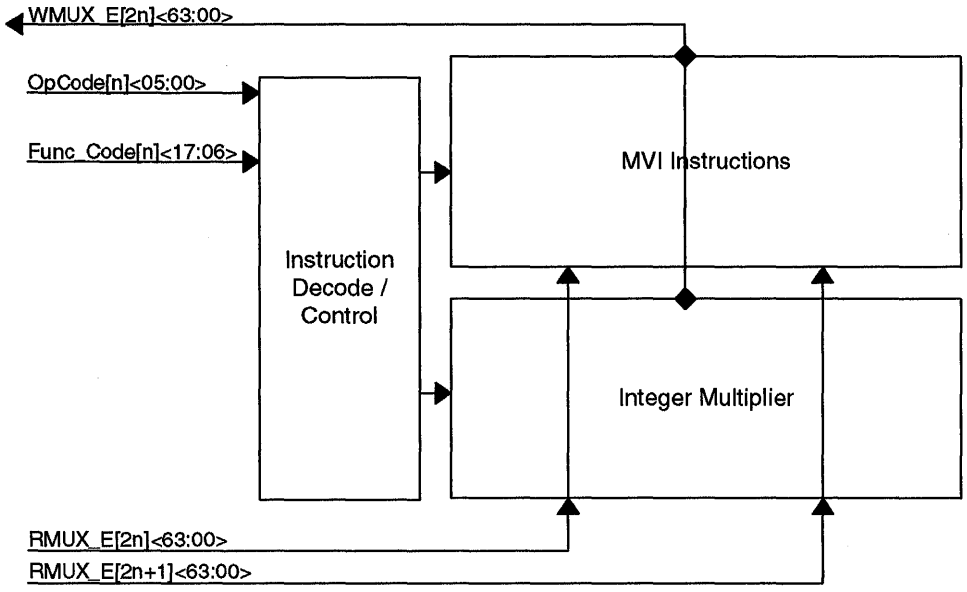
Table 6–13 Ebox Cycle Timing of Operand Control Information

	Q4 R0		Q5 R1		R2		R3 EZ		E0
	A	B	A	B	A	B	A	B	A
Qbox	Xmit Src Pointers		Xmit Dest Pointers						
Reg. File			Decode	Read	Mux	Transmit			
Ebox						OSU CAM	Drive enables	Bypass opbus	execute

6.5 Multimedia Unit

The Multimedia Unit consists of three major sections shown below. The Control Logic and occupies the left side of the unit. The computational logic is divided into two sections. The first section handles integer multiply instructions. The other section handles the MVI instructions.

Figure 6–10 Ebox Multimedia Unit Block Diagram



6.5.1 Inputs and Outputs

The opcodes arrive from below in a wiring channel from the integer execution units. The operands arrive from the bottom and are shared with the Ebox lower units. The result bus exits from the bottom of the box and goes to the register caches.

6.5.2 Signal Nomenclature

All signals belong to the E box and the Media partition. Signal names start with the EM prefix. The three section prefixes are CTL for the control section, MUL for the Multiplier section, and MVI for the MVI section. Thus, the three valid prefixes for multimedia unit signals are EM_CTL, EM_MUL, and EM_MVI.

6.5.3 Timing

Figure 6–11 Ebox Multimedia Unit Pipeline Timing

R2		E0		E1		E2		E3		E4		E5	
A	B	A	B	A	B	A	B	A	B	A	B	A	B
Transport & decode		Data xport	Execute pipeline							Drive result	X-cluster	Write rcache	Read rcache

The Op Code and Function Code are clocked in the E box on R3A. The operands and final control signals are latched on E0B. Execution begins on E0B. The longest operation completes by E4A. All instructions are delayed until E4A before being driven on the Result Bus.

6.5.4 Instruction Decode/Control Section

The Instruction Decoder looks at the OpCode and Function Code to determine the operation to be performed. From this information, it extracts the following fields:

- Arithmetic/Logic Function
- Byte/Word/Longword
- Signed/Unsigned

The OpCode and Function Codes are decoded into instruction names and latched on E0. Each signal is named Ep_CTL%"inst.name"_E0A_H. There are 8 opcode decodes for the Multiply section. There are 24 opcode decodes for the MVI section. The signal (Ep_CTL%quiece_E0A_H) is asserted if no instruction is recognized. A 2 bit code represents the Byte/Word/Longword state. For the normal IMUL opcodes (MULL, MULL/V, MULQ, MULQ/V, and UMULH, the data types are implicit in the opcode and are not included in the Byte/Word/Longword decoded state. The code is defined in the table below:

Value	State
00	Byte
01	Word
1x	Longword

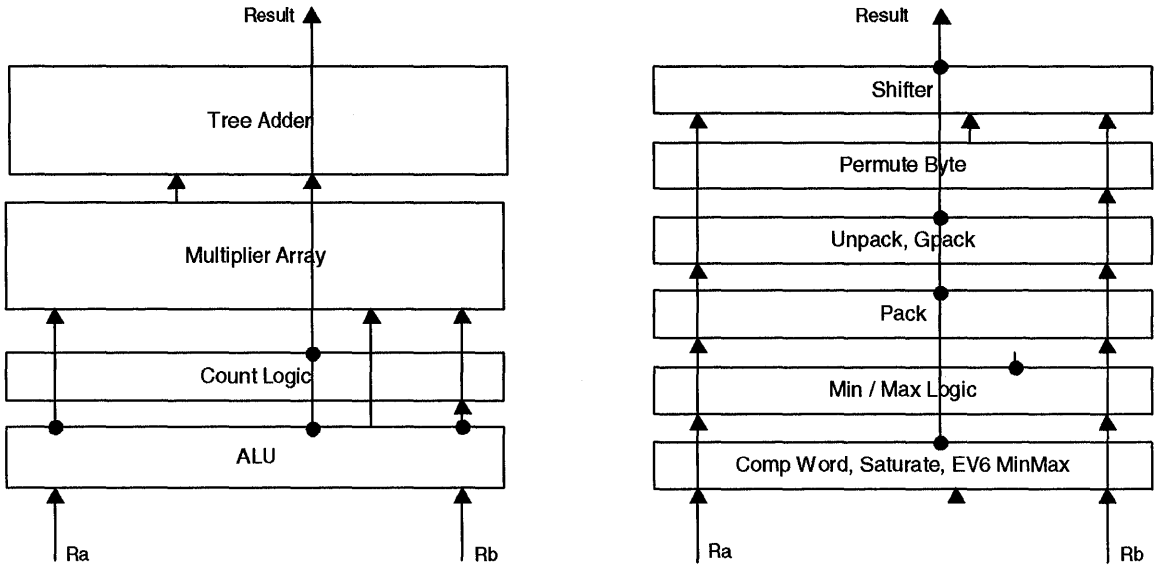
Signed/Unsigned is represented by a 2 bit code Ep_CTL%SGN_E0A_H<1:0>. This is defined in the table below:

Value	State
11	Signed
00	Unsigned
10	Signed * Unsigned (TMUL)

6.5.5 MVI Section

The MVI section accepts instructions on E0A and operands on E0B. It produces results on E4A. The block diagram of the MVI section is shown below:

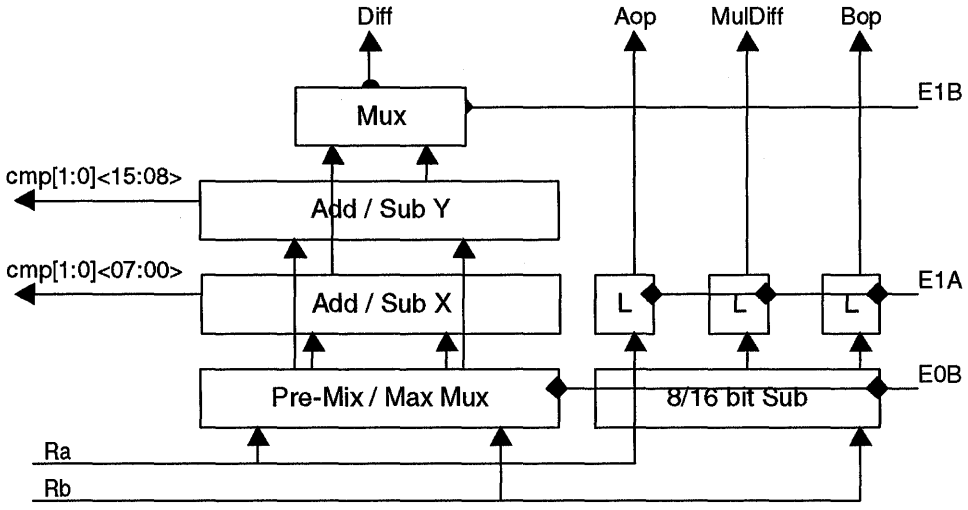
Figure 6–12 Ebox Multimedia Unit MVI Section Block Diagram



6.5.6 ALU

The ALU serves a number of instructions. It computes the magnitude of (Ra-Rb) for the TABSERR and the TSQERR instructions. It also performs the additions and subtractions for the TADD, TSUB, PADD, and PSUB instructions. In addition, it performs the first level of compares for the MINMAX instruction, the MIN instruction, and the MAX instruction. Finally, it performs the compares for the CMPWGE instruction. The block diagram is shown below:

Figure 6–13 Ebox Multimedia Unit Arithmetic Logic Unit



The Pre-MIN/MAX Mux shuffles the bytes appropriate bytes to the two adders for each instruction. It generates 4 busses; the a and b inputs to Add/Sub X and the a and b inputs to Add/Sub Y. The X adders in Ra and Rb to present the performs a+b or a-b on bytes, words, or longwords. . The Y adders perform b-a on bytes, words, or longwords. CMPLT and OVFLO for each byte from both adders is brought out for control.

6.5.6.1 TADD, TSUB PADD, PSUB, CMPWGE, MIN, MAX Instructions

The Add/Sub X block gets all the Ra inputs on it's a inputs and all the Rb inputs on its b inputs in normal byte order. The MUX passes the sum. Signed/Unsigned does not matter to ALU block. For byte operations, each byte grows to 9 bits, which is passed through the Mux to the Tree Adder (TADD) or the saturation logic (PADD). For Word operations, each word grows to 17 bits, which is passed on to the Tree Adder or Saturation Logic. The remaining instructions send the sign bits to the control logic.

6.5.6.2 TABSERR Instruction

Both Add/Sub blocks get the same data as the TADD, TSUB instructions. However, the X adder performs a-b and the Y adder performs b-a. The sign bits of the X adder are used to control the MUX. The MUX selects the positive result for each byte or word. This is passed on to the Tree adder.

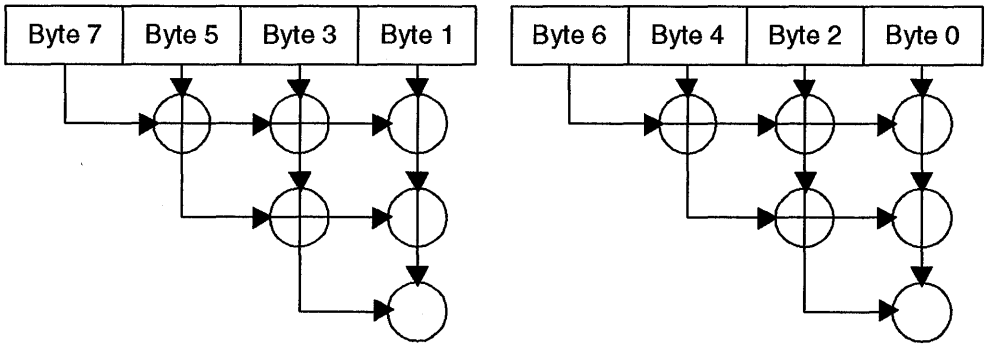
6.5.6.3 TSQERR Instruction

A separate 8 or 16 bit subtract computes the difference between A and B and passes the result to the multipliers. This is done in one phase so the multipliers can start one phase sooner than they could if the other ALU structure were used.

6.5.6.4 Min/Max Instruction

The min/max instruction uses 12 of the 16 adders to perform the first level of comparison for finding the min and max. This divides a register into 2 groups of 4 bytes and compares the bytes in each group as shown below:

Figure 6–14 Ebox Multimedia Unit Computation of the Min/Max Instruction



The byte reshuffling is defined in Table 6–14.

Table 6–14 Ebox Multimedia Unit Min/Max Instruction Byte Reshuffling

Bus	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
X a	—	—	Ra 0	Ra 1	Ra 0	Ra 2	Ra 1	Ra 0
X b	—	—	Ra 1	Ra 2	Ra 2	Ra 3	Ra 3	Ra 3
Y a	Ra 5	Ra 7	Ra 7	Ra 7	Ra 6	Ra 6	—	—
Y b	Ra 4	Ra 6	Ra 5	Ra 4	Ra 5	Ra 4	—	—

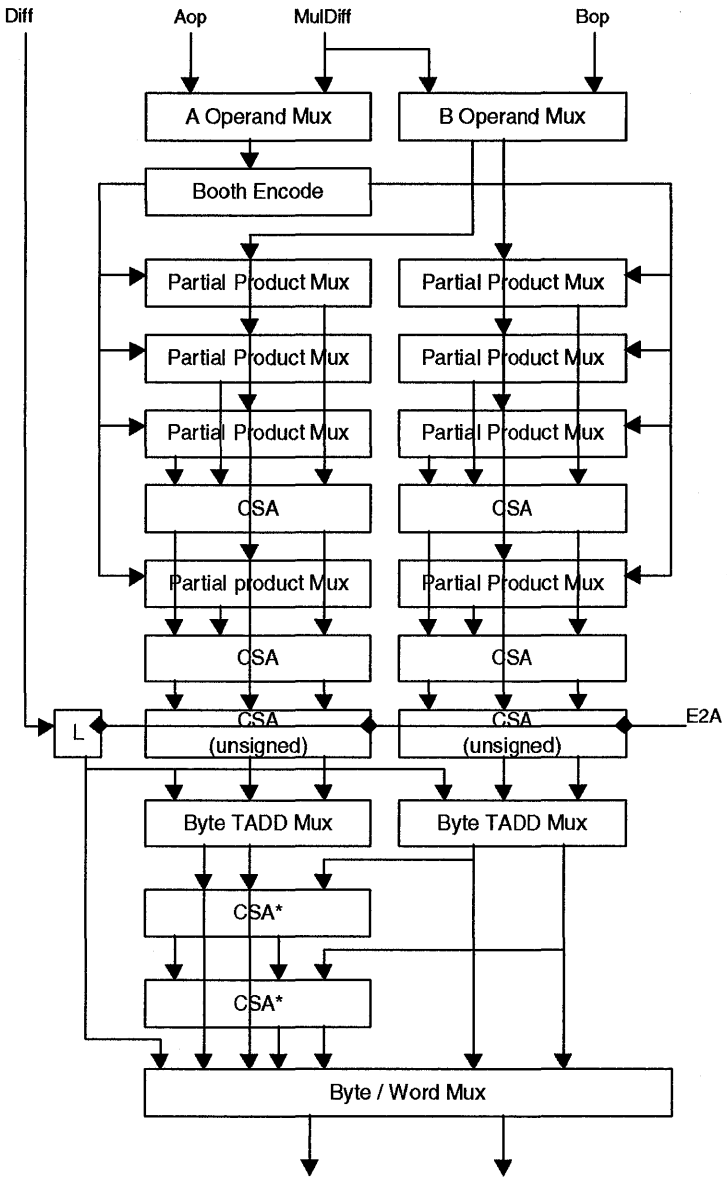
The CMPLT and OVFLO bits for each byte are sent to control logic which generates signals to control the remainder of the MINMAX logic further down the pipeline.

6.5.7 Multiplier Array

The multiplier is used for PMUL, TMUL, TSQERR instructions. It takes inputs from the ALU section and is configured as 8 8X8 multipliers or 4 16X16 multipliers. It can handle signed * signed, unsigned * unsigned, or signed * unsigned input operands. It selects inputs either from the A and B bus (PMUL, TMUL) or the multdiff output from the ALU (TSQERR). It passes either 4 32 bit results or 8 16 bit results on to the tree adder.

It is configured as 2 bit Booth coded stages followed by an array of carry save adders. The 16X16 /dual 8X8 multiplier structure is shown in the figure below. Four copies are required for the full multiplier box.

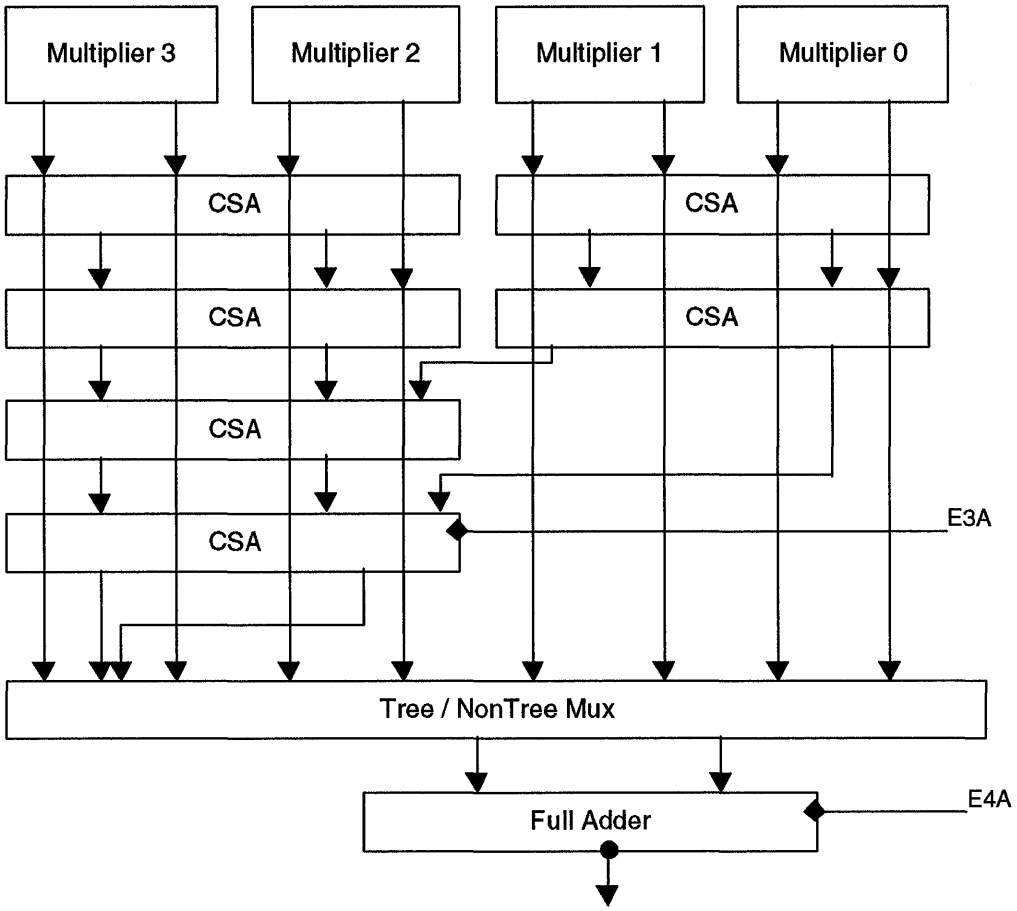
Figure 6–15 Ebox Multimedia Unit Multiplier Array Block Diagram



The two data paths show 8X16 multipliers. For word operations, the TADD Mux shifts the right data path 8 bits to the right before adding to the left data path. This is required to form a 16X16 multiply from two 8X16 multiplies.

For Byte operations, the Partial Product muxes sign extend each byte into bits<15:08>. The TADD Mux does not shift the right data path. The two CSA* blocks are used as the first stage of the Tree Adder for byte operations. (The tree adder combines 8->4, 4->2, 2->1 for bytes. For words, it only combines 4->2, 2->1 The Tree Add Muxes also bring in the data for byte Tree Operations that do not use the multiplier (i.e. Tree Add, Tree Sub, Tree ABS Val) so the CSA* blocks can be used as the first tree adder stage.

Figure 6–16 Ebox Multimedia Unit Multiplier Array Tree Adder



The results from the 4 multiplier blocks are combined in the tree adder as shown below:

The Tree/NonTree mux selects the output from the tree adder for all operations except PMULH and PMULL. For those operations, the high or low 16 bits of product from each multiplier are selected. The full adder combines the sums and carries from the carry save adder array to form the final result. It must be multiplexed with the other sources of final results before being sent to the Register Cache.

6.5.8 Count Logic

The Count Logic is used to support the CTPOP, CTLZ, and CTTZ instructions. CTPOP counts the number of bits that are "1" in Rb. CTLZ counts the number of leading zeros in Rb. CTTZ counts the number of trailing zeros in Rb. These were implemented in the 21264 by building logic to look at each bit pair and indicate whether 0 to 8 items to be counted are present. This information is then fed to the tree adder, which produces the final tally. The implementation in this section is done the same way.

6.5.9 Compare Word, Saturation, and the 21264 Min Max

From this point on, all logic blocks take inputs from the ALU and produce results that will eventually be multiplexed with the Tree Adder block. The results from the rest of the logic blocks must be delayed to line up with the Tree Adder output. The first step in this delay is to latch the inputs to this block and drive the latched data to the remaining logic blocks.

When the Compare Word instruction is executed, the ALU does 4 unsigned word subtracts and sends the sign and carry bits to the control logic. The Compare Word logic gets the control bits and forms 8 bits to be output in bits<7:0>.

Saturation is required for the PADD and PSUB operations. The ALU performs the appropriate add or sub for bytes/words, signed/unsigned and sends the compare bits to control logic. It sends the arithmetic result down the data bus allowing it to overflow. The Saturation logic either passes the result or forces the appropriate saturation result based on the control bits.

The 21264 Min and Max instructions select the minimum or maximum between A and B on a byte by byte or word by word basis. The appropriate add or subtract is performed in the ALU, the compare bits are sent to the control logic. The 21264 MIN MAX logic receives the A and B bus with control bits from the control logic. It multiplexes between the A and B inputs to select each Min or Max byte or word.

These three functions are implemented with a multiplexer controlled by bits derived from the ALU Sign and Carry bits.

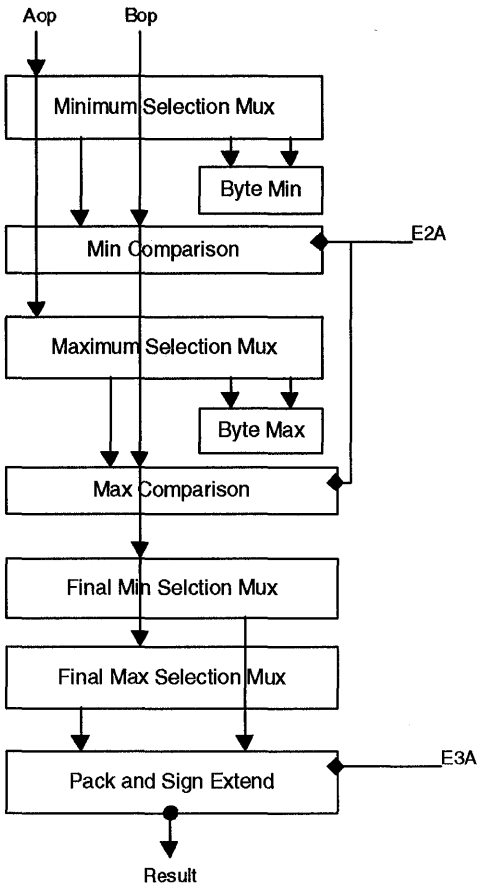
6.5.10 MinMax Logic

The MinMax logic performs the second stage of the new MINMAX instruction. The first stage was performed by the ALU, which generated 12 compare results for bytes, 6 compare results for words, and one compare result for longwords. The first step for the second stage is to take the compares generated by the first stage and assemble the minimums and maximums from the A bus inputs.

For bytes, the comparisons generated two sets of minimums and maximums; one for the first low 4 bytes and one for the high 4 bytes. For words and longwords, one minimum and maximum are selected.

Next, these must be compared with the previously found minimums and maximums in the B register. This is done with partial difference circuits. The actual difference is not needed, just the results of the comparison. This information is then used to control a second multiplexer which selects the minimum and maximum from the three (bytes) or two (words or longwords) candidates. The results are sign extended to longwords and sent to the result bus. The block diagram is shown below:

Figure 6–17 Ebox Multimedia Unit Min/Max Logic Block Diagram



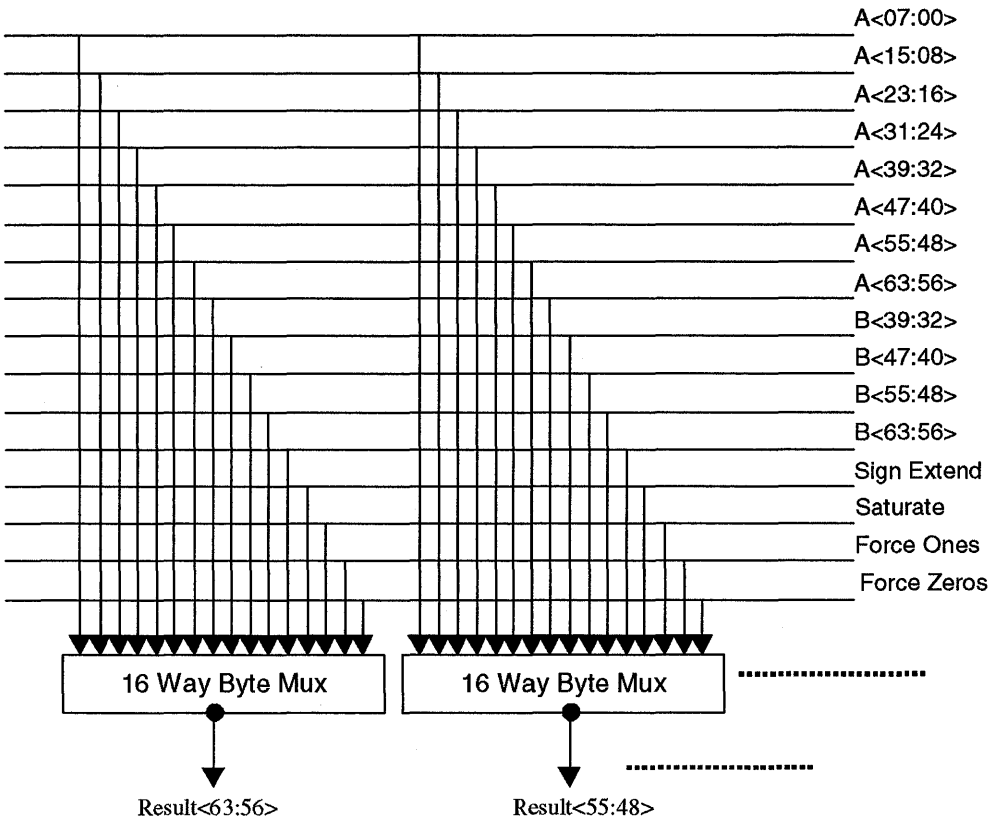
6.5.11 Pack, Unpack, Permute Byte

The Pack, Unpack, and Permute Byte logic generate control signals for the Shifter logic. Pack must detect when word->byte or longword->word overflow. The Shifter logic will be commanded to saturate. Unpack must sign extend the result for signed data types. The Permute Byte instruction decodes the B register inputs and commands the Shifter Logic to reorder bytes from the A input, force zeros, force 1's, sign extend from the result byte to the right, or select bytes from the high longword of the B register.

6.5.12 Shifter

The shifter has a horizontal bus that permits each byte to select any other byte from the A register, any of the 4 high bytes of the B register, or various littorals the support saturation, sign extension, force to ones, or force to zeros. The total bus structure and one byte slice are shown below:

Figure 6–18 Ebox Multimedia Unit Shifter



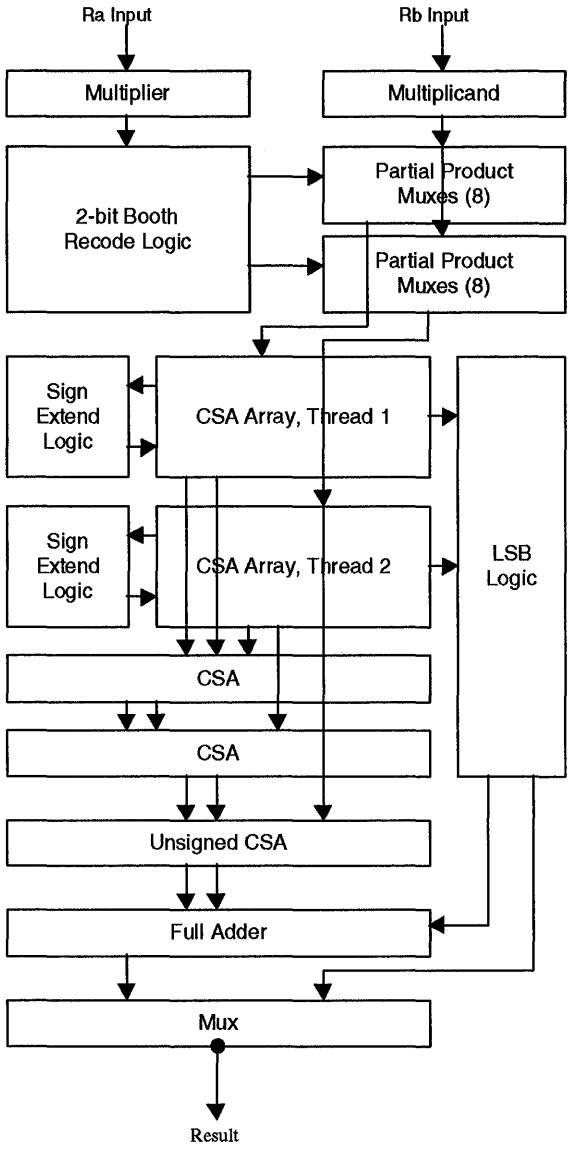
6.5.13 Delay

The Delay block aligns all the different instructions in time so that one result bus may be shared. There are 3 different busses carrying results with different timing. The result bus from the Tree adder is the longest latency bus. It is available to be latched on E4A and is sent to the result bus with no delay. The next longest delay is the result coming from the MINMAX logic which is available to be latched on E3A. It is delayed one clock and sent to the result bus. The third bus is available to be latched on E2A . It is delayed two clock cycles and sent to the result bus. No bypassing is performed during these delays.

6.5.14 Integer Multiplier

The integer multiplier handles the MULL, MULL/V, MULQ, MULQ/V, and UMULH instructions. The integer multiplier is implemented as a 2 bit Booth encoded signed multiply. It must support unsigned multiplies for the UMULH instruction. The block diagram is shown below:

Figure 6–19 Ebox Multimedia Unit Integer Multiplier



The Ra inputs are encoded into 32 2 bit Booth partial products. Each Booth encoder looks at its own 2 bits plus one bit to the right. The three bits are encoded as shown in the following table.

Code	Partial Product
000	x0
001	x1
010	x1
011	x2
100	(-1)x2
101	(-1)x1
110	(-1)x1
111	x0

Each partial product is created with a multiplexor the selects 0x, 1x, or 2x the multiplicand and inverted or uninverted outputs. The minus is formed by the inversion and a carry that is inserted in an open position in the Carry Save Adders.

The Carry Save Adder arrays are divided into two "threads". Multiplier<1:0> , <4:5>, <9:8>... partial products are summed in one thread. Multiplier <3:2>, <7:6>, <11:10>... are summed in the other. This significantly reduces the number of levels of gate propagation required for the final product. The two threads are summed together in two more ranks of CSAs. One more CSA is required to support unsigned multiplies. This also serves as the place to put the last carry in for the minus case of the highest order partial product.

The LSB logic begins the process of propagating the carry and producing the low order bits of the product while the higher order bits are working their way through the CSA array. The Full Adder combines the sums and carries from the Carry Save Adder array and from the LSB logic to produce the final product.

The multiplexor selects either the low order bits or the high order bits depending on the instructions that was decoded.

6.6 Debug Features

Debug features in the 21464 come in several flavors:

- CYA bits to disable performance features or select simpler algorithms.
- Error detection logic to halt trap to PAL or halt trace collection
- A trace bus to collect internal state.

There are currently no CYA bits defined for the Ebox.

The Ebox will be able to signal a trap based on a programmable decoder in the global control section. A value and bitmask for pipeline, tpu, opcode and function will be compared to each valid instruction issued and a signal will be sent to the global debug handler whenever a match is detected. The decoding will not be limited to Ebox instructions but will not be able to detect the NOP and MB instructions retired immediately by the Qbox.

For observability, the Ebox is considering allowing collection of the following signals onto the debug trace bus:

- Pipeline active flags which indicate the latency of the instruction issued to the pipe
- cbr_mispred and opx_poison status flags
- tpu, opcode, function bits for a specified pipe.

The objective is to incorporate all debug logic into the EQ partition and take advantage of the fact that most interesting control wires flow over the top of the EQ partition.

One of the most interesting problems is how to write the IPR bits necessary to control this logic. Some hack where the bits are actually stored in the Ibox or Mbox and captured in the Ebox on an IPR read operation might make the most sence.

6.7 Testability Features

The Ebox is considering a boundary SCAN based methodology for manufacturing fault detection.

Scan latches would be implemented in the EQ partition where virtually all control inputs from the Qbox enter the Ebox. Operation of all functional elements in the Ebox including the register caches can be achieved through this interface. To allow depositing and examination of results, the scan chains will be extended across the top of the Ebox through the latches that hold the operands and results flowing to and from the Register File. With this level of control and observability, the only major structures not covered would be the virtual address generation and load data interface blocks that interact with the Mbox.

The current belief is that the scan-based features are adequate to test the register caches and BiST engines will not be required in the Ebox.

6.8 External Interfaces: Ibox, Qbox, Pbox, Mbox, Register File, Fbox

6.8.1 Ibox

The Ebox needs to communicate instruction flow information with the Ibox. The instructions that control program flow are Branches and Jumps.

For conditional branches, the Ibox has already predicted an execution path and needs to be notified if it chose the wrong path. Since up to eight conditional branch instructions can be executed by the Ebox at once, the Ibox requires INUMs to identify which branches mispredicted and which predicted correctly. The Ebox does not have access to INUMs so it returns a set of branch mispredict flags to the Pbox. The Pbox then associates the flags with INUMs and notifies the Ibox of the oldest mispredicted branch. The Ebox drives the mispredict flags to the Pbox exception funnel from an E1A latch in the EQ partition.

The mispredict flags are not conditioned with poison so the Pbox must correctly handle branches that mispredict due to poisoned data.

To speed-up the branch mispredict path, the Qbox pre-determines the oldest issued conditional branch and guesses it will mispredict. The Ibox prefetches the PC of this branch and the Ebox sends a single bit to the Ibox indicating if that branch actually mispredicted. If it did, we are several cycles into recovery, if not, the Ibox must wait for the Pbox to figure out which CBR (if any) was the oldest mispredicted branch.

The target virtual address of a Jump instruction is sourced from register Rb. When executing a jump instruction, the Ebox forwards register Rb to the Ibox for comparison against the predicted target PC. Since the Qbox only schedules one Jump instruction per cycle and only into Functional Units 4 or 5, the multiplexed weak load address bus is used to transmit the target address to the Ibox. The Ibox was told the jump would issue in Q5 and drives the return PC in time to be returned to the Ebox over the IPR_RD data bus in cycle E3. Return PC's from jump or unconditional branch instructions flow through the Ebox multimedia result path.

The Ebox also uses these paths to execute the Ibox HW_MFPR and HW_MTPR Internal Processor Register (IPR) read and write instructions. IPR write data is sent on the weak-load address bus, IPR read data is returned with the same timing as a jump return PC through the IPR_RD data bus.

The only other signals the Ebox receives directly from the Ibox are the KERNEL_MODE and FP_ENABLE status vectors. If a thread attempts to execute a privileged CALL_PAL instruction when its KERNEL_MODE bit is not set, the Ebox will report an illegal instruction exception. If a thread attempts to execute any floating point instruction when the FP_ENABLE bit is not set, the Ebox will report an illegal instruction exception. These signals lack a timing specifier because the pipeline must be flushed before the bits can change. Because of the flush, the signal is stable for many cycles before the next instruction reaches the Ebox.

6.8.2 Qbox

Instructions are passed to the Ebox from the Qbox. With each valid instruction, the Qbox sends most of the original instruction longword, source and dest operand pointers and some control information. Exception information is the only information the Ebox returns to the Qbox.

Instruction information like opcode and function code is sent to the Ebox through the payload array in the Qbox. The opcode bits are transmitted in tact, but the rest of the original instruction longword is packed based the instruction format. For CALL_PAL instructions, the upper 11 bits of the 26-bit function code are ORed together and packed into the 16-bit info field.

Table 6–15 Instruction Information From the Qbox to the Ebox

Format	Field	Instruction Bits	INST_INFO
Memory	Displacement / Function	15:0	15:0
Branch/Jump	None	None	None
Operate	Literal & Function	20:5	15:0
Floating	Function	15:5	10:0
RS/RC	Function + Intr_flag	15:1, intr_flag	15:1, 0
CALL_PAL	PALcode Function	OR(25:15), 14:0	15, 14:0
MFPR/MTPR	Index & Class	Pal, 24:21, 3:0, 11:5	15, 14:11, 10:7, 6:0

The data is read out of the Qbox payload in cycle Q4B, transmitted to the Ebox from a Q5A latch and received by the Ebox in an EYA latch for decoding.

Since the operand data for an instruction can be found on result busses, in the register cache or in the main register file, the Ebox needs to determine where the source data is located. The operand steering unit in the Ebox keeps track of instruction results and generates the control signals necessary to drive the correct source operand select lines. The Ebox compares the instruction source operand pointers to the destination pointers from the recently issued instructions. A match is a bypass or register cache read, a miss is a register file access.

The thread processor unit is needed by the Ebox to select the correct IPR bits. The Mbox provides a per-TPU copy of the B_ENDIAN IPR bit to the Ebox for use when computing byte shifts or when generating memory addresses. The Ibox drives a per-TPU copy of the KERNEL_MODE and FP_ENABLE context bits for use decoding illegal instructions. The thread processor unit is a one-hot structure indicating the thread associated with this instruction. If no thread ID bits are set, the pipe is defined to be inactive.

The PAL_MODE bit is used to report illegal (due to insufficient privilege) instruction errors only, it does not effect Ebox processing of instructions. See the exception handling section for more information on Ebox exception processing.

6.8.3 Pbox

The Pbox handles prioritization and notification of conditional branch mispredict information to the Ibox. After issuing a set of instructions, the Qbox speculates the oldest issued branch will mispredict. True or False, the Pbox must still scan the mispredict vector for any other branches that missed. The conditional branch mispredict signals sent to the Pbox are not conditioned with poison. The Pbox exception logic must ignore all exceptions resulting from poisoned operands as neither the Ebox or Fbox factor poison into any exception reporting.

6.8.4 Mbox

The Ebox interface to the Mbox is primarily used to resolve instructions that reference or manipulate the memory system.

The load path is a super tight timing path. The Ebox needs to computes the virtual address and send it to the Mbox. The Mbox then accesses the Dcache and returns the data to the Ebox all within three cycles. The goal is for the Ebox to compute the address early in cycle E0A and begin transmitting it to the Mbox. The assumption is that transmission delay will account for most of E0B and that the Mbox will latch the address and begin processing in E1A. The Ebox intends to create adders specially tuned for the 16-bit displacement-add.

The 21464 architecture limits the parallelism to no more than three load type instructions, two store type instructions and a maximum of four memory instructions total per cycle. The load data arrives in the Ebox late in cycle E2B and the Ebox will drive store data early in E2A expecting the data will be available in the Mbox by the end of cycle E2.

For STx_C instructions, the lock flag must be returned to the Ebox and stored in the destination register. The Mbox will bubble STx_C instructions and drive this flag in cycle E2B relative to the bubble.

IPR writes will be performed through the LD/ST interfaces to the Mbox. For Ibox HW_MTRP instructions the Ebox will send the data (Rb) as the address on the weak-load address port in cycle E0B and the Mbox will forward the data along to the Ibox. Mbox HW_MTPR instructions will issue to the strong-load pipes and can issue up to two per cycle. All HW_MFPR reads issue to the weak-load pipes, one per cycle. Both the Ibox and Mbox return data on the IPR_RD bus in cycle E3A.

The Ebox drives the pipe 0 (P0) signals from partition EA, the P1 signals from partition EB and the P2 and P3 signals alternately from partitions EC and ED. The P2 and P3 signals are therefore outputs of the EY partition indicating there were multiple driving partitions in the Ebox.

6.8.5 Register File

The Register File sources operands that are not currently in the register cache. Each instruction issued can take up to two operands and eight instructions can be issued at once for a total of 16 operands per cycle.

Because of the way CMOV instructions are split into two instructions, each operand is actually 66 bits instead of the expected 64 bits; one extra bit is used to store the intermediate result for CMOV instructions. In addition to the CMOV condition bit, poison status is stored in the register file with the data. Poison is only sent to and received from the Ebox.

Although the Ebox has more than eight functional units and instructions can complete in different amounts of time, the register caches equalize the instruction latencies so no more than eight results will ever be generated to the Register File in any given cycle.

6.8.6 Fbox

The Ebox and Fbox directly exchange data relating to floating store, ItoF, and FtoI operations. For floating store operations, the Fbox sources the store data whenever the operand is resident in the Fbox register caches. Since the Ebox owns the final multiplexing of store data to the Mbox, it is responsible for forwarding floating store data located in the register file through the same datapaths used to send integer store data. This eliminates the need to route the store pipe operand from the Register File to the Fbox.

FtoI operations work just like floating stores to the Fbox. Instead of sending the data to the Mbox, the Ebox pushes the value back through the multi-media result busses into the Ebox register cache. FtoI format conversion is handled by the same logic that converts floating store data sent to memory.

ItoF data is format converted by the Ebox and sent to the Fbox register caches. The data is also pushed back through the multi-media result busses, into the Ebox register caches and eventually to the register file. This is necessary since the Fbox does not have a result path back to the main register file for these functional units.

6.8.7 Global

The intention is to clock the Ebox primarily off GCLK+2. Minimal thought has been put into reset requirements and there have minimal discussions about test requirements for the Ebox.

6.9 IPRs

The Ebox needs access to three IPR fields. The B_ENDIAN bit of the Mbox VA_CTL IPR and the KERNEL_MODE and FP_ENABLE fields of the Ibox Process Context IPR.

The B_ENDIAN bit is used by the Ebox in computing the virtual address for load and store instructions as well controlling the byte extract, insert and mask instructions. The Mbox will supply a per-TPU vector shadowing the committed state.

The KERNEL_MODE field is used to detect threads with insufficient privilege to execute a CALL_PAL instruction and flag these instructions as illegal. The Ibox will decode the current process context IPR and drive a per-TPU structure that shadows the committed state.

The FP_ENABLE field is used to force a trap whenever a floating-point instruction is executed. Software uses this bit during process context switches to detect the need to save or restore the floating-point registers.

6.10 Exceptions

The Ebox reports instruction status back to the Qbox for each executing pipeline. Prioritization, reporting and any other exception based actions are left to the Qbox. In general the Ebox does not stall or take any special action in the presence of an exception event.

There are several types of exceptions reported by the Ebox:

Table 6–16 Exceptions Reported by the Ebox

Exception	Description
EQ%ADD_OVERFLOW_E1A_H<7:0>	Integer add/subtract operation overflowed
Ep%MUL_OVERFLOW_E4A_H	Integer multiply operation overflowed
EQ%ILLEGAL_INST_E1A_H<7:0>	Illegal opcode or function code issued
Ep%Px_BAD_VA_ALIGN_E1A_H	Address Alignment error
Ep%Px_LD_PAR_ERROR_E4A_H	A parity error was detected on a memory load from the Dcache.
EQ%CBR_MISPREDICT_E1A_H<7:0>	Branch prediction was incorrect
EQ%OP{A,B}_POISON_E1A_H<7:0>	The operand to this instruction was poisoned
EQ%DRAINT_INST_E1A_H<5:4> EQ%MTFPCR_INST_E1A_H<5:4>	An IFETCHB or a non-PAL-mode MT_FPCR instruction was issued to the pipe.

The Ebox also decodes each instruction and detects the cases where exception status is either known or guaranteed to be available early and the instruction can be retired early. If late status can occur, like with MULL/V or many Fbox instructions, the Qbox must delay retirement. Early retirement frees-up resources in the Qbox allowing more instructions to enter the queue earlier.

There are two classes of illegal instructions, reserved opcode/function combinations and insufficient privilege. The Ebox decodes the following cases as reserved opcode exceptions:

Table 6–17 Ebox Reserved Opcode Exceptions

Opcode	Function
00	00 - 3F and not in Kernel Mode 40 - 7F > BF
01 - 06	All
07	Codes not defined by SIMD FP extension All when FP_ENABLE is not set.
14	Codes not defined in SRM V7.0 Codes 14.xx8 through 14.xxF when FP_ENABLE is not set.
15	All when FP_ENABLE is not set.
16	Code<5:4> = 012 OR Code<8:5> = 11012 All when FP_ENABLE is not set.
17	All when FP_ENABLE is not set.
1C	Codes not defined in SRM V7.0 or the 21464 MVI extensions
19,1B,1D,1E,1F	All when not in PAL mode
20-27	All when FP_ENABLE is not set.

The Fbox exception information although more complex, is also driven in cycle E4 (F4). Multiplexing E and F box exceptions is a task left to the P/Qbox.

6.11 Poisoned Data

Poisoned is the term given to a value that is the product of a load-miss.

With each load operation, the Mbox returns a status bit indicating if the address hit in the cache or a queue. Data returned for operations that do not 'hit' is garbage and care must be taken to ensure that this data does not alter program state. The method used is to tag each data word with a poison bit. Poison is contagious so any future product of poisoned data is also poisoned. This includes instruction results, CBR mispredict signals, load addresses, store data, target PCs of jump instructions, etc.

Eventually, all instructions issued in the shadow of a load miss will be replayed and the results of those instructions will be overwritten in the register file. Poisoning store data and load addresses protects bad data from entering the memory system and factoring poison into jump addresses or CBR mispredict signals prevents the predictors in the Ibox from training against false data.

To the Ebox, maintaining poison state simply involves ORing the poison status bits from the instruction inputs. The process is complicated slightly because the poison status bit is returned from the Mbox later than the load data, but it is still early enough to catch-up to any inflight instruction.

6.12 Format Conversions

Traditionally the only type of data formatting the Ebox handled with was sign or zero extension of data loaded from memory. In the 21464, the Mbox performs the sign/zero extensions, but the Fbox does not have a path to the register file for data loaded from memory, so the Ebox must handle conversion of floating-point load data. The recently added FTOI and ITOF instructions also define data format conversions. These instructions allow for direct movement of data between Integer and Floating-point registers.

Table 6–18 Ebox/Fbox/Mbox Data Conversion Matrix

SRC	DST	Description
Mbox	Ebox	Integer Loads
Mbox	Fbox	Floating Load
Ebox	Mbox	Integer Store
Fbox	Mbox	Floating Store
Ebox	Fbox	ITOF Instruction
Fbox	Ebox	FTOI Instruction

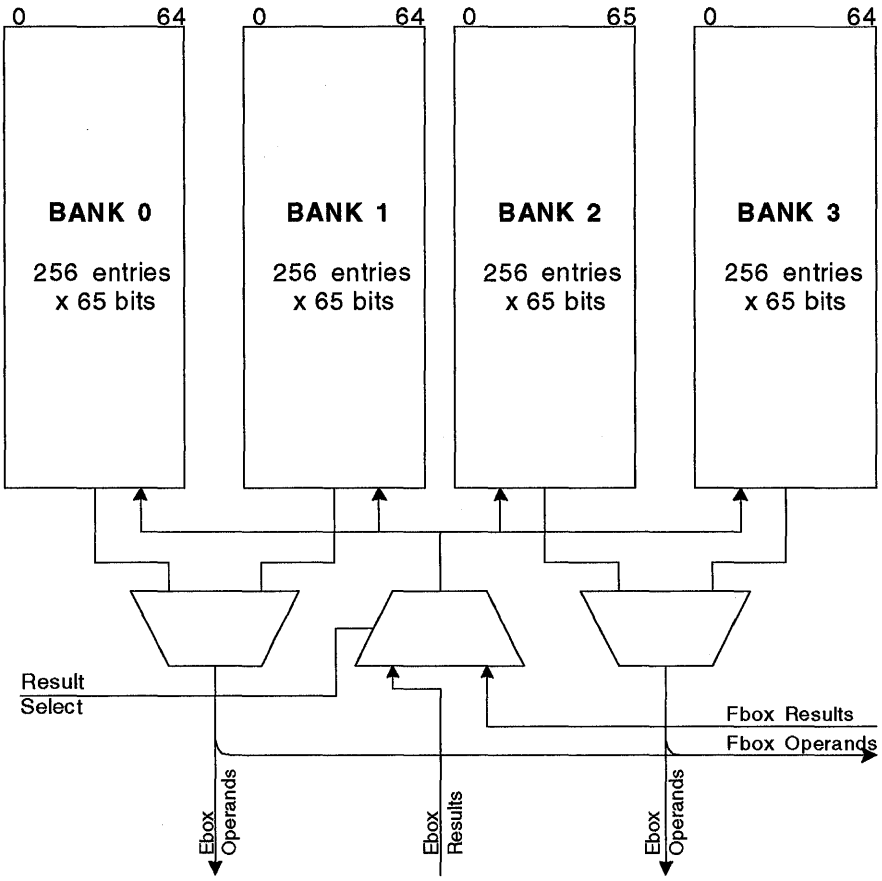
7
Register File

Although the Alpha architecture only defines 64 registers, the 21464 is a multi-threaded, out-of-order machine that requires many more than just 64 registers to keep its pipelines full. The four independent threads require 64 registers each and an additional 256 temporary registers are used to rename registers of inflight instructions to eliminate write-after read and write-after-write conflicts. At 65 bits per entry, 512-entries totals to a 4KB register file.

Eight parallel execution units can consume up to 16 source operands and can produce up to eight results per cycle. Although implementing 32K 'not-so-little' ram cells with 16 read and 8 write ports each is not trivial, defining a register file with fewer than 16 read or 8 write ports would create many other problems. The Qbox would either be forced to issue instructions based on the number of operands needed from the register file, or trap whenever the set of issued instructions needed more than the available number of ports. Brute force was deemed preferable to further complicating instruction picking or sacrificing performance to traps so the current Register File has a full 16 read and 8 write ports.

Internally, the Register File is structured as two iden ical 512-entry register groups each with eight read and eight write ports. Each group services half the operands needed. Coherency is maintained by writing both groups at the same time. To keep the physical structures more controllable, each group is further partitioned into two 256-entry banks where the high-order address bit serves as a bank select.

Figure 7–1 Register File Block Diagram



7.1 Test Structures

7.1.1 Timing

Cycle mnemonics are used throughout this document to identify the relative timing of signals. The following table identifies the cycle relationships assumed by this document.

Qbox	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8						
Reg. File				R0	R1	R2	R3	R4	R5	R6	R7	R8	Rw	R1
Ebox								E0	E1	E2	E3	E4		
Fbox								F0	F1	F2	F3	F4		
Mbox									M0	M1	M2			

Each cycle is further subdivided into two phases, the first half of a cycle is the 'A' phase; the second half of a cycle is the 'B' phase. A timing specification R1A refers to the first phase of cycle R1.

All timing references in this spec refer to the latch that launched the data, when significant transit time may be involved, that time or an expected arrival time will be separately stated.

7.1.2 Read Timing

Table 7–1 shows the Register File read timing.

Table 7–1 Register File Read Timing

	Q3	Q4 R0		Q5 R1		Q6 R2		Q7 R3		Q8 E0 F0
	B	A	B	A	B	A	B	A	B	A
Qbox	Lookup src	Drive Source Pointers								
Registry File				Decode	Read	Bank Mux	Drive to Ebox & Fbox			
Ebox									Bypass Opbus	Execute
Fbox									Bypass Opbus	Execute

7.1.3 Write/Read Timing

Table 7–2 shows the Register File write/read timing.

Table 7–2 Register File Write/Read Timing

	Q13 R9 E4 F4		Q13 Rw E5 F5		R1 E6 F6		R2 E7 F7		R3	
	A	B	A	B	A	B	A	B	A	B
Registry File	Internally distribute mux & write control		Decode/ Mux	Write	Decode	Read	Bank Mux	Drive to Ebox & Fbox		
Ebox	Drive Results									Bypass Opbus
Fbox	Drive Results									Bypass Opbus

7.2 External Interfaces

7.2.1 Qbox to Register File Interface

The Register File is controlled completely by the Qbox. As soon as the set of instructions to execute next is known, the Qbox sends the set of source operand pointers and the Register File begins the lookup process. Since there are often situations where fewer

than eight instructions are picked, some of the instructions need fewer than two operands or some of the operands map to architectural registers R31 or F31, a valid bit is also passed with each source operand pointer. Operand source pointers and valid flags are driven from a Q4A latch in the Qbox, spend a cycle in transit and are received by the Register File in an R1A latch.

Most instructions that issue eventually return result to the Register File. The Qbox supplies a set of destination physical register numbers so the Register File knows where to write results. The Register File must also know 'if' it should write a result. When fewer than eight instructions are issued or instructions are issued that either do not write a result or write registers R31 or F31, the Register File must be prevented from trashing valid physical register contents. A valid bit is also provided with each destination pointer to disable updates.

To kick-off the decoding as early as possible, the Register File needs the write control information before the actual result data. The Qbox sends the write control signals from a Q5A latch; a cycle is spent in transit before being received by the Register File into an R2A latch. The Register File then pipes the data along for seven cycles before decoding for the write. Placing the FIFO in the Register File was convenient for the Qbox and allows the Register File to push the distribution delay back into the FIFO stages.

The Register File receives separate result busses from the Ebox and Fbox and merges them into a common result stream since the instruction picking and result caching guarantees that there are no conflicts. The Qbox supplies a control bit for each of the four Fbox result pipelines that are shared with Ebox results to indicate which result is valid. This bit is sent with the other write control information from a Q5A latch. Since the Register File can do no wrong, there is no need for any status or return information.

7.2.2 Ebox to Register File Interface

The Ebox has eight execution pipes; each pipe requires two input operands and produces a single result. The operand and result vectors are directly mapped such that the A operand for picker N is simply the Ebox_OPA[N] and the B operand is Ebox_OPB[N].

7.2.3 Fbox to Register File Interface

The Fbox only has four execution pipelines corresponding to pickers 0, 1, 2 and 3. The Fbox also has store pipes on pickers 4 and 5, but the Ebox forwards floating-store data to the Mbox whenever the operand is in the Register File eliminating the need to forward the extra two operands to the Fbox. The Register File latches the corresponding Ebox operands and sends them to the Fbox from an R3A latch.

7.2.4 Global Register File Interface

The intention is to clock the Register File primarily off MAC+2. Minimal thought has been put into reset requirements.

8

Floating-Point Execution Units — the Fbox

The Fbox executes all Alpha floating-point instructions, in addition to the new paired single-precision instructions. It receives instructions from the Qbox via the Ebox, and operands from the Register File, the Load Data buses (up to three), or its own Register Caches. The Fbox returns floating-point results to the Register File and floating-point store data to the Mbox, again via the Ebox. The Fbox returns exception information to the Qbox.

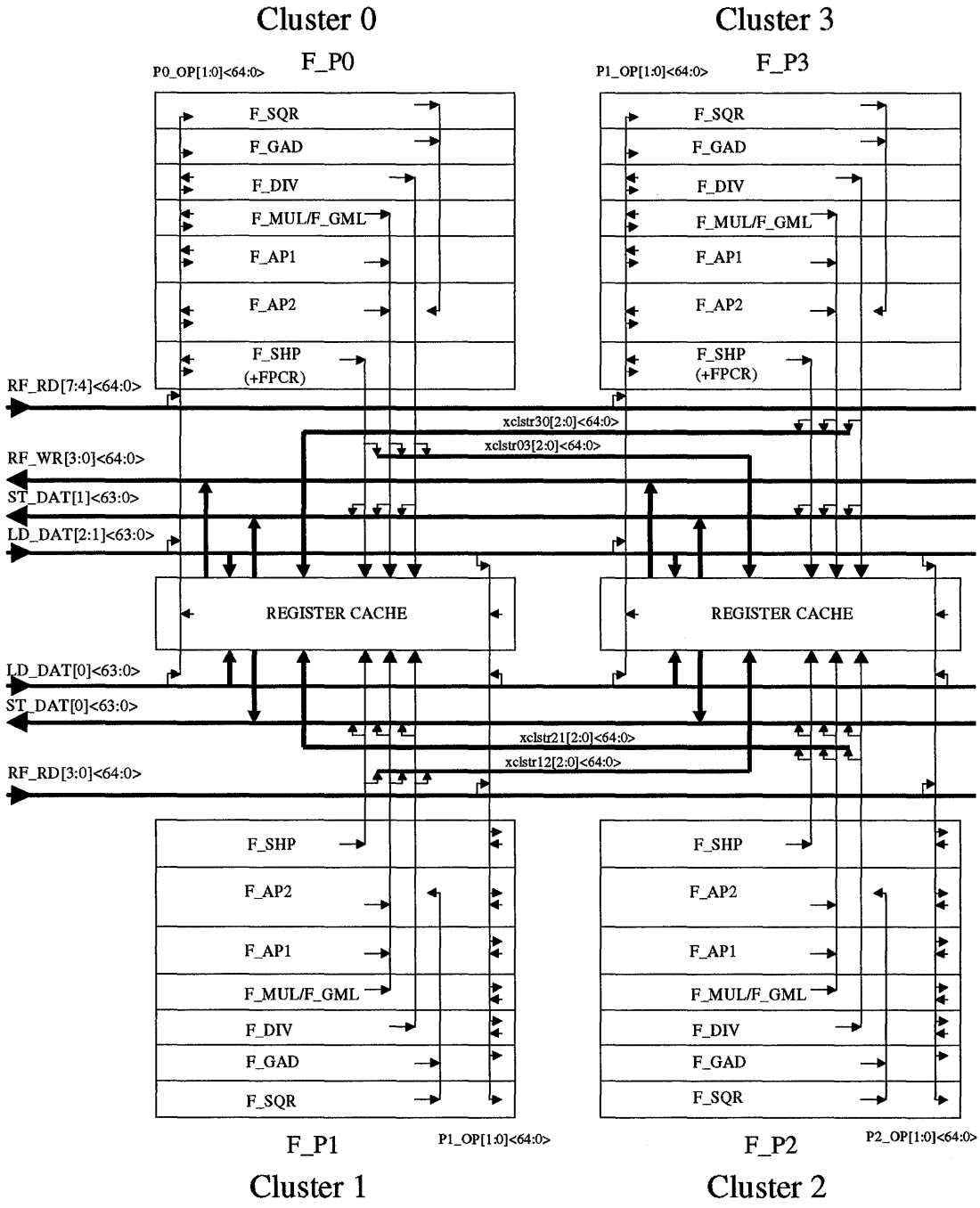
The Fbox is organized as four identical clusters, each cluster consisting of one execution pipeline. The four pipelines, referred as F_P0, F_P1, F_P2, and F_P3, allow up to four floating-point operate instructions to be issued each cycle. Two copies of a register cache – one for each set of two pipelines, are included to allow the results of recently completed instructions to be used with minimal delay. Each pipeline contains the functional units needed to execute the various floating-point instructions. The functional units, their latencies, and the instructions they execute are shown in Table 8–1. Figure 8–1 shows a high-level Fbox block diagram.

Table 8–1 Fbox Pipeline Functional Units, Instructions, and Latencies

Functional Unit	Instructions	Latency
Graphics ADD : F_GAD	Paired SP except PMUL, PARCPL, and PARSQRT	4 cycles
Graphics MUL : F_GML	Paired SP MUL type instructions: PMUL, PARCPL, PARSQRT	3 cycles
Mull Unit : F_MUL	MUL	3 cycles
Divider : F_DIV	DIV	13 cycles – double precision 8 cycles – single precision
Square root : F_SQR	SQRT	33 cycles – double precision 18 cycles – single precision
Add pipe 1 : F_AP1	ADD,SUB,CMP	3 cycles
Add pipe 2 : F_AP2	ADD/SUB (align>1), CVTff, CVTfq, CVTqf, CVTql, CVTlq	3 cycles
Short pipe : F_SHP	CPYSx, FCMOV, FBxx	1 cycle
	Special operands (Zeros, Denormal OPD, NANs, INF,RES.OPD),INPUT EXCEPTIONS, Mx_FPCR	3 cycles

NOTE: The F_SHP unit can supply a result for CPYSx, FCMOV, and FBxx instructions in one cycle. This pipeline is also used to compute results for all non –finite operands such as Denormals, NaNs, Infinity as well as zero operands. The F_SHP pipeline also detects all input exceptions and supplies the appropriate result.

Figure 8–1 Fbox Organization



8.1 Major Sections

The Fbox consists of an interface section and four pipelines organized as four clusters. Each of the pipelines has several functional units. The following sections describe each of these units and the last section describes the instruction flows for each of the floating-point instructions.

8.2 Interface Section

The Interface section is responsible for communications between the Fbox and the rest of the chip, and for internal communications between the four Fbox pipelines and two register caches.

8.2.1 External Interface

The Fbox Interface can receive incoming operand data from the Register File or the Mbox, and instructions from the Qbox, both of which it transmits to any of four Fbox pipelines (F_P0, F_P1, F_P2, & F_P3). The instructions and load operands from the Mbox are piped through the Ebox before reaching the Fbox. The interface is subdivided into the following three subsections:

- Register Cache (F_RGC) – contains staging logic and static ram which latch and hold recently generated result data of the Fbox pipelines as well as copies of incoming floating point loads. The result data is eventually dispatched to the Register File. However, this result and load data can be used in subsequent floating point operations without incurring the transit time delay in returning data from the Register File.
- Operand Steering Unit (F_OSU) – performs comparisons against incoming Physical Register (Preg) numbers to determine the source of input operands to the Fbox pipelines.
- Interface Control (F_INT) – performs a partial decode of opcode, function code and thread processor unit (tpu) to determine if a valid floating-point instruction has been issued. It also contains logic which allows direct access to internal operand buses from Register File operand buses, and logic to dispatch floating point store data to the Ebox from either result data of Fbox pipelines F_P0 and F_P1, or from the register cache.

8.2.2 Qbox Timing to Fbox

Floating-point instructions are issued by the Qbox, which transmits the opcode, function code, thread select information, and source and destination physical register numbers to the Fbox Interface. The Preg numbers go to the Operand Steering Unit (OSU) to control operand bypassing and reads from the register cache. The thread select information controls updates of the Floating-Point Control Registers (FPCR), and is used to determine if a valid instruction has been issued by the Interface Control. Source Preg numbers are transmitted from the Qbox to the Ebox in Q4 (same as FW), are latched and travel through the Ebox in Q5 (FX), reach the Fbox and are latched in Q6 (FY), to begin comparisons in the F_OSU in that B phase. The destination Preg numbers are dispatched by the Qbox a cycle later in Q5, travel through the Ebox in Q6, and are latched in the F_OSU in Q7 (FZ). The opcode, function code, and thread select information

leave the Qbox in cycle Q5 and travel through the Ebox to be latched near the Fbox in cycle Q6 (FY). This allows approximately 1 1/2 cycles for internal Fbox routing and decoding prior to execution in cycle F0.

The Fbox vcu returns exception information back to the Qbox. The Fbox VCU returns branch mispredict signals in F0B, to arrive at Pbox in F2A.

The following diagram illustrates the timing of the Fbox pipelines. Tables describing the Fbox /Qbox interfaces are also shown.

8.2.3 Fbox Pipeline Timing

Table 8–2 shows the operation of a single Fbox pipe with all operands coming from the

Table 8–2 Operation of a Single Fbox Pipe — all Operands From Register File

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
Q5	Q6			Q0	Q1	Q2	Q3	Q4	Q5	Q6			
R1	R2	R3						RW	R1	R2	R3		
			E0	E1	E2	E3	E4					E0	E1
FX	FY	FZ	F0	F1	F2	F3	XMT					F0	F1
			M0	M1	M2								
SRC PREG	OPC INFO	RF DATA	Fbox EXECUTE				RES TO RF				RES BACK		

Register File.

8.2.4 Register File/Operand Bus

Input operands for issued floating-point instructions can be supplied without delay from another functional unit in the same pipe, with a one cycle delay from a functional unit in another cluster, from the register cache, or from the Register File. The Register File supplies up to eight operands per cycle, corresponding to the maximum issue rate of four floating-point instructions per cycle. Up to four results are returned to the Register File per cycle, one for each Fbox pipe.

Operands from the Register File are input to the Fbox pipelines on differential, low-swing, operand buses. These busses are also used for bypassing results from other functional units within the same cluster, results from other clusters, and incoming load data. They are also used to transfer operands to the functional units from the register cache.

The operand buses begin evaluation at the start of the B-phase (FZB). The rising edge of clock at the start of the following A-phase is used to sense the differential data on the operand bus, while the operand buses pre-charge in the same A-phase in preparation for a new transaction in the following B-phase. Source operands from either the register cache or the Register File must be valid at the input to the Fbox pipelines by phase R3B (FZB), one phase before instruction execution begins in F0A. The output result buses are sent back to the Register File early in phase F4A.

8.2.5 Loads/Stores to/from Fbox

Fbox doesn't have direct access to store or load data moving to and from the memory hierarchy and the Register File. The Ebox is responsible transmitting floating-point load data to the Fbox from Mbox. The Mbox is expected to re-align the load such that the sign bit and exponent are contained in the most significant byte of the data quadword for VAX style floating point formats, as is already the case for IEEE style formats. The only portion of the floating point format the Fbox is responsible for is extending the eight bit exponent of single precision data to eleven bits. Three dedicated load busses are used to transfer up to three loads to the Fbox each cycle. The load data can be bypassed directly onto the operand buses, and they are also written to the register cache for later use. Floating point load operands are generated and transmitted to the Fbox with a four cycle latency, one cycle longer than for the Ebox, and arrive near the Fbox Interface at the beginning of phase F3A, to complete formatting for a potential bypass to an internal operand bus in phase F3B.

When a load misses in the cache or the Mbox queue, the data returned to the Fbox is not correct, and the operation using the load data itself is retried at a later time when the load data is ready. To insure that the load data as a result of miss does not corrupt rest of the processor state, the Mbox supplies a bit called poison bit a few gate delays after the load data. This poison state for floating point loads is maintained in the Ebox. The logical state of the poison bit in a resulting operation is maintained by a logical OR of the poison bits of both input operands.

If the source of floating-point store data originates from the Fbox, it is forwarded to the Ebox, which is responsible for formatting the store data and transmitting it to the Mbox. The Fbox can deliver up to two store results per cycle, each on a dedicated bus. Each store bus can source data directly from the result buses of one of the four Fbox pipes, either F_P0 or F_P1, or the register cache. A previous result from any of the four Fbox pipes or three floating-point load pipes located in the register cache can be the data source on either store bus. There are also direct connections between the load and store buses in the Fbox. A floating-point load that is the source operand for a store the following cycle is allowable without first having to access the register cache.

The floating point store data buses are driven to the Ebox in F0A.

The Ebox dispatches the data for integer-to-floating-point convert instructions [ITOFx] to the Fbox as if it were a floating-point load, over the weak load data bus. Fbox is responsible sign extension of the exponent as in the case of a normal floating-point load operation, before use in the Fbox pipelines. Ebox is responsible for sign extension of the exponent before transmitting ItoF data to the Register File. The Qbox can issue only a single ItoF instruction per cycle.

Floating-point-to-integer (FtoI) convert instructions [FTOIx] operate in a similar manner to floating-point stores. Two FtoI instructions can be issued per cycle by the Qbox, and only in place of stores. Therefore, Ebox asserts control information to Fbox to indicate a floating-point store, whether it is a store or a FtoI instruction, and the Fbox Interface is not aware of the distinction. As with stores, Ebox is responsible for conversion of the data.

Table 8–3 shows a timing diagram for load data.

Table 8–3 Timing for Load Data

Cycle	1	2	3	4	5	6	7	8	9	10
Q5	Q6			Q0	Q1	Q2	Q3	Q4	Q5	Q6
			E0	E1	E2	E0	E1			
			F0 LD	F1	F2	F3	F0	F1		
			M0	M1	M2					
	OPC INFO	RF DATA		OPC INFO	LOAD DATA	Floating-point instruction using LD data				

Table 8–4 Pipeline Stages of Fbox Register Cache

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
Q5	Q6	Q7	Q8	Q9	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
R1	R2	R3	R4	R5	R6	R7	R8	R9	Rwrt	R1	R2	R3	R4
EX	EY	EZ	E0	E1	E2	E3	E4	E5	E6	E7	E8	EZ	E0
FX	FY	FZ	F0	F1	F2	F3	F4	F5	F6	F7	F8	FZ	F0
Frgc Stages	1Cyc	Latency	Byp	Stg1	Stg2	Stg3	Stg4/ Rc0	Rc1	Rc2	Rc3	Rc4	RgFl	Byp
	3Cyc	Latency			Byp	Stg3	Stg4/ Rc0	Rc1	Rc2	Rc3	Rc4	RgFl	Byp
	4Cyc	Latency				Byp	Stg4/ Rc0	Rc1	Rc2	Rc3	Rc4	RgFl	Byp

8.2.6 Register Cache (F_RGC)

The Fbox Register Cache is used to store local copies of recent results from all four Fbox pipes, as well as incoming loads forwarded from the Ebox. The stored data can be used as source operands in subsequent floating-point operations without waiting for the data to traverse the round trip to and from the Register File. The register cache is subdivided into two separate structures. First the staging logic is used to equalize the cycle latency of the result data to match the longest execution time of the functional units in the Fbox pipes. The execution times of the functional units in each Fbox pipe are one, three, and four cycles, with a result bus required for each different latency. The staging logic serializes the results from three data buses to one after four cycles. This single result is forwarded to the Register File and to the static RAM, the final data structure of the Register Cache. There are seven sets of staging logic in each copy of the Register Cache, one set for each of four Fbox pipes and three floating-point load pipes. A single SRAM structure per copy of the Register Cache is organized physically into an array of 35 rows or entries, each 65 bits wide, consisting of 64 data bits and a single predicate bit for the FCMOVx instructions. Logically the SRAM is organized into seven banks of five entries each for the four Fbox pipelines and three floating load pipelines.

The Register Cache will hold result data for a window in time that varies from five to eight clock cycles, depending on the execution time of the floating-point operation. After this the result data is available in the Register File and it is dropped from the Register Cache. Floating-point load data is held in the Register Cache for five cycles.

There are two copies of the register cache, one each for two pipes (or clusters) in the Fbox. Each copy of the register cache is an exact duplicate of the other. The staging logic in the register cache consists of level sensitive d-type latches and 2-to-1 multiplexers. It can be viewed logically as a shift register, with the ability to transfer result data from each stage to the internal operand buses for use in subsequent operations. Each entry in the SRAM is also capable of transferring stored result data to the operand bus. The result of a floating-point operation with a one cycle execution time in an Fbox pipe must be latched and held in the Register Cache for eight cycles until the result data is available in the Register File, as shown in the timing diagram above. The staging logic holds this result for four cycles (Stg1, Stg2, Stg3, and Stg4), and the SRAM holds the result for five cycles (Rc0, Rc1, Rc2, Rc3, and Rc4), with one overlapping stage between the staging logic and the SRAM (Stg4 and Rc0). This A-phase latch in the staging logic is used chiefly to hold the result data valid to ensure it is written successfully to the SRAM, but it also helps alleviate a critical timing path. In the event that result data is accessed or read from the Register Cache the same cycle it is written into the SRAM, the hold latch in the staging logic is used to transfer the result to the operand bus instead of the SRAM. This prevents a read after write access to the same SRAM entry within a single cycle.

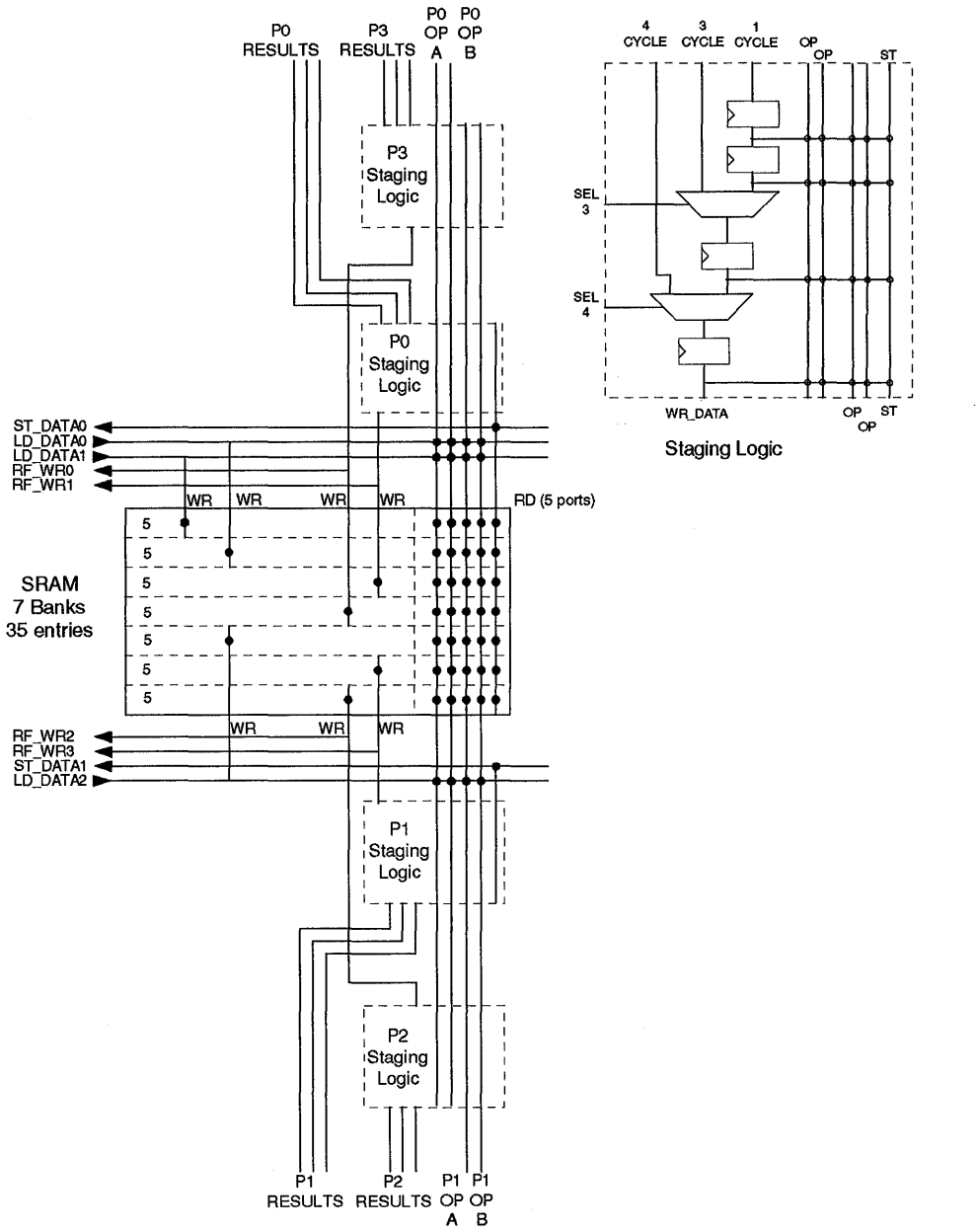
The result of a three cycle floating-point operation is held in the Register Cache for six cycles before it is dropped, two cycles in the staging logic (Stg3 and Stg4), and again five cycle in the SRAM, with one overlapping stage. A result of a four cycle flop is held in the Register Cache for five cycles, which is also the case for floating-point load data.

Figure 2 is a diagram of one copy of a Fbox Register Cache. Each of the four Fbox pipelines can produce a maximum of three results per cycle, one for each flop execution latency, plus up to three floating-point loads must be stored each cycle. Therefore, each copy of the Register Cache must latch and hold up to fifteen results and loads per cycle. Since this data first enters the staging logic, which serializes the data to one result from each of seven pipes, the SRAM receiving this data requires only a single write port, though seven separate inputs.

Each Register Cache can source up to five operands, requiring five read ports in both the SRAM and staging logic. The pair of Fbox pipelines associated with each Register Cache copy require two operands each for a total of four, while the fifth read port sources one of two floating-point store buses driven to the Ebox. Each Register Cache independently supplies store data, thereby supporting two floating-point stores per cycle.

Register Cache entries are read in the B-phase, and they directly drive the operand bus in the same phase. Writes occur in the A-phase of the cycle immediately following the arrival of the results from the functional units (also driven during the B-phase). Because results from each pipe are written into both copies of the register cache, the register cache serves as the transfer point for cross-cluster data between pipes (clusters) in the Fbox. A one-cycle delay penalty is imposed for bypassing results of one cluster to any other cluster.

Figure 8–2 Register Cache



8.2.7 The Operand Steering Unit (F_OSU)

The structure of the Operand Steering Unit is analogous to the register cache described above. It consists of a ten-bit control datapath which stores the destination Physical Register (Preg) numbers, and is organized into shift registers that feed into a content addressable memory (CAM). This corresponds to the staging logic and static ram of the register cache. There are two copies of the OSU, one for each copy of the Register Cache. Each copy of the OSU contains seven sets of shift registers, one set for each of four Fbox pipes or three floating-point load pipes. Each set of OSU shift registers controlling read access to a set of staging logic in the Register Cache for an Fbox pipe contains four stages of registers, and in some cases a fifth stage to handle local bypasses of

result data in an Fbox pipe. These pipeline stages are named as Byp, Stg1, Stg2, Stg3, and Stg4 in the above timing diagram and correspond to cycles F0, F1, F2, F3, and F4 in the Fbox pipeline. Only two stages of shift registers are necessary for controlling the staging logic of a floating load pipe in the Register Cache. Each copy of the OSU also contains a single CAM physically organized into one array of 35 entries that are 10 bits wide. Logically the CAM is organized into seven banks of five entries each, one bank for each of the seven Fbox and floating-point load pipes. The structure is identical to the SRAM in the Register Cache.

At each shift register stage and entry of the OSU a nine-bit exclusive or (XOR) is performed to compare incoming source Physical Register numbers to the stored destination Preg numbers every cycle. The tenth bit or valid bit of both source and destination Preg numbers is logically anded to validate the comparison. Typically there are five XOR's per stage of shift registers or entry in the CAM, one for each read port of the Register Cache (the bypass stage has only three). A match in an XOR of a shift register stage or CAM entry of the OSU indicates that the equivalent stage of the staging logic or SRAM entry of the Register Cache is the source for an input operand to one of the Fbox pipes or store buses. As a consequence of the hit in the OSU, result or load data is transferred to the appropriate operand bus in the Register Cache. A hit in the XOR of the bypass shift register stage of the OSU indicates the source of an input operand to an Fbox pipe is the result of a functional unit in the same Fbox pipe. The result data is bypassed locally to the operand bus, without incurring any delay passing through the Register Cache. If there is no hit in the shift registers or CAM of the OSU, source operands to the Fbox pipelines are supplied from the Register File.

The implementation of the internal Fbox operands as low-swing differential buses constitute a large distributed multiplexer with connections to one Register Cache copy, two Fbox pipes, and a store bus. If the Qbox should issue identical destination Physical Register numbers too frequently, these are loaded into the F_OSU and would cause multiple XOR matches in this logic. The consequence of this is multiple sources driving the operand buses, causing invalid data and indeterminate results in the Fbox pipelines. This problem must be prevented at the architectural level by ensuring the Qbox can issue a destination Preg number only once within a nine cycle window of time that it would remain in the F_OSU. In other words, once the Qbox issues a destination Preg number to the F_OSU, it can't be issued again for another ten cycles.

The valid bits of source or destination Physical Register numbers have several uses in the F_OSU (the most significant bit). If a source Preg number is not valid in an otherwise valid floating-point operation, this indicates that architectural register F31 is intended as the source operand, and the internal operand bus is grounded. Should the Qbox issue an non-pipelined or bubble operation to an Fbox pipe such as a divide or square root, it is presently required to issue control information (opcode, function code, tpu, Pregs) to the Fbox twice, the second time once the operation is nearly completed in the Fbox. To ensure that the architectural constraint described above is met, the first time the Qbox issues an non-pipelined operation to the Fbox, the destination Preg number must be invalidated. The second time the instruction for the same bubble operation is issued, the source Preg numbers should be invalidated.

8.2.8 Interface Control (F_INT)

The Interface Control does a partial decode of the opcode, function code and thread processor unit (tpu) passed to it each cycle by the Ebox. It performs this function to determine if a valid floating-point operation (flop) has been issued, and ignores any integer instructions that have been issued. (The Ebox indicates via separate control signals to F_INT whether any floating-point loads or stores have been issued). The instruction decode is also used to determine the execution time of the flop, either one, three, or four cycles, and also to detect non-pipelined or bubble operations, such as floating-point divide or square root operations.

This section also takes in control information from the F_OSU indicating whether any successful comparisons have occurred between source and destination Preg numbers. If not, data from the Register File operand buses is transferred directly to the internal operand buses of the Fbox. The F_INT also contains multiplexers to transfer result data from an Fbox pipe to the store bus in the event of a store bypass. This would be indicated by a match in an XOR of the bypass shift register stage of the F_OSU. Only Fbox pipelines F_P0 and F_P1 of bypassing result data directly to a store bus. Result data from pipes F_P2 and F_P3 can only reach a store bus from the Register Cache, and incur a delay of at least one cycle from the completion of an operation.

8.2.9 Divide and SQRT – Qbox interface

The Divide or the Square Root units in the Fbox pipelines are not pipelined and require multiple cycles to finish the operation. The latencies of the operations are shown in Table 1. The divide unit computes the fraction result and uses the multiplier for exponent result and the multiplier result bus to write the results. The square root unit computes the fraction results and relies on the add pipe (F_AP2) to calculate the exponent results, final rounding, and for exception detection. For this reason, at the end of a square root operation, the square root unit sends the results to the F_AP2 pipeline in the second stage of the pipeline. After computing the final result the F_AP2 pipeline actually writes the result. This also eliminates extra write ports in the register cache and drivers for the operand bus.

In order to reinsert the divide or square root unit results in the Mul or add pipe, issue of all other instructions have to be stopped and a 'bubble' has to be created by the Qbox. The Qbox keeps track of the divide and square root completion and inserts the bubble appropriately. The following time diagrams show the relationship between done and bubble signals.

It is possible to have a divide and square root to request a bubble at the same time. This can be seen in the timing diagrams below, by lining up a divide and square root which are issued at different times. This problem is solved as follows: For divides and square roots, the Qbox detects a collison and always delays the square root by one cycle and

inserts two bubbles – one for divide followed by another for square root. The sequencer in the square root detects this condition and delays the result transfer to the F_AP2 pipeline.

Table 8–5 FDIV_SP (9 cycles) , FDIV_DP (14 cycles)

FDIV in Divider											↓ Drive Exceptions		
R1	R2	R3	F0	F1	F2	F3	F4	F5	F6	F7	← **FDIV_SP**		
F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	← **FDIV_DP**		
—	—		Q1	Q2	Q3	R0	R1	R2	R3	F0	F1	F2	
—	—	—	—	—	—	—	Q3	R0	R1	R2	F0		
				Bubble req - ↑ at Qbox					Bubble ---↑-↑---**Result Bypass**				

Table 8–6 FSQRT_SP (12 CYCLES), FSQRT_DP(28 CYCLES)

FSQRT IN SQRT Unit										Sqrt result in AP2			Drive Exceptions ↓	
F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	← **FSQRT_SP**		
F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31	F32	← **FSQRT_DP**		
—	—	Q1	Q2	Q3	R0	R1	R2	R3	F0	F1	F2	F3	F4	
—	—	—	—	—	—	—	Q3	R0	R1	R2	R3	F0	F1	F2
↑Bubble req to Qbox				↑-Bubble req at Qbox					↑-Bubble			Result bypass		
									SQRT reinjected					

8.2.10 Fbox Exceptions

The Fbox detects the following arithmetic exceptions.

Table 8–7 Arithmetic Exceptions

Exception	Description
Integer overflow	Detected and generated for CVTFQ and CVTQL instructions.
Invalid operation	In addition to illegal operations and invalid operands, VAX reserved operands are included.
Floating overflow	Generated during operate instructions
Floating Underflow	Generated during operate instructions
Inexact result	Generated during operate instructions
DIV by zero	Generated by Divides, and approximate .reciprocal and square root instrcutions.

The Ibox detects reserved opcodes and generates traps. The Ebox detects reserved values in the function fields of valid Fbox opcodes and generates a RESOPC trap.

During the floating-point instruction execution it is possible to generate more than one exception. The possible multiple exceptions are Inexact with floating underflow or floating overflow. The graphics units can generate exceptions for each half of the result, including multiple exceptions on each half as mentioned before. Within the exceptions the input exceptions (Invalid operation, Div by zero) take precedence over output exceptions.

When an exception is detected, the Fbox needs to record the exception status in the FPCR. These status bits are sticky bits i.e. once set only an explicit write using MT_FPCR can clear them The Fbox implicitly reads the status bit and requests an update by PALCODE only if the corresponding status bit is clear. In addition, depending on the trap enables, it needs to signal traps. The trap enables can be part of the instruction as a qualifier or from the FPCR in the form of trap disable bits. The Fbox looks at the opcode and the FPCR bits to enable traps. The Fbox writes the result (IEEE specified non-trapping result for IEEE instructions) to the destination whether traps are enabled or not. Since there are four pipelines in the Fbox, at any given cycle it transmits exception status and traps for four instructions. The various pipelines in the Fbox have different latencies(1 cycle, 3 cycles, or 4 cycles). However, the Fbox signals exception status and traps at one fixed point, F4 cycle of the pipeline as shown in the diagram. Since the Fbox processes instructions in SMT mode, up to four threads, it also sends back the thread ID along with the exceptions to match the trap with the instruction. In addition, when a trap occurs, the software completion flag encoded in the opcode has to be transferred to the trap handler through the operating system. For this purpose it sends the '/S' bit along with the exception information. The PALCODE corresponding to the arithmetic exceptions also updates the IPR – EXCEPTION SUMMARY REGISTER. The mechanism used for updating the FPCR is detailed in the FPCR section.

Table 8–8 Fbox Exception Signaling Timing

	R0	R1	R2	R3					
				E0	E1	E2	E3	E4	E5
					F0	F1	F2	F3	F4
OPCODE VALID			VV						
RESULT BYPASS					--VV 1 CYC		--VV 3 CYC	--VV 4 CYC	
EXCEPTIONS DRIVEN									--VV

The Fbox encodes the exception information, taking multiple exceptions into account and to signal traps as one vector exc_enc. Table 8–9 shows the legend for Table 8–10, which lists the various combinations:

Table 8–9 FPCR Update/Floating-Point Arithmetic Trap Legend

Symbol	Meaning
DZE	Division by zero
INE	Inexact result
INV	Invalid operation
IOV	Integer overflow
OVF	Floating-point overflow
SW	Software completion flag
UNF	Floating-point underflow

Table 8–10 Fbox Retire-Time Exception (RTE) Encodings

Encoding	Retire-Time Disruption	Encoding	Retire-Time Disruption
00XXXX	No Fbox exception	100111	INV DZE
01XXXX	FPCR Update	101000	INV UNF INE
010000	IOV INE	101001	INV OVF INE
010001	INV	101010	INV INE
010110	DZE	101011	DZE UNF INE
010011	UNF INE	101100	Reserved
010100	OVF INE	101101	Reserved
010101	INE	101110	UNF OVF INE
010110	IOV INE INV	101111	Reserved
010111	INV DZE	11xxxx	FP Arith Trap (SW = 1)
011000	INV UNF INE	110000	IOV INE
011001	INV OVF INE	110001	INV
011010	INV INE	110110	DZE
011011	DZE UNF INE	110011	UNF INE
011100	Reserved	110100	OVF INE
011101	Reserved	110101	INE
011110	UNF OVF INE	110110	IOV INE INV
011111	Reserved	110111	INV DZE
10xxxx	FP Arith Trap (SW = 0)	111000	INV UNF INE
100000	IOV INE	111001	INV OVF INE
100001	INV	111010	INV INE
100010	DZE	111011	DZE UNF INE

Table 8–10 Fbox Retire-Time Exception (RTE) Encodings (Continued)

Encoding	Retire-Time Disruption	Encoding	Retire-Time Disruption
100011	UNF INE	111100	Reserved
100100	OVF INE	111101	Reserved
100101	INE	111110	UNF OVF INE
100110	IOV INE INV	111111	Reserved

8.3 Fbox Floating-Point Control Register (FPCR)

The FPCR contains rounding information and trap disable bits used by the floating-point operate instructions, and exception status information from floating-point operate instructions. The FPCR is read from and written to the floating-point registers by the MF_FPCR and MT_FPCR instructions. In addition, all operate instructions use the dynamic rounding mode bits to round the results and the trap disable bits to signal traps when an exception is detected. The Fbox implements all bits specified by the Alpha architecture except the Denormal operand exception disable bit (DNOD). The Fbox does not implement Denormal operand processing. The FPCR format is shown below.

Since the 21464 issues the floating-point instructions out of order, a mechanism to correctly read (for both implicit and explicit readers) and write the FPCR is used. In addition, in SMT mode there can be four threads with their own FPCRs. The floating-point instructions from each thread can be issued to any pipeline in the Fbox. In order to support these features, the Fbox implements two sets (copies) of FPCRs, one for each group of two pipelines. Each set of FPCRs contain four FPCRs. – one FPCR per thread. The thread ID is used to access the correct FPCR. Each FPCR has two elements, a 'committed state' and a 'speculative state'. In order to avoid the score boarding of the registers, the Fbox uses PALCODE and trap mechanism to synchronize the updates.

8.3.1 FPCR Format

Table 8–11shows the format for the Floating-Point Control Register.

Table 8–11 Floating-Point Control Register Format

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47 … 0
SUM	INED	UNFD	UNDZ	DYN_RM		IOV	INE	UNF	OVF	DZE	INV	OVFD	DZED	INVD	DNZ	RAZ/IGN

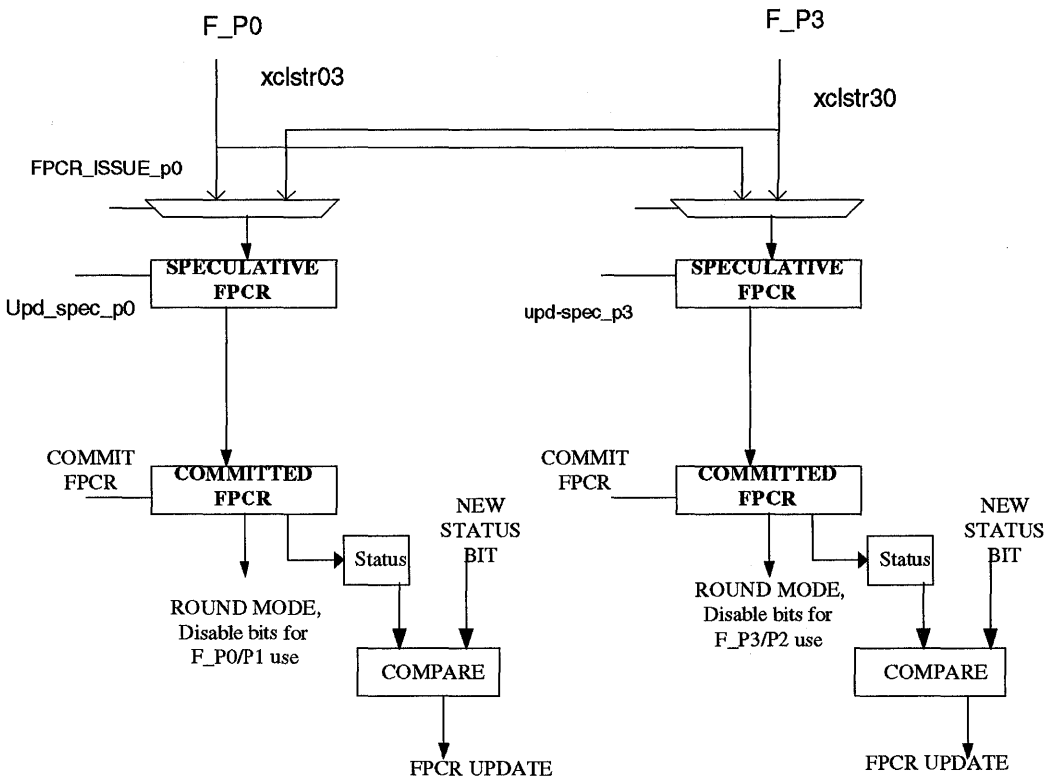
The FPCR is read and written as follows, shown in Figure 8–3:

1. The FPCR needs to be updated for two reasons:
   a. As a result of MT_FPCR instruction.
   b. To update the status information from each operate instruction.

The data in the FPCR needs to be read as a result of MF_FPCR and for the Fbox to use round mode information and disable bits for executing a floating-point operate instruction(implicit read).

2. Whenever a MT_FPCR instruction is issues, Qbox compares the INUM of the last instruction that changed the FPCR(corresponding to the FPCR in the 'speculative register') with the INUM of the current instruction. If the current instruction is older then it signals 'update speculative register'and the data is written to the speculative register; otherwise the data is ignored. When the MT_FPCR is retired a trap to PALCODE is taken where an IFETCHB instruction is executed. At this point, Qbox sends a 'commit FPCR' signal to the Fbox and the speculative register is copied to the committed register. Whenever FPCR status update is required, the Fbox signals a trap to the PALCODE and supplies the exception information on exc_enc signals.
3. This exc_enc information is written to the exception summary register by the hardware. When this trap occurs, all instructions younger than the trigger instruction are invalidated. The PALCODE reads the exception summary register and executes one HW_MT_FPCR to write the status to FPCR as mentioned before, followed by an IFETCHB instruction and exits. The HW_MT_FPCR instruction is executed by the Fbox (in F_SHP pipeline) and the data is written to the speculative FPCR.
4. Since the status bits (IOV, INV, DZE, OVF, UNF, INE) are sticky bits, whenever one of these bits need to be set, Fbox checks if the old bit is already one. If it was a one, Fbox does not request a trap to update the FPCR. Otherwise it transmits the exc_enc, the encoded exception information. This can produce up to six trap requests – one for each status bit for a program. Once the corresponding sticky bit is set, no further traps occur. The Fbox uses the speculative FPCR status information for this purpose.
5. The Qbox sends a commit_FPCR signal to the Fbox as soon as the instruction that triggered the FPCR change is retired.
6. The Fbox uses the committed FPCR DYN_RM and disable bits to round the results and to signal traps.
7. The FPCR is implemented in the F_SHP units of pipelines corresponding to P0 and P3. Whenever a MT_FPCR is executed in any of the two pipes, the second speculative register is copied using the cross cluster bus for the 3-cycle result.

Figure 8–3 FPCR Update Mechanism



8.4 Fbox Multiplier Unit — F_MUL and F_GML

The floating-point multiplier executes the following instructions:

MULF, MULG, MULT, MULS

PMUL, PMULL, PMULH, PARCPL, PARCPLH, PARCPLL, PARSQRT, PARSQRTH, PARSQRTL

8.4.1 FMUL Operation

The 21464 FMUL is fundamentally different from previous Alpha implementations. Unlike previous processors, which used an odd-even array multiplier, the 21464 uses a Wallace tree. This one feature has many far reaching implications:

- 3 cycle FMULs are possible
- The array datapath is 106 bits wide
- Radix-4 booth recode will be used instead of Radix-8
- The least significant bits of the product are not known early.

In phase F0A, the two source operands are read off the operand busses with sense amps. The B operand goes to f_mul_mpc, which does the swizzle and drives the multiplicand. Because radix-4 booth recoding is used, a 3x add which would normally be needed isn't. The swizzle is there to support PMUL instructions. Specifically the PMULL and PMULH instructions require that the low (or high) B operand must be used for both

multiplies. To accomplish this I potentially needed to move either the low or high operand to the opposite location. Also, in phase F0A, the A operand is booth recoded for radix 4. No swizzle is needed but a 3 bit shift is required for PMUL operations. After recoding, the 53 bit fraction results in booth control signals which will result in 27 partial products. One extra partial product is also generated which corrects for deficiencies in the array. (ie the array doesn't fully sign extend all partial products or add the +1 term needed for two's complement arithmetic)

In phase F0B, Wallace compression begins. To sum up the 28 partial products 7 stages of CSA's are required. The quantity of CSA's for each stage are:

Stage 1 – 9

Stage 2 – 6

Stage 3 – 4

Stage 4 – 3

Stage 5 – 2

Stage 6 – 1

Stage 7 – 1

Total – 26

Both phases F0B and F1A are required for CSA stages 1-6. Stage 7 will be performed in phase F1B. The stage 1 CSA has some extra logic incorporated which can conditionally force zero's into the array. This is done to support PMUL's. Ordinarily the array would compute: Ah*Bh<<52+Ah*Bl<<29+Al*Bh<<29+Al*Bl<<0

This would be gibberish for a PMUL, but by selectively introducing zero's into the array the correct result can be had: Ah*Bh<<52 +Ah*0<<29 + Al*0<<29 + Al*Bl = Ah*Bh<<52 + Al*Bl

Notice that Al*Bl can never result in a number that is big enough to affect the Ah*Bh sum which is sitting 52 bits to the left.

Phase F1B and F2A are used for rounding. Two round adders will be built. The first only handles double precision multiplies. This will be the most critical. The second adder handles single precision multiplies, including PMULS, and an add required for the approximate instructions. Because a single precision add is inherently faster than double precision, this adder can be a degenerate copy of the double precision version. Having two round adders significantly simplifies the design and speeds up the hard double precision add. The additional area required for this scheme is approximately 1K cdu's. These round adders differ from the 21264 in two respects. First, the carry in from the least significant bits of the product are not known ahead of time. Instead, they have to computed at the same time the add of the high bits is being done. The second complication is that the sticky bit is also not known ahead of time. It's possible to compute sticky early but it requires a trailing zero count and an add. Because of the PMUL instructions this logic would be doubled. By making the round adder tolerant of a late sticky bit a fair amount of hardware and complexity can be saved.

The final F2B phase is used to route the result back to the operand drivers and then drive the operand bus.

Two parallel exponent additions are required prior to F2A. There is plenty of time for these so they don't merit further discussion.

The approximate instructions are not handled in the main multiplier array. Instead a ROM is used plus 3 PP mux's and 2 CSA's. The most significant 6 bits of the source fraction (or for 1/sqrt, 5 fraction bits plus the lsb of exponent) are used to index into a 64 entry ROM. Three numbers are retrieved: slope(9 bits), slope*3(10 bits), offset(18 bits). While the ROM lookup is happening the next less significant 8 bits of the operand are radix-8 booth recoded. The slope and slope*3 signals act as a multiplicand to 3 PP mux's. These 3 partial products plus the offset are then compressed to 2 numbers with the help of 2 CSA's. The 2 number's will be muxed into the single precision round adder. The ROM is 64x2x2x19=4864 bits. The error of the result will be less than 1 part in 2^14.

8.5 Fbox Add Pipeline

The Fbox ADD pipeline executes ADD, SUB, CMP, CVTXX instructions, and completies the DIV and SQRT instructions. The Add pipeline is divided into two pipelines F_AP1 and F_AP2 pipelines. The F_AP1 pipeline is used for CMP instruction and for effective subtract operations with an exponent difference of 0 or 1.The F_AP2 pipeline executes all the other instructions. The partitioning of the add pipeline is based on the requirements of the effective subtract operation and is described below.

The steps required for implementing an effective subtract operation in a straight forward manner follow:

1. Find the exponent difference of the two operands to align the fractions.
2. Align the smaller operand by shifting the smaller fraction right by the absolute difference of the exponents. This alignment shift can be very large and a 54-bit shifter is required.
3. Subtract the aligned operands (smaller operand from the other). The result can have many leading zeroes if the result is positive or leading ones if the result is negative, when the operands are close.
4. Find the leading zero or the leading one position in the result to normalize. When the exponent difference is zero, the result of subtraction can be negative. In this case the position of leading zero is needed to shift left.
5. Normalize the result by an amount indicated by the leading 1/0 position. When the operands are very close, many leading bits may be canceled and a large left shift may be required.
6. Round the result.

These steps can be minimized by separating the operation into two domains – 1) for exponent difference of 0 or 1 labeled 'near domain' and 2) for exponent difference greater than 1, labeled 'far domain'. This separation uses the following observations:

1. In the near domain, the alignment shift is atmost 1, which can be accomplished by a mux. Thus, there is need for a huge alignment shifter.
2. In the near domain, if any normalization is performed there is no need for rounding. The reason behind this is since the alignment is atmost 1, only the round bit(the bit below the LSB) can be one and when a normalization is performed this bit also shifts back into fraction. Hence no roundin is required.

3. In the far domain, the maximum normalization required is 1. Since the original operands are normalized (≥ 1.0 for IEEE, ≥ 0.5 for Vax), and the aligned operand with a right shift ≥ 1 has a value < 0.5 for IEEE, a value < 0.25 for VAX, the result of the subtract has to be ≥ 0.5 for IEEE, or ≥ 0.25 for VAX.

With those observations, the effective subtract can be performed using the following steps:

Near Domain(Exp difference =0,1)	Far Domain(Exp difference > 1)	
1N.Predict Exponent difference and align. Determine Leading-1/0 position using the input operands.	1F. Determine exponent difference	Observation 1.
2N.Subtract the smaller operand.	2F. Align the smaller operand	
3N.Normalize the result	3F. Subtract the smaller operand with rounding	Observations 2,3

In step 1N above, the least two significant bits of the exponent are used to predict the exponent difference of 0 or 1. If the actual exponent difference turns out to be > 1, the far domain computes the result.

After step 2N, the most significant bit of the result can be used to determine if normalization is required. By observation 2 above, if no normalization is required, rounding may be necessary; in this case we can switch to step 3F. If normalization is required there is no need for rounding and the operation can be completed in step 3N.

With the above principles, the F_AP1 implements the 'near domain' and F_AP2 implements the 'far domain'.Since the COMPARE instruction is similar to an effective subtract with exponent difference of 0, it is implemented in the F_AP1 pipe. The far domain pipe (F_AP2) implements all the other instructions.

8.6 Fbox Add Pipe1 — F_AP1

The Fbox add pipe is a 3-cycle pipe. F_AP1 is used for the effective subtract (Ediff = 0,1 case) and for the compare instructions. The data into the add pipe is assumed to be binary vectors coming from the register file, register cache, or from the other pipes, etc. The input operands are always assumed to be non-zero.

The F_AP1 pipe does not handle the cases when one or both the operands is a true zero, NaN, infinity or denormal or when it sees reserved operands or dirty zeros. The short pipe handles these exceptions. The output latch that drives the F_AP1 result bus is disabled in these cases. The add pipe also does not do rounding. Addition is done in an earlier stage taking advantage of the fact that rounding is not required for Ediff =0, or Ediff =1 with normalization. The control is transferred to F_AP2 if normalization is not required and rounding may or may not be required.

The add pipe F_AP1 has a fraction data path, exponent data path and control. The fraction data path is 55 bits wide, including the sign bit, hidden bit and the round bit. The exponent data path is 11 bits wide.

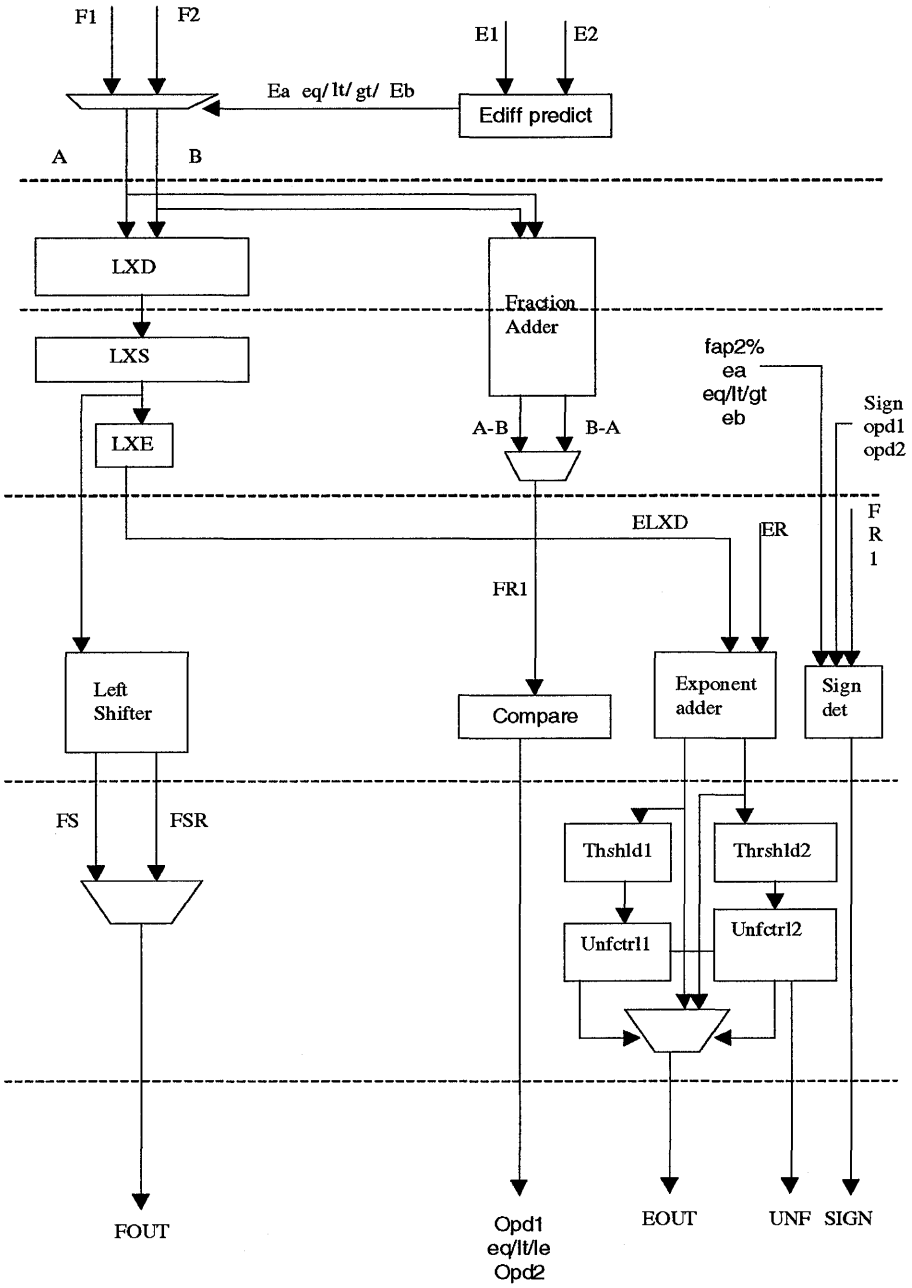
Fig 1 shows the basic outline of the addpipe F_Ap1 and the main functional blocks, namely, Ediff Predict, LXD, LXS, LXE, Adder, left shifter, compare_blk, and exponent adder, and the underflow detect. A more detailed version of the block diagram is also available.

Two least significant bits of the exponent are used in 'Ediff predict' to determine if the exponent1 is equal to, less than, or greater than exponent2. This data is used as control signals to select the vectors A and B from the fractions F1, F2, F1/2, and F2/2. The fraction adder does effective subtract on the two vectors A and B. The leading 1/0 in A-B vector is partly detected in LXD, which determines all the 1's in the vector. LXS completes the leading 1(0) detect operation by doing a strip of all 1's(0's) except the first 1(0) from the output of LXD and drives the shifter control that normalizes the output of the adder. The output of LXS, in other words, the normalization amount is encoded into a 6-bit vector (ELXD). ELXD is later subtracted from the result exponent (Er). This operation is done in the exponent adder, the output of which is the final exponent.

Compare instructions for A=/<=/< B are handled by the compare_blk which examines the signs, exponents and the fractions (in that order). In the event that the vectors A and B have the same sign and exponent, the difference fracA-fracB is examined to generate the compare results.

Underflow detection is done in two stages namely threshold and unf_detect. Threshold determines if there is an underflow or not, or if there is a possibility of having an underflow. In the case of a possible underflow, Unf_ctrl1 and Unf_ctrl2 produce the necessary controls that select the appropriate fraction and exponent.

Figure 8–4 F_API Block Diagram



8.6.1 Operation

8.6.1.0.1 Phase F0A

The operands are passed through differential sense amplifiers in this phase. The sense amps are assumed to be B latches followed by A latches. Hence the data is read from the sense amps after the rising edge of the clock with some delay due to the sense amps. We are assuming that the sense amps will introduce a delay of 300 ps. The output of the

sense amps is a 52 bit vector to which the round bit is concatenated at the LSB position. The two 53 bit vectors, one for operand fa and the other for operand fb are sent to the multiplexer which selects either fa, fa/2, fb or fb/2. The select lines of the multiplexer come from the ediff predict unit. Ediff predict is a 2 bit predict logic that uses bits 0 and 1 of the exponent to determine whether ea = eb, ea > eb or ea < eb.

Table 8–12 Exponent Difference Estimation

Ea<1:0>	Eb<1:0>	Potential Exponent Difference	Fraction Operation Performed[1]
00	00	0	Fa-Fb
00	01	-1	Fb-Fa/2
00	10	>1	X
00	11	+1	Fa-Fb/2
01	00	+1	Fa-Fb/2
01	01	0	Fa-Fb
01	10	-1	Fb-Fa/2
01	11	>1	X
10	00	>1	X
10	01	+1	Fa-Fb/2
10	10	0	Fa-Fb
10	11	-1	Fb-Fa/2
11	00	-1	Fb-Fa/2
11	01	>1	X
11	10	+1	Fa-Fb/2
11	11	0	Fa-Fb

[1] Fa and Fb are the fraction parts of operands.

The outputs of the mux are sent to LXD and the Adder in the next phase along with the hidden and the sign bits.

8.6.1.0.2 Phase F0B

LXD is the leading 1/0 detect logic. It does f_ap1_m1%A-f_ap1_m1%B and does leading 1/0 detect. It generates a vector with a '1' in the ;eft most position corresponding to the leading 1 and possibly several 1's to the right. These spurious 1's to the right of the leading 1 are stripped in LXS. LXD is needed to know the number of bits we need to shift the result of the effective subtract for normalization. Leading 1 detect is needed if f_ap1_m1%A-f_ap1_m1%B >0 and leading 0 is needed if f_ap1_m1%A-f_ap1_m1%B <0.

The Adder for effective subtract and compare instructions is started off in the same phase as LXD. The adder (FAD) performs A-B and B-A based on the following logic equations:

```
F_AP1_M1%A - F_AP1_M1%B =  F_AP1_M1%A + ~ F_AP1_M1%B + 1
F_AP1_M1%B - F_AP1_M1%A =  F_AP1_M1%B + ~ F_AP1_M1%A + 1
                        =  ~ ( F_AP1_M1%A + ~ F_AP1_M1%B + 0)
```

8.6.1.0.3 Phase F1A

LXS is the leading 1/0 strip logic. It keeps only the 1st '1' and strips all the rest of the '1's from the leading 1 detect's output. This will be used to directly drive the shifter control lines. Input to the LXS is the output of LXD with some modifications to the round bit. Since it is possible to have an LXD output without any 1's in the case of zero result that occurs in ea = eb, we modify the round bit as follows. If ea_eq_eb is true, the round bit which is the LSB is forced to a logic high. Otherwise, the original bit value of R bit is maintained. This ensures that LXS will never produce an all zero result which is necessary for the shifter control lines.

Remaining part of the addition in the Adder block too is completed in this phase. One of the outputs of the adder is selected in this phase by the mux M2. A-B is selected if the adder result is positive or if the ediff predict predicted that ea *lt*eb or ea_gt_eb. But on the other hand, if ea = eb, then there is the possiblity that the adder result is negative, in which case we need to select the B-A result.

The output of the mux M2 is checked to see the hidden bit. If the hidden bit is a '1', it indicates that normalization is not required. This signal is sent to the add pipe FAP_2. Once it gets this signal, it does rounding if necessary and drives the output bus. FAP_1 does not drive the output bus in this case.

The mux M3 selects the correct exponent, ea or eb. Logic equations for M3:

```
F_ap1_m3%Exp  = f_ap1_sa%ea  if f_ap1_sa%ea - f_ap1_sa%eb >= 0
              = f_ap1_sa%eb  if f_ap1_sa%ea - f_ap1_sa%eb < 0
```

For normalizing the adder result, the *Left shifter* shifts it by the number of bit positions indicated in the output of LXS

LXE encodes the LXS result into 6 bits. The encoded LXS is used by the exponent adder to generate the final exponent.

8.6.1.0.4 Phase F1B

The exponent adder is used to generate result exponent and result exponent+ 1. The exponent adder used here is a 13 bit exponent adder. The 13th bit in the MSB position is '0' and is added to the exponent 'exp' to keep track of overflow and sign. The extra 6 MSB bits added to the elxd are also all zeros.

```
F_ap1_ead%Res_exp   =  f_ap1_m3%exp - f_ap1_lxe%elxd
F_ap1_ead%Res_expp1=  f_ap1_m3%exp - f_ap1_lxe%elxd + 1
```

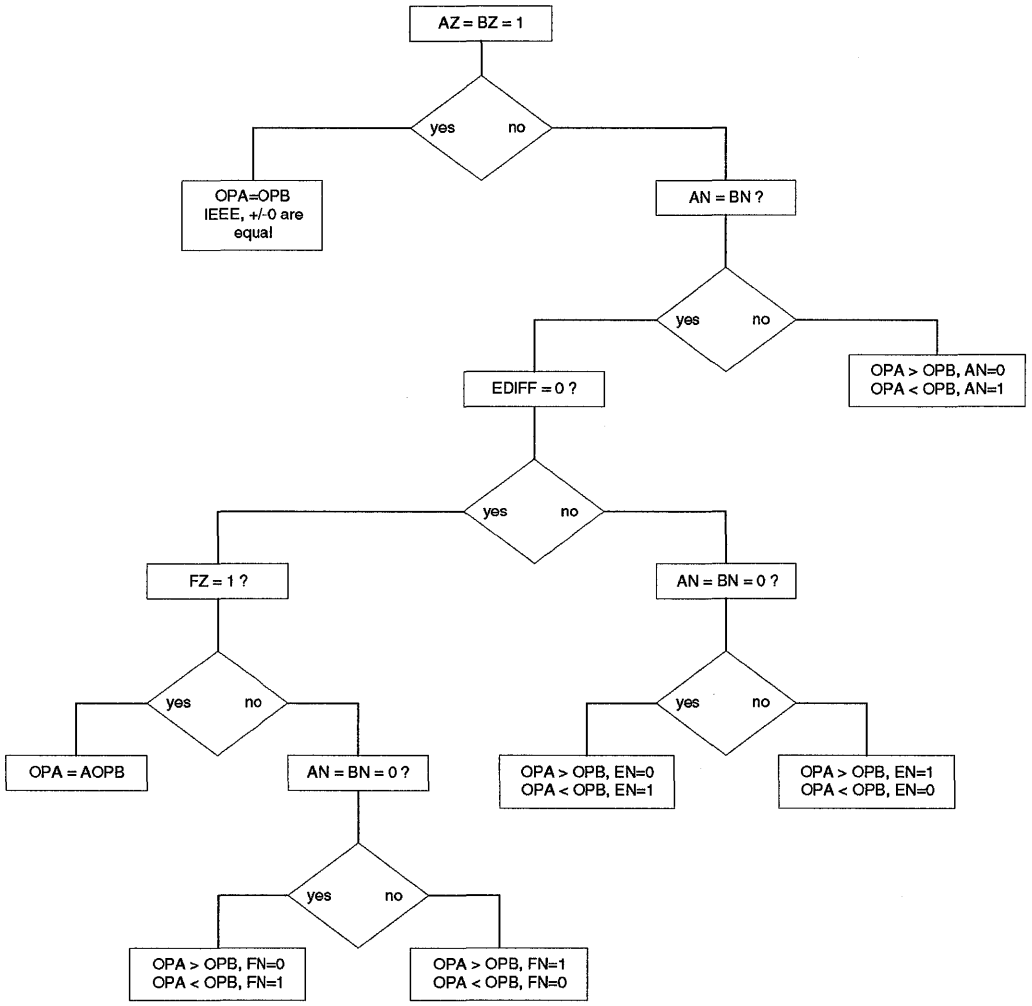
Res_expp1 is the result exponent + 1. This is used when a right shift is done on the left-shifter output to correct the over-estimated LXD.

The compare_blk is used to execute the compare instructions. It looks at the signs of the two operands, their exponents and the sign of the adder fraction result, in this order. Depending on which compare instruction is to be executed, it will determine if a = or < or > b. Each of the compare conditions are calculated as follows:

F_ap1%Cmp_eq = '1' if ((sa=sb) & (ea=eb) and (fa=fb))
F_ap1%Cmp_gt = '1' if ((sa='0' & sb='1') | (sa=sb & ea>eb) | (sa=sb & ea=eb & fa > fb))
F_ap1%Cmp_lt = '1' if ((sa='1' & sb='0') | (sa=sb & ea<eb) | (sa=sb & ea=eb & fa < fb))

The compare instruction decision flow is hsown in the following figure. AZ/BZ indicate A/B is zero, AN/BN indicate sign of A/B.

Figure 8–5 CMP Instruction Logic



Sign-detect detects the sign of the result. The logic equations for sign-detect follows:

Sign = (((ea > eb) & sign of opd a) or ((ea < eb) & sign of opd b) or ((ea = eb) & sign of fraction difference))

8.6.1.0.5 Phase F2A

The output of LXD may be off by 1 bit. Hence the output of the left shifter may have to be shifted right by one bit position for the overestimate in the LXD. F_ap1_ls%FS_* represents the adder result left shifted by what LXD and LXS indicated. F_ap1_ls%FSR_* is f_ap1_ls%FS_* right shifted by one bit position. One of these two vectors are selected by the mux M4 and one of the vectors, res_exp or res_expp1 are selected by the mux M5 as follows:

F_ap1_ls%FS<54> (bit A0) = '0'	Fraction output = F_ap1_ls%FS ;
	Exponent output = F_ap1_ead%res_exp
F_ap1_ls%FS<54> (bit A0) = '1'	Fraction output = F_ap1_ls%FSR ;
	Exponent output = F_ap1_ead%res_exp + 1

The *underflow* detect is done in two stages, namely *threshold* and *unf_detect*:

- *Threshold* looks at the result exponent from the previous phase and the data type, for example S,T,G or F and by doing range checking, will determine if there is a definite underflow, definite no underflow, or a predicted underflow.
- *Unf_detect* on the other hand determines if there is an underflow if threshold has indicated a predicted underflow. It does so based on the following equations:-

Underflow occurs if :

- Threshold gave a definite underflow, or
- Threshold gave a predicted underflow & fraction result != 0 & bit A0 of fraction =0

UNF1 = def_unf or (pred_unf & FS != 0 & FS<54> or A0 = '0')
UNF2 = def_unf or (pred_unf & FSR != 0 & FSR<54> or A0 = '0')

No underflow occurs if :

- Threshold gave a definite no underflow, or
- Threshold gave a predicted underflow & bit A0 of fraction =1, or
- Threshold gave a predicted underflow & fraction result = 0

NO_UNF1 = def_nounf or (pred_unf & FS = 0) or (pred_unf & FS<54>or A0='1')
NO_UNF2 = def_nounf or (pred_unf & FSR = 0) or (pred_unf & FSR<54>or A0='1')

Mux M4 selects one of the two fraction results, FS or FSR as given in the following equations:

```
FOUT = FS if A0 = '0'
     = FSR  if A0 = '1'
     = 0 if LXS_H<0> ='1' (zero detect) or
           UNF1 or UNF2
```

Similarly,

```
EOUT = exp - elxd if A0 ='0'
     = exp - elxd + 1 if A0 ='1'
```

```
= 0 if LXS_H<0>= '1' (zero detect) or
      UNF1 or UNF2
```

The sign, exponent and fraction outputs are all zeros if there is an underflow or if the input operands are equal. Otherwise the sign, exponent and fraction results are driven out on the output bus in phase F2B.

8.6.1.0.6 Phase F2B

This is the output bypass phase. The outputs are driven on the result or the operand bus at the beginning of this phase.

8.7 Fbox Add Pipe2 — F_AP2

F_AP2 is responsible for eff.ADD, eff.SUB with ediff>1, CVTLQ/QL, CVTqf, CVTff and CVTfq. In the case of eff. SUB with ediff=1 and no normalization happening, rounding is required because there is one bit shifted out of the datapth and wasn't brought back by normalization. Therefore, we need to account for that bit not in datapath by rounding. Since F_AP1 doesn't have round adder, F_AP2 will output the result instead of F_AP1. In this case, F_AP1 will send a signal to F_AP2. The exponent and rounding of DIV/SQRT will also be handled by F_AP2. Multiplier handles its own rounding and exponent. The latency of F_AP2 pipeline is three cycles. The datapath can be divided into 3 sections: fraction, exponent, and control. Fraction dateapath is 64 -bit plus R bit and exponent datapath is 12-bit. At top, the data is driven by sense amps which are fired by the rising edge of fclk.

In the case of unary instructions, OPB is used and OPA is ignored. Special operands are handled by F_SP.

The following sections describe the operation of F_AP2.

8.7.1 Cycle 1 Operation

8.7.1.1 Fraction:

Floating-point operands : 1 bit Sgn; 11 bit Exp; 52 bit Frac. --> Sign goes to control section; Exp goes to Exp datapath.

Fraction part is droped into [B01....B52] bit by bit.

[B00] is hidden 1 and [A10..A00] are forced 0.

Integer operands : 64 bit --> The operand is passed as [A10..B52]

This transformation happens in "format_a" and "format_b". OPB can be signed- magnitude floating-point numbers or 64 bit 2's complement numbers. OPA is always a floating-point number in signed magnitude format. They are transformed, according to data type and op code, to fit into fraction datapath. The interpretation of datapath format depends on op code. Note that for IEEE, hidden 1 is located at A00. Exponent datapath doesn't need to do anything for this because the 1 will go back to A00 eventually. For eff. Sub, the result out of round adder will be either 0.1xxx or 0.01xxx, which are different from 1.xxx and 0.1xxx for eff. Add. To simplify rounding overflow detection and exp calculation, both eff. SUB operands are always shifted left by 1 to align with eff. Add. This is done here too. Note that Exp needs to take these situations into account.

For CVTQL, bits<31:30> is copied to bits<34:33>. Later, shifter left shifts the operand by 29 bits. For CVTLQ, <63:62> are moved to <60,59> and <63:62> are sign extended(by copying <63> to <62>). Later, shifter right shift the operand by 29 bits.

For D-floating, fraction is moved right by 3 bits and chopped. Bits 1 and 0 are lost for D format.

SHF_MUX and PASS_MUX are used to determine which operand will be sent to shifter. In the cases of CVTxx, OPB is selected by default. For ADD/SUB, the fraction of the smaller operand is sent to shifter for alignment. The larger operand is sent to Rounding CSA in F1A. Since Ediff>=1 is always true, the result of round adder is always positive. Therefore, we don't need magnitude comparator here. The control signal to both MUXes is EN, which is the sign of Ea-Eb. SHF_MUX also handles the first step of negation. If the instruction is effective SUB or CVTQf/CVTfQ with a negative operand, the operand will be inverted bitwise before shifting. However, we still need to add 1 to LSB to complete 2's complement. This is done by the help of TRZ and the one is combined with sticky bit. For instance, a number to be negated is shifted right by 10 bits. There are 10 bits out of datapath and a 1 need to be added at B52+10, which doesn't exist in datapath. However, for the 1 to be propagated to B52 inside datapath, all 10 bits shifted out must be all 1's. Otherwise the 1 will be killed somewhere and ignored. Therefore, to see a 1 coming in at B52, TRZ must be larger than 10. Although the 1 for 2's complement can be killed, it may still change sticky bit.

The output of SHF_MUX is sent to Lo_mux and Hi_mux controlled by shf_setup. These cells will setup for left shift or right shift. The shifter needs a 128-bit operand. In the case of a left shift, the operand is sent to LO_word with HI_word filled with 0 or 1. In the case of a right shift, the operand is sent to HI_word with LO_word filled with 0 or 1. Left/Righ shift is determined by the sign of elxd. The filling of extension word is determined by instructions shown as following.

Table 8–13 Filing of Extension Word for F_AP2 Instructions

	Hi_Word	Lo_Word	
Add	0	operand	-- Rsh only
Sub	1	~operand	-- Rsh only
CVTQF(Rsh,posQ)	0	operand	
CVTQF(Rsh,negQ)	0	~operand	-- operand is inverted if neg
CVTQF(Lsh,posQ)		operand	
CVTQF(Lsh,negQ)	1	~operand	
CVTFQ(Lsh,posF)	0	operand	
CVTFQ(Lsh,negF)	1	~operand	Sign ext
CVTFQ(Lsh,posF)	operand	0	
CVTFQ(Lsh,negF)	~operand	1	

Table 8–13 Filing of Extension Word for F_AP2 Instructions

CVTLQ(Rsh)	Sign_ext operand		
CVTQL(Lsh)	operand	0	
CVTFF			

Note: The inversion of operand is done by SHF_MUX

TRZE(A) and TRZE(B) counts and encodes the number of trailing zeros in A and B respectively. It's also possible to save one encoder by putting TRZ_mux before encoder. TRZ_mux pick the eTRZ of the smaller operand, which is to be used to calculate sticky bit. It may be useful to do TRZ on B_bar for 2's complement negation. TRZ_MUX is also controlled by EN. CVTQF_FLE detects the position of leading 0/1 for CVTQF only. It strips all lower order 1's or 0's and leaves only the leading 0/1 in output like LXS. Physically, the stripping is done in encoded domain so that encoding and stripping are done in one step. The output is then decoded to control shifter. For detailed description of shifter control, see 2.2. If leading one is in [B53..B0], the encoded output is (53..0). If leading one is in [A0..A9], the output is encoded as (-01..-10) for exponent calculation. The sign also indicates the direction of shifting.

FLE will also detect if there is a 1 in the upper LW, whcih will cause an integer overflow in CVTQL.

Note the calculation of the sticky bit needs 3 elements : etrz, ediff or elxd, and a constant. This will be explained later.

8.7.1.2 Exponent

In the first phase, in order to compute absolute value of exponent difference(ediff), two 11-bit adders calculate Ea-Eb and Eb-Ea concurrently. On top of the adders, muxes are used to force value of Ea and Eb for specific instructions. The results of adders determine which fraction to shift and exact_ediff. The sign of Eb-Ea and exact_ediff>1 is sent to F_AP1 because F_AP1 doesn't have the adders. Ediff is used to determine shift amount of Add/Sub alignment, calculate sticky bit, and control many muxes.

In the second phase, if F_AP2 will handle exponents for DIV/SQRT, their exponents(Eb-Ea) will be frozen in a latch, div/sqrt frz, and wait until the fraction part is almost done. ermux_1 picks a constant for certain instructions like CVTQF. SHR_3 is for CVTDG.

Er_mux determine the result to be used in final exponent calculation. In the case of eff. ADD/SUB, the exponent of larger operand is picked. For DIV/SQRT, the exponent is from DIV/SQRT frz. For CVTxx instructions, a constant is supplied.

8.7.1.3 Control

Exp_mux chooses among Constants, Ea-Eb, and Eb-Ea for the ediff to be used in driving shifter and sticky bit calculation. Since ediff is encoded, it needs to be decoded before driving the shifter. The decoding is done in 2 steps here but this may change with physical implementation.

CVT_mux, CSA&CASC-LAT, and PGK is the first step of sticky bit calculation. To calculate sticky bit, 3 numbers are involved: etrz, ediff (elxd, in the case of CVTQF), and a constant specified by instruction. Ediff/elxd selection is done by CVT_mux. Then

a CSA reduce 3 numbers to 2 numbers so they can be used to drive PGK. To save time, CSA is combined with a cascode header to work as a latch. The width of this datapath is 6-bit.

8.7.2 Cycle 2 Operation

8.7.2.1 Fraction

L/R shifter handles the alignment for ADD/SUB and normalization for CVTxx instructions. For alignments, it is always a right shift. For normalizations, both left and right shifts are possible. To handle all conditions fast, both operand and control are arranged accordingly before the clock edge. Therefore, we can squeeze in the rounding CSA in F2A. There are 65 control lines which are also arranged specifically for different instructions. The 65 control lines are coded as 00-64. The shifter works as a 65-1 mux. 00 selects A10-B53 of INPUT_LOW; 01 selects B52 of INPUT_HI and A10-B52 of INPUT_LOW; 02 selects B51-B52 of INPUT_HI and A10-B51 of INPUT_LOW, and so forth. In the case of CVTQF, if leading one is in [A9..A0] (A10 is sign), shifter sets up to do right shift and LXD(A9..A0) is mapped to control lines (01..10). For leading one's in [B00..B53], shifter sets up to do left shift and LXD(B53..B0) is mapped to control lines (11..64) respectively. For alignments, shifter sets up for right shift only. Ediff(00..63) is mapped to control lines (0..63).

Rounding CSA compresses two operands and sticky bit and rounding constant to 2 operands for PGK. Note the sticky bit is only one bit, so we can encode more informaiton into this number. Say STICKY=1 and another 1 is required for negation, STICKY is forced to be 2 (10b). For different data format, the 1 of 2's complement and the sticky bit need to be inserted in different bit position. Round Adder starts in F2B and takes one cycle.

8.7.2.2 Exponent/Control

Exponent adder calculates (er,er+1) for eff. Add and (er,er-1) for eff. Sub. The selection is based on A0 and B0 of round_adder result.

Threshold logic is to detect conditions which may become overflow or underflow by the result of rounding adder. In other words, it detects if the instruction is on the verge of OVF/UNF. It is basically a ROM taking inputs from Exp_adder and Op_code decoder. Note that in register file, all single precision exponents are extended to fit into double precision fields by adding 896d=380h.

OVF/UNF Detect is tightly coupled with Threshold logic to detect definite OVF/UNF conditions and force exponents to Emin or Emax accordingly.

The sticky bit calculation is done in F1A with the following equations:

eff. sub & fs	s= etrz - (edif+27) < 0	
eff. sub & gt	s= etrz - (edif -2) < 0	
eff. add & fs	s= etrz - (edif+28) < 0	
eff. add & gt	s= etrz - (edif -1) < 0	
cvtfq & ~en	s= etrz - (edif -1) < 0	right shift

cvtqf & FS	s= etrz - (29-elxd) < 0 [53 - elxd - etrz >24 or 53]
cvtqf & gt	s= etrz - (-elxd) < 0
cvtts	s= etrz - (edif+28) < 0
cvtfq & en	s= etrz - (edif+63) < 0 left shift

Note that *s* is a Boolean function, so we only need a carry chain here to determine sign. The exact sum is unnecessary.

8.7.3 Cycle 3 Operation

8.7.3.1 Fraction

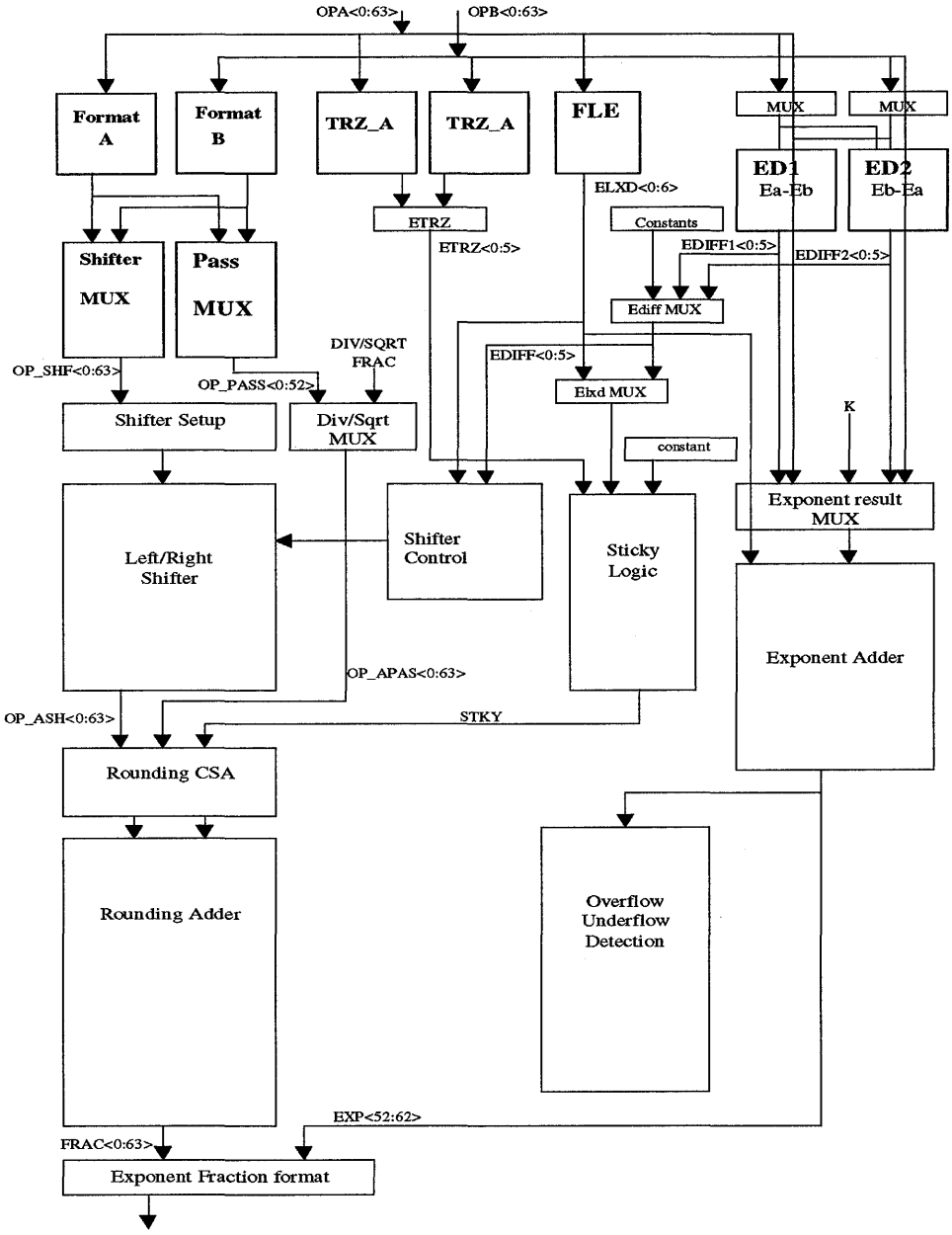
Round Adder takes 3/4 cycle to complete. Logic is built into the adder to detect potential overflow/underflow. Fraction mux handles possible one bit shift of fraction and pick the right exponent and merge it into 64-bit datapath for output. There is a special caese for F_AP2 to handle. For ediff=1, it's possible that there is no need for normalization and thus rounding is necessary. Since F_AP1 has no rounding capability, the rounding is done in F_AP2.

In F3B, F_AP2 drives operand buses.

8.7.3.2 Exponent/Control

If exponent has the potential to ovf/unf, one of the exponent is forced to Emax/Emin.

Figure 8–6 F_AP2 Block Diagram



8.8 Fbox Short Pipe — F_SHP

The short pipeline F_SHP implements single cycle latency instructions FCMOVxx, CPYSx, and FBRxx instructions. It also implements special or unusual operand handling and FPCR in two of the pipelines. These operations have full 3- cycle latency. The F_SHP also supplies rounding information to other pipelines and functional units from the instruction and the FPCR. It collects all the exception information and communicates exception information to the Qbox. The F_SHP mainly consists of three distinct sections, sharing instruction decode and output drivers but little else.

8.8.1 Short Instructions

These ten instructions require little processing - in six cases a zero compare and in the remaining four just selection of operand bits via a mux.

They could be executed in a single cycle, but wiring constraints[1] may favour a three cycle unit. Any performance impact of this is likely to be dominated by FCMOVxx latency - as FCMOVxx is the most commonly used of the short instructions and FCMOVxx passes through the F_SHP twice, so lengthening the pipe adds four cycles to FCMOVxx latency.

8.8.1.1 CPYS, CPYSN, CPYSE

These three instructions merely copy fields of the input operands to the result, as shown below. The fields copied are controlled purely by the instruction — there is no data dependence.

Instruction	P_{in}	S_{out}	E_{out}	F_{out}	Description
CPYS	x	S_A	E_B	F_B	Copy Sign
CPYSN	x	$!S_A$	E_B	F_B	Copy Sign Negated
CPYSE	x	S_A	E_A	F_B	Copy Sign and Exponent
FCMOVxx1	x	S_B	E_B	F_B	Conditional Move part 1
FCMOV2	0	S_A	E_A	F_A	Conditional Move part 2
FCMOV2	1	S_B	E_B	F_B	Conditional Move part 2

8.8.1.2 FCMOVEQ, FCMOVGE, FCMOVGT, FCMOVLE, FCMOVLT, FCMOVNE

As apecified in Section 2.11.2.3, FCMOVxx copies the second source operand to the destination if the first source operand passes the test specified in the instruction else leaves the destination unchanged. This can often replace sequences of code using conditional branches, avoiding possible branch misprediction giving code that is faster and has more predictable delays.

```
FCMOVxx Fa, Fb, Fc
```

is functionally equivalent to

```
FByy Fa, label ; yy = not xx
CPYS Fb, Fb, Fc
label:
```

but avoids the branch.

See Section 2.11.2 for a complete discussion of FCMOVxx instruction processing.

8.8.2 Unusual Input Operands

There are several unusual input operands that each arithmetic function must handle specially:

1 A single cycle unit requires its own output and bypass busses, whereas a three cycle unit can share the busses used by the other three cycle units (add, multiply etc.)

- IEEE NaN
- IEEE Denormal
- IEEE Infinities
- Zero
- VAX dirty zero
- VAX reserved operand

Rather than require that each unit detects unusual input operands and generates correct results and exceptions, the unusual cases are handled by the F_SHP.

The F_SHP snoops on the operation and operand busses. When it notices a combination that requires special handling it asserts a pipe-global suppress output signal[1] in phase F1B. On receiving this an arithmetic unit must suppress its output (and may cancel its calculation), and the F_SHP will drive the correct result instead, and possibly throw an exception.

The F_SHP must detect unusual input operands in all floating-point types used by the Fbox - IEEE single and double precision, VAX single and double, and packed graphics (two IEEE single format values packed into bits 63:32 and bits 31:0 of the bus).

8.8.2.1 Unusual Cases

- If Fb is a NaN, propagate the quietened NaN

  Else if Fa is a NaN and Fa is used, propagate the quietened NaN

  (and throw an INV exception if FPCR<INVD> is clear and either was an SNaN).
- If either operand is denormal then

  if FPCR<DNZ> is set, treat the operand as zero

  else throw an INV exception.
- If both operands are usual numbers (non-zero finite numbers) the F_SHP does nothing, and the result is generated by the arithmetic unit.
- If either operand is non-usual the result is generated as shown in the following tables. For each instruction the result to select is defined by the type of the two operands A and B, and can be one of the two operands, a true zero, IEEE infinity or the canonical quiet NaN (sign bit, all exponent and fraction MSB set, remaining bits of fraction cleared).
- In the case of VAX or IEEE single or double format unusual result the F_SHP asserts SUPPRESSLOW_H and SUPPRESSHIGH_H to force the active arithmetic unit to release the output bus, and drives the value shown below. The sign bit is treated as a special case (it may have to copied from A, B, a constant, not B or A xor B), the fraction and exponent are copied from A, B or a constant (0, Inf or CQNaN).

---

1 Actually two signals SUPPRESSLOW_H and SUPPRESSHIGH_H to handle graphics instructions where the graphics unit or multiplier may have to drive one half of the output whilst the F_SHP drives the other. For non-graphics instructions both will be asserted.

- In the case of graphics format data there are two independent values in each operand, one on the upper half of the bus, one on the lower. The two halves are handled independently, each as described above. It is possible that one half of the result needs to be driven by the F_SHP whilst the other needs to be driven by the arithmetic unit. SUPPRESSLOW_H and SUPPRESSHIGH_H are driven independently to signal the arithmetic unit which bits it needs to drive.

8.8.2.2 IEEE Data

8.8.2.2.1 ADDS, ADDT

A	+Inf	-Inf	0	Normal
B	—	—	—	—
+Inf	A	CQNaN	B	B
-Inf	CQNaN	A	B	B
0	A	A	A	A
Normal	A	A	B	(driven by addpipe)

8.8.2.2.2 DIVS, DIVT

$S_{out} = S_a$ xor S_b

A	Inf	0	Normal
B	—	—	—
Inf	CQNaN	0	0
0	Inf (DZE)	CQNaN (DZE)	Inf (DZE)
Normal	A	A	(driven by AP2)

8.8.2.2.3 MULS, MULT

$S_{out} = S_a$ xor S_b in all cases

A	Inf	0	Normal
B			
Inf	A	CQNaN	B
0	CQNaN	A	B
Normal	A	A	(driven by mulpipe)

8.8.2.2.4 SQRTS, SQRTT

Operand	Result
0	0
Inf	Inf
-ve	CQNaN (INV)
Normal	(driven by AP2)

8.8.2.2.5 SUBS, SUBT

A	+Inf	-Inf	0	Normal
B	—	—	—	—
+Inf	CQNaN	A	-B	-B
-Inf	A	CQNaN	-B	-B
0	A	A	A	A
Normal	A	A	-B	(driven by addpipe)

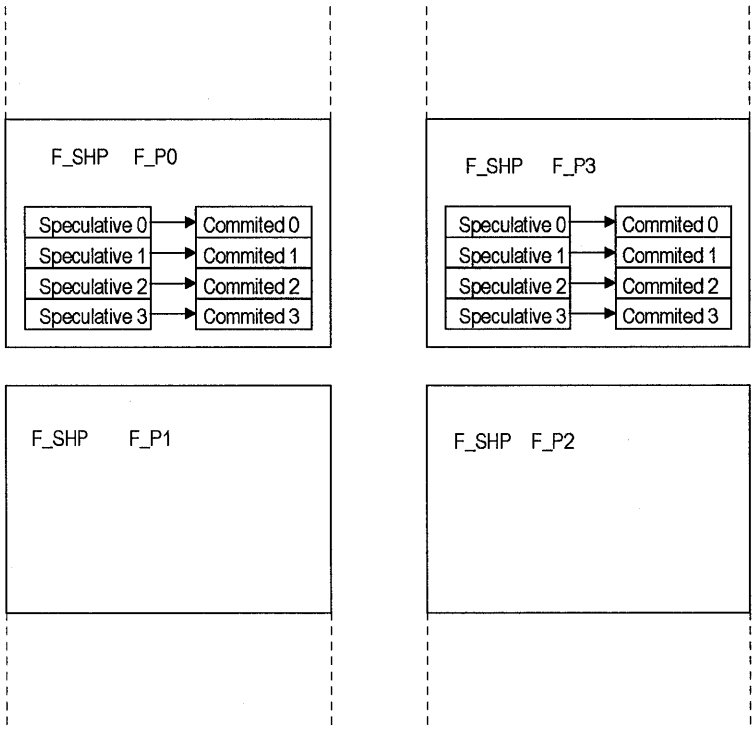
8.8.3 Floating-Point Control Register (FPCR)

The FPCR contains dynamic rounding information, trap disable bits and exception status.

Two of the four F_SHPs include a copy of the FPCR. The north-west F_SHP broadcasts rounding mode information to and handles exceptions for the two western pipes whilst the north-east F_SHP handles the two eastern pipes. The two southern F_SHPs contain no FPCR state, but do contain some FPCR related logic.

Each of the two northern F_SHPs contains four FPCRs, one FPCR for each thread. Each FPCR consists of two registers, one containing speculative state, the other committed machine

Figure 8–7 Fbox Floating-Point Control Registers



8.8.3.1 Reading the FPCR

The FPCR is read explicitly by the MF_FPCR instruction. In response to this instruction, the F_SHP reads the FPCR for the current thread and writes the result bus. As the two copies of the FPCR are defined to be identical, this instruction can be issued to either of the two northern pipes. Any implicit execution barriers needed will be performed by the palcode routine, so the current committed value of the FPCR can be returned immediately.

The FPCR (commited) is read implicitly by every dynamically rounded floating-point instruction (DYN_RM bits) and by every possible arithmetic trap (trap disable bits).

8.8.3.2 Dynamic Rounding

Every arithmetic instruction includes explicit rounding bits <12:11>

Table 8–14 Arithmetic Instruction Explicit Dynamic Rounding Bits

Bit	Meaning
00	Round chop
01	Round to -infinity
10	Normal (nearest/even) rounding
11	Dynamic rounding

In the case of dynamic rounding, the instruction should use the rounding mode specified by <59:58> of the FPCR for that thread instead, as follows:

Table 8–15 FPCR Dynamic Rounding Bits

Bit	Meaning
00	Round chop
01	Round to -infinity
10	Normal (nearest/even) rounding
11	Round to +infinity

This is handled by each of the four F_SHPs rewriting the rounding mode of each instruction issued to its pipe. This means each arithmetic unit can use the rounding mode it receives directly without needing to handle the details of dynamic rounding.

As any change to the dynamic rounding mode bits must be isolated by execution barriers the committed state of the FPCR of the appropriate thread can be used directly.

This functionality must be provided by the southern F_SHPs also, so the northern F_SHPs must drive the dynamic rounding mode of each thread to their southern equivalent (eight wires).

8.8.3.3 Exceptions

There are five maskable arithmetic exceptions:

Table 8–16 Maskable Exceptions

Exception	Meaning
INE	Inexact
OVF	Overflow
UNF	Underflow
DZE	Divide by zero
INV	Invalid operation

If FPCR<DNZ> is not set then any attempt to use a denormal operand will throw an INV exception, *even if* FPCR<INVD> is set. If FPCR<DNZ> is set all denormal values are treated as true zero.

INV and DZE are generated only by the F_SHP, so can be handled by the unusual operand handling logic. Because the F_SHP can handle any operation with a zero operand (is this true? TBS) it can handle forcing denormals to zero without involving the arithmetic unit.

An arithmetic unit that produces an overflow, underflow or inexact result must assert *unit*_UNF, *unit*_OVF or *unit*_INE respectively. In the case of overflow or inexact result the unit must drive the IEEE correct result. In the case of an underflow a unit must drive true zero. The F_SHP will throw the appropriate exception if it is unmasked, and if the instruction enables the exception (for INE, if bits 15:13 = 111. For UNF, if bit 13 = 1).

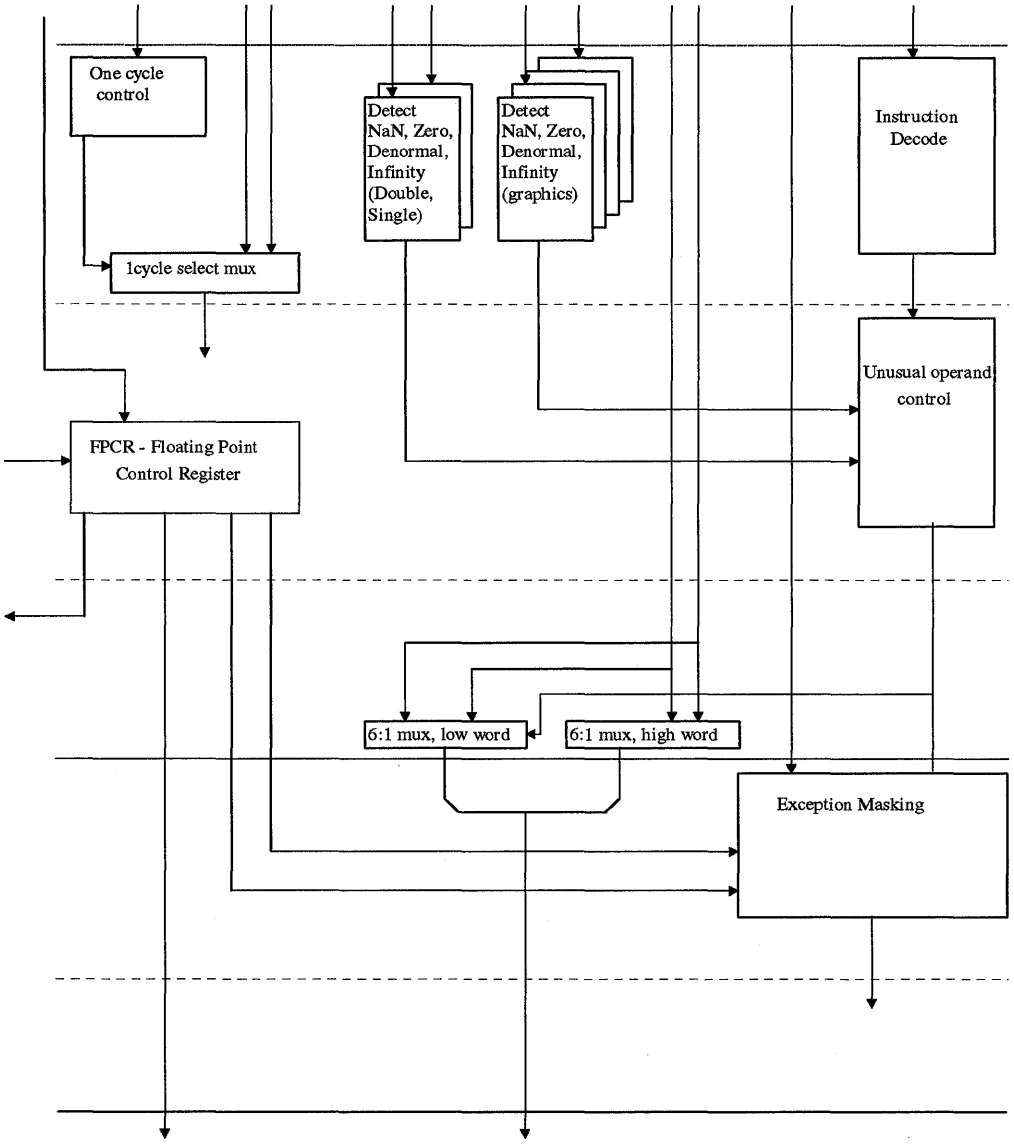
All exceptions are thrown in cycle F3 over a dedicated 8 bit bus to the Ibox (also used for requesting traps to palcode for setting FPCR exception flags).

(Exception bus coding TBD).

The denormal to zero bits must be available to the southern F_SHPs, so the northern F_SHP drives the four DNZ bits to the southern F_SHP (four wires).

The southern F_SHP has no direct access to the exception masks, so drives exceptions to its northern counterpart, where they are masked and sent to the Qbox.

Figure 8–8 F_SHP Block Diagram



8.9 Fbox Divider — F_DIV

8.9.1 Divider Description

The floating-point divider executes the following instructions:

- DIVS
- DIVT
- DIVF
- DIVG

The 21464 FDIV is the PCA57 divider. The only difference between PCA57 and the 21464 is that the 21464 inserts the divide result in the add pipe before the round adder. This costs 2 cycles of additional latency. This divider uses a split remainder algorithm which allows 6 bits/cycle and very little overhead. The timing breakdown is as follows:

Table 8–17 F_DIV Timing Sequence

Cycle	Action
0a	operand transit
0b-10a	divider array (10 passes are needed)
10b	sticky detect/add
11a	add/mux
11b	transit for result
12a/1a	mux in add pipe
13b/2b	bypass

As you can see the total latency is 14 cycles for double precision. For single precision 5 cycles are removed from the array resulting in a latency of 9. The algorithm is unique (we are patenting it) and as such isn't available in a text book. Basicly, a split remainder divider does a SRT type operation. The true remainder is never exactly known, however, the uncertainty is kept low enough and bounded to still make quotient decisions.

8.9.2 The Divider in Detail

The first phase is used to transport the A and B operands to the divider. Everything should be setup for dynamic logic for the f0b edge. Because EV8 doesn't support denormals, both the divisor and dividend fractions are correctly normalized and can be thought of as always being between 1.0 and 2.0. (1.0<= operand fraction <2.0) The logic to compute the exponent is done in the add pipe and is uninteresting.

For the next 10 cycles the actual divide occurs. The divider array consists of 6 stages and a recirculating mux. Because each stage retires one quotient bit the divider will end up computing 6*10=60 bits which is more than sufficient since only 56 are needed to correctly round. All 6 stages are substantially identical to each. The only exception to this is that two stages have a ghost latch built into them. This results in virtually zero latch overhead for the divider.

Each divider stage is composed of two different pieces. The low 49 bits of the remainder are kept in a redundant sum carry format. Carry save adders (CSA) are used for the divisor add on this portion of the remainder. All the remaining high remainder bits are kept in an exact fully encoded form. Because after each stage there is a 1 bit left shift, signals spillover out of the CSA array. These signals have the numerical value of ½, ¼ , ¼. The spillover signals skip a stage before they are incorporated into the state machines. Because they skip a stage they have a numerical value of 1, ½, ½ by the time they are incorporated. The total uncertainty of the divider is 3½. This number is calculated this way:

CSA portion	(1/8 +1/16...)*2	½
Spillover Stage -1	(½ + ¼ + ¼)	1
Spillover Stage -2	(1 + ½ + ½)	2
Total		3½

To compensate for having a rather large uncertainty of 3½, an over-redundant digit selection was used. The range for allowed digits is {-2,-1,0,1,2}. This requires the exact remainder to be bounded by +- 4*divisor.(R<(R-2*divisor)*2 -> R<4*divisor) This has the effect of changing the MUX in the CSA array from 3 to 5 inputs. Luckily 2 times the divisor is available by looking 1 bit to the right.

The dynamic state machine circuit is a 4 high N stack. One transistor is needed for the input, one for a mux to handle the various divisors, and 2 to do the spilover add. There are three signals coming from the CSA array as part of the spillover add. Since two of these are coming from the same CSA they were condensed into a single 4 bit vector. (2 vectors -> {0,1}{0,½} becomes one vector -> {0, ½, 1, 1½})

The NMOS devices in the state machine that handle the various divisor combinations were problematic for speed. There are a total of 8 different divisor combinations(1.000,1.001,1.010,1.011...) the state machine must handle. To speed up this logic, the state machine was broken into 2 pieces. One state machine handles divisors between 1.0 and 1.5 and the other divisors between 1.5 and 2.0. This reduced the size of the muxing logic which meant much less source drain capacitance.

When the divide is finished information is needed about the remainder. There are 8 different ranges that the remainder could be in:

R = -4d

-4d< R <-2d

R = -2d

-2d< R <0

R = 0

0< R <+2d

R = +2d

+2d< R <+4d

Determining these is complicated because the remainder is a little fuzzy. However, because I did a spilover add in the ghost latch at the bottom of stage 6, the round state machine under it knows the remainder with an uncertainty of only 1½ which is sufficient. Basicly the round state machine looks at the remainder and if it's definitely less than zero then it adds +2d to the remainder. If the remainder is definitely greater than zero then it adds -2d to the remainder. If the remainder is too close to zero to be sure then no add is performed, for this case the remainder is guaranteed to be bounded by -2d<R<+2d. The round state machine also takes its portion of the remainder and transforms it from a precise fully encoded form to a binary vector. This binary vector is merged with the CSA remainder to make two 56 bit vectors. These vectors are what the add, prescribed by the round state machine, are performed on. The result is two vectors, the sum of which we need to know the sign and if zero. This is done with the sign and zero detect block at the bottom of the datapath diagram. The sign detect is actually a carry lookahead adder except only the final carry out is needed. The actually sum values are never computed. The zero detect uses logic similar to the leading zero detect logic used elsewhere in the box. The zero detect is faster than the sign detect.

The one case in the table above that the hardware can not handle is the R=-4d case. This is a degenerate case that can only occur when the divisor is precisely 1.0000...0 (exponent can be anything so it is really when the divisor is a precise power of 2) When this case occurs with the hardware described above, the round state machine will add +2d to the remainder. The resulting sum will then be -2d which is nonzero and negative making the downstream logic think the -2d<R<+2d case was encountered. When doing infinity rounding the final answer will be 1 LSB greater than the correct answer. For chop rounding mode the answer would be correct, however, it would appear to be inexact. An existing zero detect in the add pipe tells the divider that the divisor is 1.0000...0 When this is known the divider forces the rounding mode to be chopped. Because dividing a number by 1.0000...0 is always exact, the rounding mode is irrelevant. The logic also masks the inexact flag.

8.9.3 Over-Redundant Digits to Binary and Rounding

The quotient digits coming from the divider array are in the form of {-2,-1,0,1,2} and must be converted to a binary number with correct rounding and normalization. I included a figure showing this logic. This conversion is done in two parts. First, {-2,-1,0,1,2} must be changed into {-2,0,2}. This conversion is done with two stages of logic and relies on the fact that I can add 1 to a digit as long as I subtract 2 from the digit to its right. Likewise, I can subtract 1 and add 2. The first stage looks at each digit and determines whether it is negative or positive and also if it is even or odd(even=-2,0,2 odd=-1,+1) The second stage then does the following (in C syntax):

```
switch(This digit)
  CASE -2: if (left is odd) then return(0);
  CASE  -1:if (right is positive AND left is odd) then return(+2);
         if (right is positive AND left is even) then return(0);
         if (right is negative AND left is odd) then return(0);
        if (right is negative AND left is even) then return(-2);
  CASE 0:  if (left is odd) then return(-2);
  CASE  1: if (right is positive AND left is odd) then return(0);
```

```
        if (right is positive AND left is even) then return(2);
        if (right is negative AND left is odd) then return(-2);
       if (right is negative AND left is even) then return(0);
CASE 2: if (left is odd) then return(0);
```

No carry propagation is required for this. The next conversion is from {-2,0,2} to {0,1,2}. This is accomplished by shifting things left 1 bit so I now have {-1,0,1}. This can be converted a standard sum carry format by a simple mapping of -1 -> 0,0 0 -> 0,1 1->1,1. This conversion is so simple no additional logic stages were needed. The ORL block in the figure generates sum carry outputs.

Because every bit needs to know the sign of the bit to right, a complication is introduced. The least significant digit (stage 6) can't be converted because it needs to know the sign of stage 1 of the next pass. This is handled by pipelining the stage 6 digit one cycle so that it gets converted with the following pass. Its sign is still needed in the current pass for the stage 5 digit.

Rounding for the divider is virtually free. Two observations about division make this possible. First, whether the quotient needs a one bit normalization shift can be determined with out ever doing a divide. Simply if the dividend fraction is greater than or equal to the divisor fraction no normalization shift will be needed. This is independent of rounding modes. Second, the infinitely precise quotient can never be exactly half way between two representable numbers. This means that the IEEE round-to-even case never occurs. Here's why these two observations are so powerful. To do round to nearest all that is necessary is to add ½ LSB to the quotient. And for infinity rounding, 1 LSB should be added to the quotient with a -1 added to the smallest possible digit. I can do all of this because I know where the LSB is. In the figure the rounding is accomplished with the CSA blocks while the quotient is still in a redundant form. The 'magic rounding vector' is computed by control logic based on the rounding mode, the datatype precision, and the normalization shift result. The total overhead for rounding in this divider is a dynamic CSA delay. Because a CSA shifts the carry one bit left I end up with a seven bit vector. This is corrected by pipelining the most significant carry until the next pass. The result of the CSA stage is two 6 bit vectors ready to be added.

One approach to generate the final binary quotient from the sum carry vectors would have been to accumulate the vectors until the divide finished. A 52 bit carry propagate add would then be performed to yield the binary quotient. This method requires building a fast adder in the datapath. The method PCA employed is different. The add is performed serially six bits at a time. This removed the fast adder from the datapath. It's also faster to build a 6 bit adder than a 52 bit.

The hardware to do the serial add is located with the over-redundant logic in the control section. Basicly 2 six bit adders were built. These two adders compute the sum of the sum carry vectors with a carryin of 0 and of 1. In addition, I detect the case where the sum of the two vectors would result in a carry out. This is the generate term. The prop signal is asserted when the carryout of the block is equal to the carryin. The two 6 bit sums (quo<5:0>,quo_plus_one<5:0>) are then routed to the datapath. These are muxed into two registers(Q0,Q1). After the first pass the 6 most significant bits of the Q0,Q1 registers receive the sums. On the second pass, the six less significant bits get loaded. This continues until the divide finishes and both registers have been loaded. The Q0 register gets loaded with the quo<5:0> signal and the Q1 with quo_plus_one<5:0> signal. The loading mechanism on the diagram is accomplished with the M1,M2 muxes.

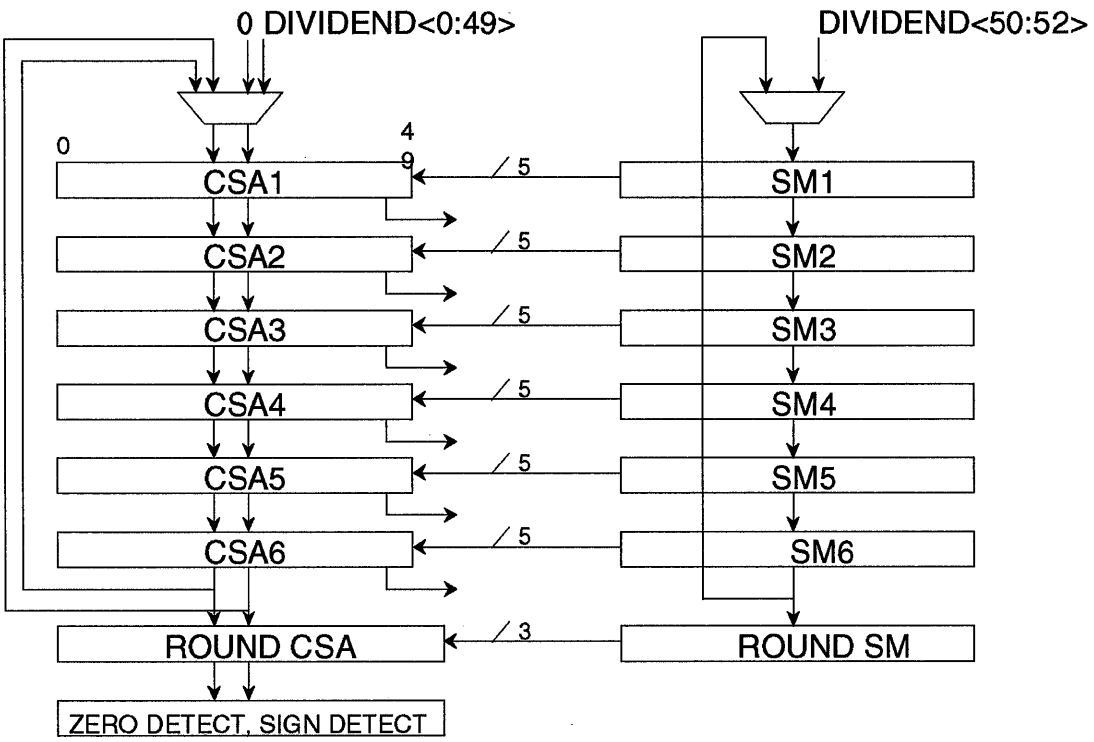
You'll notice that once the bits of the register are loaded they are recirculated with some extra muxes(M3,M4) in the path. These muxes propagate carries across bits and work this way:

If the current block doesn't have PROP or GENERATE asserted then the more significant bits already in the registers can never get a carry from this block. For this case I force Q0=Q0 and Q1=Q0. For the case where this block does assert GENERATE then I force Q0=Q1 and Q1=Q1. The last case is where the block assert PROP then I don't know whether the more significant bits already loaded will receive a carry because it determine by a the carry out of a future block. For this case I force Q0=Q0 and Q1=Q1. This keeps the status quo until I eventually encounter a block that I definitely know the carryout status. I switch the entire register with the muxes. The high bits that have their carryin determined are unaffected by this muxing action because for them Q0 is equal to Q1. The bits of the current block are also unaffected because I placed the loading mux (M1,M2) after M3 and M4. The bits to the right of the current block are don't cares because I will eventually overwrite their values with a load operation. As you can see a fast adder was replaced with 2 extra lactched and 2 extra muxes plus a 6 bit adder in the control section.

A good way to think about the quotient registers is that Q1=Q0+k. The infinitely precise result is always bounded between these two registers. Every pass through the divider increases the precision of Q0 and Q1 by 6 bits. Put another way, for every pass through the divider, k gets reduced by 64(2^6). By the time the divide is finished k is less than one LSB. The M0 mux at the top of the diagram is used to pick whether the final rounded quotient should the Q0 or Q1 register. The M5 and M6 muxes perform the one bit right normalization shift if necessary.

Back in the control section all of the signals that had to be pipelined one pass are now needed for rounding since there won't be another pass. These are fed into a PLA along with the rounding mode in effect and the remainder add selection from the round state machine. The output of the PLA is whether Q0 or Q1 should be used for three different cases. The three cases are if the final remainder is less than zero, greater than zero, or plain zero. The result of the sign and zero detect in the datapath swing this 3 to 1 mux which then drives the final M0 mux at the top of the quotient registers. This result is then written into the register file and forward for use the next phase.

Figure 8–9 F_DIV Block Diagram



8.10 Fbox Square-Root Unit — F_SQR

The F_SQR unit is responsible for computing the square root of the fraction of the both VAX and IEEE SQRT instructions. The square root unit does not have an exponent processor. It receives the input operand from the MUL unit and returns the result to the divide unit. The divider unit selects either the divide result or if no divide result need to be transferred the square root result. This result is sent to the F_AP2 pipeline for rounding. The square root unit computes the sticky bit for rounding and sends it along with the result. Since the square unit uses the F_AP2 pipeline for rounding, a bubble needs to be inserted in the F_AP2 . For this purpose the square unit sequencer sends a 'square root done' signal to the Qbox <tbs> cycles ahead so that Qbox stops issuing a new instruction to the pipeline. In addtion, it is possible to have a divide and a square collide for using the F_AP2. To prevent this square root receives a signal from the divide unit which is used to delay the bubble request.

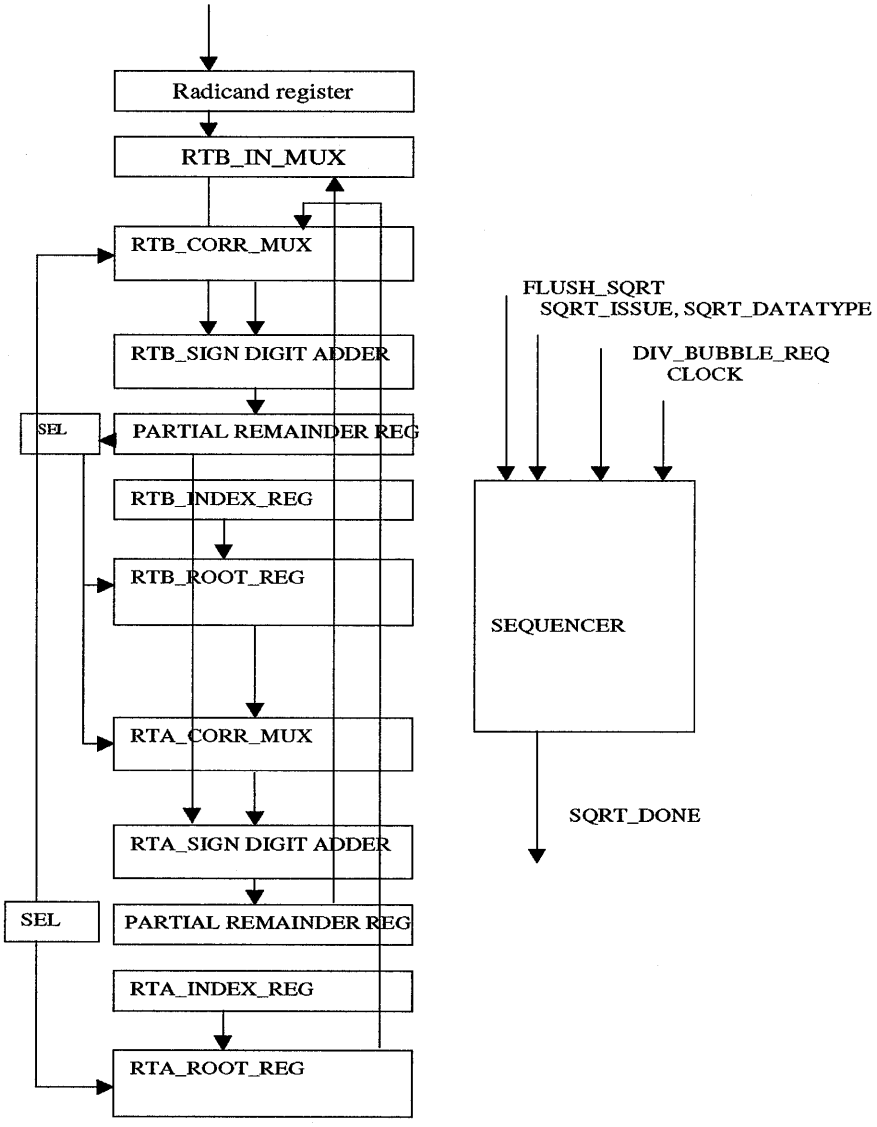
There is no exception checking in the square root unit. The square root unit assumes its operand is a non zero operand and injects the hidden bit. The input exceptions including the zero case is handled by the F_SHP unit. It is possible to abort the square root unit during a square root by asserting the flush signal.

Operation

The square unit uses a SRT type algorithm and computes two root bits per cycle. As shown in the block diagram, the square root unit consists of two identical cascaded sections, one section per bit, and a sequencer. The fraction part consists a RTB_IN _MUX that selects in the first cycle the input operand and in subsequent cycles the second

row's partial remainder . The RTB_CORR_MUX provides the correction to be added to the partila remainder. The select controls are based on the previous remainders range. The output of the correction term and the partial remainder are added in the sign digit plus binary adder to compute the redundant new partial remainder. The index regester is a shift register that keep track the insertion point of the root bits. The root register and its logic serves two purposes: it saves the new root bits and converts the root into a binary value so that it can be used for the next iteration. The sequencer depending on the data type sequences the operand input, bubble requests, and result transfers.

Figure 8–10 F_SQR Block Diagram



8.11 Fbox Graphics Pipeline

A graphics instruction set has been added to the Alpha architecture(ECO 118). The following is a brief summary of the ECO for ready reference. The next section provides a brief description of the implementation.

The Fbox implements the new paired single precision instruction set. These paired SP instructions are intended to accelerate the front end of the 3D graphics pipeline - object physics, geometry transformations, clipping, and lighting calculations. The proposed instruction set also includes several instructions that aid in the calculations involving complex numbers. They use now popular, single instruction multidata implementation. Two single precision operands are packed into a 64-bit register and each instruction operates on two sets of operands thus doubling the performance of these operations. They use the existing floating-point register file. The graphics ISA is general enough for many Single precision applications to gain significant performance improvement.

There are 36 new instructions in this proposal. The proposal is to use one new opcode (07 hex) for the graphics instructions. Since the graphics ISA uses only paired SP/LW integer, we will use the source datatype (2 bits) field to expand the function field to 6 bits. All other bits of the FP operate instruction remain the same.

8.11.1 Paired SP Floating-point Operate Instruction Format

Table 8–18 Paired SP Floating-point Operate Instruction Format

Bits:	<31:26>	<25:21>	<20:16>	<15:13>	<12:11>	<10:5>	<4:0>
Contents:	Opcode	Fa	Fb	Trp	Rnd	Fnc	Fc

Bits (10:9) used to be SRC field. Now they are part of the FNC field.

The two data types used by the new instructions are shown below:

8.11.2 Register and Memory Formats

Table 8–19 Paired Single-Precision

Operand High			Operand Low		
<63>	<62:54>	<53:32>	<31>	<30:23>	<22:0>
Sign	Exponent	Fraction	Sign	Exponent	Fraction

8.11.3 Rounding Modes

All four IEEE rounding modes are supported. For PARCPLx and PARSQRTx instructions, only chopped rounding mode is available and the round mode bits are ignored.

8.11.4 Exceptions

All exceptions as defined by the IEEE standard are generated individually on each half and the exceptions are ORed together to report. It is possible to get two different exceptions. The results written for each half follow the existing rules specified by the Alpha Architecture. Note that all 64 bits of the register are always written. It is possible to get no exception result on one half and an exception result on the other. The FPCR status flags are updated per the existing rules for the regular floating-point instructions. The regular floating-point trap control mechanism is also used for the graphics instruction set.

8.11.5 Paired Single-Precision Instructions

Table 8–20 lists the paired single-precision instructions. In the table, || means register concatenation.

Table 8–20 Paired Single-Precision Instructions

Instruction	Opcode	Operation
PADD fa,fb,fc	07.10	fcH ← faH + fbH,fcL ← faL + fbL
PARCPH fb,fc	07.09	fcH ← 1/fbH, fcL ← 0
PARCPL fb,fc	07.0A	fcH ← 1/fbH, fcL ← 1/fbL NOTE 1
PARCPLL fb,fc	07.0A	fcH ← 0, fcL ← 1/fbL
PARSQRT fb,fc	07.0C	fcH ← 1/SQRT(fbH), fcL ←1/SQRT(fbL)
PARSQRTH fb,fc	07.0D	fcH ← 1/SQRT(fbH), fcL ←0
PARSQRTL fb,fc	07.0E	fcH ← 0, fcL ←1/SQRT(fbL) NOTE 1
PCADD fa,fb,fc	07.11	fcH ← faH + fbL,fcL ← faL + fbH

PCMPEQ fa,fb,fc	07.28	IF (faH .xx. fbH) THEN fcH ← IEEE 1.0 ELSE fcH ← 0 (true zero)
		IF (faL .xx. fbL) THEN fcL ← IEEE 1.0 ELSE fcL ← 0 (true zero)
		NOTE 3,4

PCMPLE fa,fb,fc	07.2D	IF (faH .xx. fbH) THEN fcH ← IEEE 1.0 ELSE fcH ← 0 (true zero)
		IF (faL .xx. fbL) THEN fcL ← IEEE 1.0 ELSE fcL ← 0 (true zero)
		NOTE 3,4

PCMPLT fa,fb,fc	07.2C	IF (faH .xx. fbH) THEN fcH ← IEEE 1.0 ELSE fcH ← 0 (true zero)
		IF (faL .xx. fbL) THEN fcL ← IEEE 1.0 ELSE fcL ← 0 (true zero)
		NOTE 3,4

PCMPNEQ fa,fb,fc	07.29	IF (faH .xx. fbH) THEN fcH ← IEEE 1.0 ELSE fcH ← 0 (true zero)
		IF (faL .xx. fbL) THEN fcL ← IEEE 1.0 ELSE fcL ← 0 (true zero)
		NOTE 3,4

PCMPUN fa,fb,fc	07.2A	IF (faH .xx. fbH) THEN fcH ← IEEE 1.0 ELSE fcH ← 0 (true zero)
		IF (faL .xx. fbL) THEN fcL ← IEEE 1.0 ELSE fcL ← 0 (true zero)
		NOTE 3,4

PCPYS fa,fb,fc	07.20	fcH ← faH<s> || fbH<exp.frac>,fcL ← faL<s> || fbL<exp.frac> NOTE 7
PCPYSE fa,fb,fc	07.22	fcH ← faH<s.exp> || fbH<frac>, fcL ← faL<s.exp> || fbL<frac> NOTE 7
PCPYSN fa,fb,fc	07.21	fcH ← NOT.faH<s> || fbH<exp.frac>, fcL ← NOT.faL<s> || fbL<exp.frac> NOTE 7
PCVTFI fb,fc	07.39	FcH ← cvt.integer(fbH), FcL ← cvt.integer(fbL
PCVTSP fa,fb,fc	07.30	fcH ← {CVT 64b SP to 32b SP of fa}, fcL ← (CVT 64b SP to 32b SP of fb) NOTE 5,7
PEXTH fb,fc	07.3C	fc ← {CVT 32b SP of fbH to 64 SP)
PEXTL fb,fc	07.3A	fc ← {CVT 32b SP of fbL to 64 SP) NOTE 6.7
PFMAX fa,fb,fc	07.2E	fcH ← MAX(faH,fbH), fcL ←MAX(faL, fbL)
PFMIN fa,fb,fc	07.2F	fcH ← MIN(faH,fbH), fcL ←MIN(faL,fbL)

Table 8–20 Paired Single-Precision Instructions (Continued)

PHADD fa,fb,fc	07.12	fcH ← faH + faL, fcL ← fbH + fbL	
PHSUB fa,fb,fc	07.16	fcH ← faH - faL, fcL ← fbH - fbL	
PHSUBR fa,fb,fc	07.1F	fcH ← faL - faH, fcL ← fbL - fbH	
PMOVHH fa,fb,fc	07.1B	fcH ‖ fcL ← faH ‖ fbH	
PMOVHL fa,fb,fc	07.1A	fcH ‖ fcL ← faH ‖ fbL	
PMOVLH fa,fb,fc	07.19	fcH ‖ fcL ← faL ‖ fbH	
PMOVLL fa,fb,fc	07.18	fcH ‖ fcL ← faL ‖ fbL	NOTE 2,7
PMUL fa,fb,fc	07.00	fcH ← faH * fbH, fcL ← faL * fbL	
PMULH fa,fb,fc	07.02	fcH ← faH * fbH, fcL ← faL * fbH	
PMULHN fa,fb,fc	07.06	fcH ← -(faH * fbH), fcL ←faL * fbH	NOTE 8
PMULL fa,fb,fc	07.01	fcH ← faH * fbL, fcL ← faL * fbL	
PMULLN fa,fb,fc	07.05	fcH ← faH * fbL, fcL ← -(faL * fbL)	NOTE 8
PSUB fa,fb,fc	07.14	fcH ← faH - fbH,fcL ← faL - fbL	
PSUBC fa,fb,fc	07.15	fcH ← faH - fbL,fcL ← faL - fbH	

Notes from Table 8–20:

1. The result for these two instructions is accurate to a minimum of 14 bits of precision only. For PARCPLx and PARSQRTx instructions, only chopped rounding mode is available and the round mode bits are ignored. For PARCPLH, PARCPLL, PARSQRTH, and PARSQRTL instructions, no checking is done on the unused operand and no exceptions are generated.
2. With fb = f31, the lower half of the result can be cleared. Similarly high half can be cleared with fa = f31. Operands can be swapped or duplicated with fa = fb.
3. There are no separate graphics branch instructions. With these compares one needs to use the regular floating-point branches. There was a suggestion for a specialized Branch for clip test - branch when either is negative. This can be accomplished as PCMPLT fa, f31, fc; test faH/faL LT 0, fc = 0 only if both positive FBREQ fc,X
4. Paired single-precision compares write the destination if condition is TRUE with a value of IEEE 1.0, if the condition is FALSE, a true zero. This behavior is different from normal floating-point compares. A combination of CMP and multiply instructions can be used to conditionally clear a register, for normal operands.

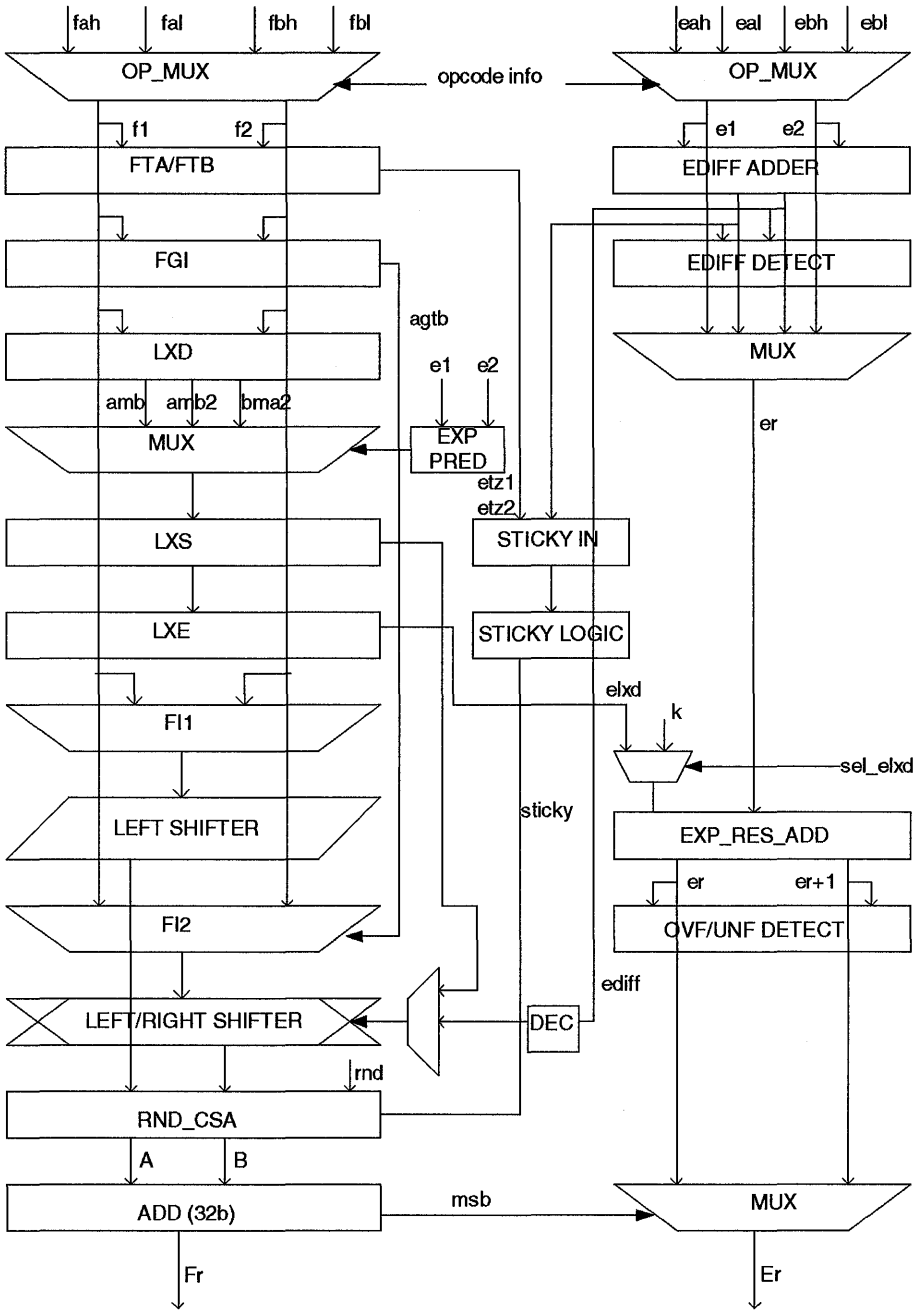
   PCMPxx f1, f2,f3 ;Each half of f3 gets 1.0 if true, 0.0 otherwise

   PMUL f1,f3,f4 ;Each half of f4 gets f1(H/L) if condition is true, 0 otherwise.
5. PCVTSP instruction converts two 64 bit single-precision floating- point operands (with 11 bit exponents) to two 32 bit SP (with 8 bit exponents) and packs them as paired single-precision operands. Dropping three bits (61:59) and ignoring (28:0) does the conversion. No checking is done and there are no arithmetic exceptions. {The operation is similar to a STS instruction.}
6. PEXTx instruction converts a 32 bit SP in paired format to a 64-bit SP number. The operation uses MAP_S exponent mapping (see SRM Chapter 2 Table 2-2) similar to a LDS instruction. No checking of bits is done.

7. PMOVxy, PCPYSx, PCVTSP, PEXTx are bit manipulation instructions and generate no arithmetic exceptions.
8. For this instruction the product is rounded first and then the appropriate half (specified by the instruction) of the result is negated.

The graphics instruction set is implemented in two pipelines in the Fbox. The PMULxx, PARCPLx, and PARSQRTx instructions are implemented in the F_MUL pipeline. All the other instructions are implemented in a separate pipeline F_GAD. The implementation details for the PMULxx, PARCPLx, and PARSQRTx instructions are described under the F_MUL section. In the next section the implementation details for the F_GAD are given.

Figure 8–11 F_GAD Block Diagram for One-Half of the Pair



8.11.5.1 Graphics Add Pipeline: F_GAD

A block diagram of the graphics add pipeline is shown in Figure 8–11. The F_GAD pipeline has two identical units – one for each half of the paired data. In the block diagram only the high half (F_GHx) of the F_GAD pipeline is shown. The low half (F_GLx) is identical to F_GHx and it processes operands in the low half (31:0) of the pair. The F_GAD has been leveraged from the 21264 Fbox.

The F_GAD is a 4 cycle latency pipeline. The input operands to the pipeline are driven by the Multiplier pipeline instead of the interface section, to minimize the length of the low swing operand wires. The input operands for the GAD pipeline are single precision floating point operands with 23b fractions and 8b exponents . Hence at the beginning of the pipeline the fraction datapath is narrower. To accommodate the convert from floating point to 32b integer instructions the shifter and the final adder are 32b wide.

The implementation of add/sub instructions in the GAD pipeline is slightly different compared to the main Fbox add pipeline. Referring to the section Fbox add pipeline, in the 'near domain', instead of subtracting the smaller operand at the very beginning of the pipe (step 2N) using an adder, the two operands are first normalized (left shifted) removing the bits that produce the leading zeros. This requires an additional left shifter in the data path. Once the two operands are normalized, the subtraction is done in the final adder/rounder. Since there is only one adder in the path, for the ediff = 0 case, the final subtraction must produce a positive result to conform to the sign-magnitude representation of the result. To ensure this, the smaller operand is to be always subtracted from the other. This is accomplished by first comparing the two fraction operands (the exponents are same) to determine the smaller operand.

In addition to the regular floating-point instructions, the GAD pipeline needs to implement several variations of add/sub instructions, Moves, floating MAX/MIN instructions. The add/sub instructions and the MOVE instructions require selection of different halves of the two operands. The FMAX/FMIN instructions are similar to the compare instruction.

The operation and the implementation of the GAD pipeline are described below using the high half of the pipeline as shown in the block diagram. The description applies equally to the low half also.

8.11.5.2 Fraction Datapath

8.11.5.2.1 OP_MUX

The OP_MUX selects the two operands for each half of the pipeline from the possible 4 operands- the two pairs from each source. For the high half of the pipeline the final operands are f1h, f2h on the fraction side and e1h, e2h for the exponents. For the normal PADD/PSUB instructions t f1h = fah and f2h = fbh, for PADDC/PSUBC instructions, f1h = fah, f2h = fbl; for PHADD/PHSUB – f1h = fah, f2h = fal and so on. For the PMOVxx instructions the OP_MUX selects the operand indicated by the instruction on f2h and zero on the f1h and similarly on the exponent and sign parts of the operands. This allows passing the f1h on high half and f11 on the low half before they are packed into a 64 bit result. For the PCVTSP instruction, three bits from the exponent parts of Fa and Fb registers are dropped and fraction, sign and exponent are selected onto f2h/f2l, s2h/s2l, e2h/e2l respectively. The output of the mux is available at the end of F0A.

8.11.5.2.2 FTA, FTB

The FTA, FTB count the number of trailing zeros in the two operands f1h and f2h for calculating the sticky bit. The results of this block are the two signals etz1 and etz2. For effective add/subtract operations, since it is not known until the exponent subtract is done, which of the operands is smaller, trailing zero for both operands are counted. The sign of the exponent difference indicates the smaller operand and this is used in calcu-

lating the sticky bit. If ediff is the exponent difference, which indicates the right alignment shift, the sticky bit can be calculated by comparing the ediff and 'etz'- if etz < ediff then a '1' must have shifted out and hence sticky is 1.

8.11.5.2.3 FGT

The FGT block compares the two fractions f1h and f2h and produces a signal f1h > f2h ('agtb'). This signal is used for effective subtract with ediff =0 case, PCMP, and PFMAX/PFMIN instructions. During an effective subtract operation when the efiff = 0, the samller operand has to be subtracted from the other operand. This signal is used to correctly compliment the smaller operand in the FI2 Mux. For the compare instructions this signal is used to set the indicated condition is TRUE or FALSE based on the fraction compare. During the PFMAX/PFMIN operations this signal is used to coorectly pick the correct fraction to be passed to the result in the FI2 MUX.

8.11.5.2.4 LXD and EXP PRED

The LXD block calculates the leading 1/0 using the input operands f1h and f2h for exponent difference <2 cases. It computes the LXD for three cases: ediff = 0 (f1h-f2h/ f2h-f1h) and ediff = 1 (f1h – f2h/2), and ediff = -1 (f2h-f1h/2).

The EXP PRED block at same time examines the two least significant bits of the two exponents and predicts the exponent difference using the logic presented in Table II. One of the three possible LXD vectors is selected based on this prediction. This result vector contains a '1' for all possible positions of the leading 1 where the very first '1' in the vector is the correct position. To be able to drive the left shifter for normalization, this vector needs to be stripped off of the unnecessary 1s.

8.11.5.2.5 LXS and LXE

The LXS block strips the extraneous '1's after the leading 1 from the LXD vector from the previous stage. This vector, containing a '1' in the leading 1 position and zeros everywhere else, is wired to the left shift control to shift the input operands left. The LXS vector is also encoded in the LXE block so that the left shift amount can be subtracted from the exponent result.

8.11.5.2.6 FI1/FI2 MUX and the LEFT/LR Shifters

The FI1/FI2 muxes select the input operands for the two shifters. The left shifter is used only during the effective subtract operation to shift the f1h operand left to remove the leading zeros before they are input to the adder in the final stage. For all other operations the left shifter simply passes the input operand. The L/R shifter which is capable of shifting the operand left or right is used in all operations. Since conversion from floating to integer operationrequires a 32b result, the L/R shifter is 32b wide, whereas the left shifter is only 24b wide. The left shifter takes control directly from the LXS

vector. The L/R shifter is controlled by LXS and by the ediff from the exponent data path. The following table lists the operand selected by the FI1, FI2 muxes, control and the conditions. The operands are shown as frac_1 or fract_2 etc.

Table 8–21 FI1/FI2 Shifter Operand/Control Selection

Oper.	Condition	Control LSHF	LRSHF	Left Shifter Input	High Input	Low Input
PADDx						
PSUBx						
Effective add:						
	exp_1 >= exp_2	L(0)	R(ediff)	frac_1	0	frac_2
	exp_1 < exp_2	L(0)	R(ediff)	frac_2	0	frac_1
Effective sub:						
	exp_1 >= exp_2+1	L(0)	R(ediff)	frac_1 * 2	0	frac_2 * 2
	exp_1+1 < exp_2	L(0)	R(ediff)	frac_2 * 2	0	frac_1 * 2
	exp_1 == exp_2	L(lxd)	L(lxd)	frac_1 * 2	frac_2 * 2	0
	exp_1 == exp_2+1	L(lxd)	L(lxd)	frac_1 * 2	frac_2	0
	exp_1+1 == exp_2	L(lxd)	L(lxd)	frac_2 * 2	frac_1	0
PCVTFI	exp_2 - bias >= 22	L(0)	L(ediff)	0	frac_2	0
	exp_2 - bias < 22	L(0)	L(ediff)	0	0	frac_2
PCVTFF		L(0)	R(0)	0	0	frac_2
PCPYSX		L(0)	R(0)	0	0	frac_2
NaNs: where x is not a Nan						
	F1 F2					
	x NaN	L(0)	R(0)	0	0	frac_2
	NaN x	L(0)	R(0)	0	0	frac_1
	NaN NaN	L(0)	R(0)	0	0	frac_2
PCMPX		L(0)	R(0)	0	0	0
PFMAX	A GT B	L(0)	R(0)	0	frac_1	0
	B GT A	L(0)	R(0)	0	frac_2	0
PFMIN	A GT B	L(0)	R(0)	0	frac_2	0
	B GT A	L(0)	R(0)	0	frac_1	0

8.11.5.2.7 RND CSA and ADDER

The RND CSA and the final ADDER perform rounding and the final addition/subtraction. The round CSA enables rounding and add in one step. The rounding CSA combines the two fraction outputs from the shifters and a rounding constant and prepares the two inputs for the adder. The rounding constant depends on the rounding mode,

sticky bit from the sticky bit logic. The rounding constant is added in the CSA in the least significant bit positions. Since there are more than 2 inputs, a CSA in these positions enables adding them. In the high order bit positions a half adder is used.

The adder computes two results – one assuming the MSB = 0 and the other assuming that the MSB = 1. If MSB = 1, the fraction needs to be shifted down and the exponent has to be adjusted. Note that effective sub, when the ediff > 1, actually may need a 1 bit normalization. In order to avoid looking at two bits (hidden bit and the bit below it), the two input operands have been shifted left by 1 in the shifter input muxes as shown in the Table. This preshift is also used for the effective sub ediff < 2 cases also to move the 1 bit uncertainty in the LXD logic so that it can be detected as an overflow and corrected.

8.11.5.3 Exponent Data Path

The exponent data paths is simpler than the fraction data path and deals with only 8b exponent parts of the operands. The exponent section receives the exponent parts of the final selected operands from the OPD_MUX for each half of the pair.

For floating point arithmetic operation it computes the exponent results and for bit manipulating instructions it simply passes one of the exponents.

8.11.5.3.1 EDIFF ADDER

The ediff adder computes the absolute exponent difference for add/sub instructions and the length of the integer portion in the floating-point operand for the convert to integer operation. To calculate the absolute exponent difference, the ediff adder actually contains two adders – one that computes A-B and the other B-A. Based on the sign of the first adder (En), the positive result is selected and is used to drive the L/R shifter control. For the convert floating to integer operation, the bias needs to be subtracted from the input exponent to determine the integer portion of the floating-point value. This is done in the B-A adder. For PEXTx instructions the bias is subtracted in the B-A adder and DP bias is added back later to complete the conversion. For PCMPx and PFMAX/PFMIN, the A-B adder provides the comparison of exponents

8.11.5.3.2 EDIFF DETECT

The Ediff detect logic determines exponent ranges and if the exponent difference is 0,1, GTR 1, or GTR 25, etc. The exponent range is used to classify the input operands into denormals, infinities, NaNs, and zero operands. The ediff detects are used for choosing the near domain vs far domain, out of range in add/sub instructions, to determine if the exponent comparison for CMP type operation.

8.11.5.3.3 ER MUX

The ER MUX selects the intermediate result for the exponent. For add/sub instructions it picks the MAX(e1,e2), for convert type instructions the result of B-A adder, and for CPYSX instructions one of the input operand exponents. For PFMAX?PFMIN, depending on the instruction, MAX(e1,e2) or MIN(e1,e2) is selected. The intermediate result 'er' is driven to the next stage.

8.11.5.3.4 EXP_RES_ADD

The exp_res_add computes the two final results of the exponent. It computes exponent results Er and Er+1 - in case a fraction overflow occurs and the fraction has to be shifted down. The MSB from the fraction is used to determine which result. During the

effective sub operations when the ediff <2, the 'elxd' – the normalization amount has to be subtracted from the intermediate exponent. For this, the mux selects the elxd from the fraction data path. Since the LXD logic can overestimate the left shift by upto 1 bit, it is possible to shift the fraction result right by one bit The Er+1 result is used in this case. During the PEXTx instructions, er represents the the unbiased exponent result. The DP bias is selected and and is added back to the 'er'. For all other operations, a zero is added to the intermediate 'er' and the MSB from the fraction data path is used to select the final result exponent.

8.12 G_AD Control

The G_AD control pipeline reveves the opcode information during the FY cycle. In addition, it receives the rounding mode information from the F_SHP pipeline. The control pipeline (not shown in the block diagram) decodes the opcode and controls the fraction and exponent data path. The control logic also detects the various exception conditions and signals the exceptions to the F_SHP pipe to communicate to the Qbox using the trap disable bits from the FPCR.

8.12.1 Fraction Data Path

The bits in the Fbox fraction data path are numbered as follows:

Table 8–22 Fraction Data Path

Weight								4	2	1		.5	.25	...	
A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	.	B0	B1	...	B25
											Binary point				
										Hidden bit for floating-point numbers					

Bits <A10:A0> are used for Integer operands and D-type operands.

The exponent data path, E<12:0> is 13 bits wide, with E<12>representing the exponent sign bit. The pipeline also maintains a sign (N) and a zero (Z) bit for each operand.

Operand data is formatted onto the fraction and exponent data paths depending on the data type as follows:

Table 8–23 Operand Data Fraction and Exponenet Data Paths

BITS	F/S/G/T	D	Q	L
A10:A7	0	0	OP<63:60>	OP<63>R=3
A6	0	0	OP<59>	OP<62>
A5:A0	0	0	OP<58:53>	OP<58:53>
B0	1(NZ)	1(NZ)	OP<52>	OP<52>
B1:B3	OP<51:49>	OP<54:52>	OP<51:49>	OP<51:49>
B4:B52	OP<48:0>	OP<51:3>	OP<48:0>	OP<48:0>
B53(R)	0	OP<2>	0	0

Note:

- In D format, OP<1:0> are lost. OP<2> is used as R-bit for rounding in CVTDG.
- In L format, operand is sign extended.

The predicate(p) bit and sign bits are taken directly from RF<64:63>.

8.13 Sticky Bit Calculation

The sticky bit represents the aggregated information of the bits shifted out of datapath in the process of alignment. With the help of the stick bit and round bit, we know whether the infinite precision result is above midway point or below midway point in round stage and the finite precision result can be rounded with specific rounding mode.

Essentially, the sticky bit checks every bit shifted out of datapath to see if there is any '1'. If there is '1', the sticky bit is set to 1; otherwise, it's 0. The most straight-forward way is to do a zero detect on the string of bits shifted out. However, this approach needs a 128 bit datapath in the worst case.

A smarter approach is to calculate the number of trailing zeros in the operand that is going to be aligned. If the number of trailing zeros is more the the amount of right shift, we can be sure that bits lost by right shifting are all zeros. Hence it is concluded that the sticky bit is 0. Otherwise, there must be a one somewhere in the lost bits, which makes the sticky bit 1. Note that we don't care about the precise value of the lost bits. All we need to know is the relation between precise value and midway point.

For instance, the number of trailing zeros (etrz) is 6. If the right shift amount (ediff for elxd) is more than 6, then Sticky bit is 1. If ediff, or elxd, is less than or equal o 6, then the sticky bit is 0.

In physical implementation, we need to consider many more factors than the simple example. First, the R bit is in the datapath. Second, all formats are sharing a single 65 bit datapath, so we need to take their differet lengths into account. Besides, we need to handle integer and floating-point numbers differently. Third, the amount of right shift can be from ediff or elxd. In addition, the equations are affected by how we encode ediff and elxd, and the sign of them.

Table 8–24 Equations of Sticky Bit Calculation

Condition	Sticky
eff. sub & fs:	etrz - (edif+27) < 0
eff. sub & gt:	etrz - (edif -2) < 0
eff. add & fs:	etrz - (edif+28) < 0
eff. add & gt or CVTFQ & ~en:	etrz - (edif -1) < 0 right shift
cvtqf & FS:	etrz - (29-elxd) < 0
cvtqf & gt:	etrz - (-elxd) < 0
cvtts:	etrz - (edif+28) < 0
cvtfq & en:	etrz - (edif+63) < 0 left shift

In summary, these equations are composed of three elements: etrz, ediff(or elxd), and a constant to account for different situations. In physical implementations, sticky is computed with a CSA to compress the 3 numbers and a carry chain to detect the sign, which determine the value of sticky.

9

Memory Instruction Execution Unit — the Mbox

The 21464 Mbox is responsible for executing Alpha memory reference instructions, including integer and floating point load and store, memory barrier, prefetch, write-hint, load-locked, and store-conditional.

The Mbox processes several instructions per cycle, out of order. Each cycle, the Mbox can accept as many as three load instructions, and as many as two store instructions, for a maximum of four operations. Unlike the other function units, it is responsible for keeping track of memory reference instructions which have issued but not retired, and for ensuring that the final effect of memory reference instructions is equivalent to sequential execution of the thread, within the Alpha SRM definition of equivalence. The Mbox also receives fill data from the Cbox and, to maintain cache coherence, processes probes that the Cbox receives from the rest of the system.

The Mbox has four instruction ports to handle loads, stores and prefetches. Three of these ports can support returning data from the Mbox. Thus, the maximum number of loads able to be issued to the Mbox each cycle is three. Two of input ports can perform loads and prefetches; one can perform Loads, Stores and Prefetches, and one port performs only Stores. There are two data input busses, each is associated with a Store port.

The major components of the Mbox are:

Table 9–1 Mbox Major Components

Components	Description
Dcache	64KB of data storage, with a write-allocate, write-through write-policy
Dtags	1K entries of tag storage, arranged as 2-way set-associative with 4 read ports and 1 write port
Load Queue	64-entry queue that holds issued, but not-retired Load addresses. Handles Load ordering traps and re-issuing of Loads
Merge Buffer	16-entry buffer that accumulates Store data before writing it into the Dcache and Cbox.
Pre-MAF	Sixteen-entry buffer that holds the addresses of loads that have missed in the Dcache and need further activity in the Cbox.
Store Queue	64-entry queue that holds Store addresses & data before Stores have retired. Used to satisfy Load requests to addresses with uncompleted Stores
Translation Buffers	128-entry, fully-associative with 4 read ports to perform the virtual-to-physical address transactions

9.1 Major Inputs & Outputs

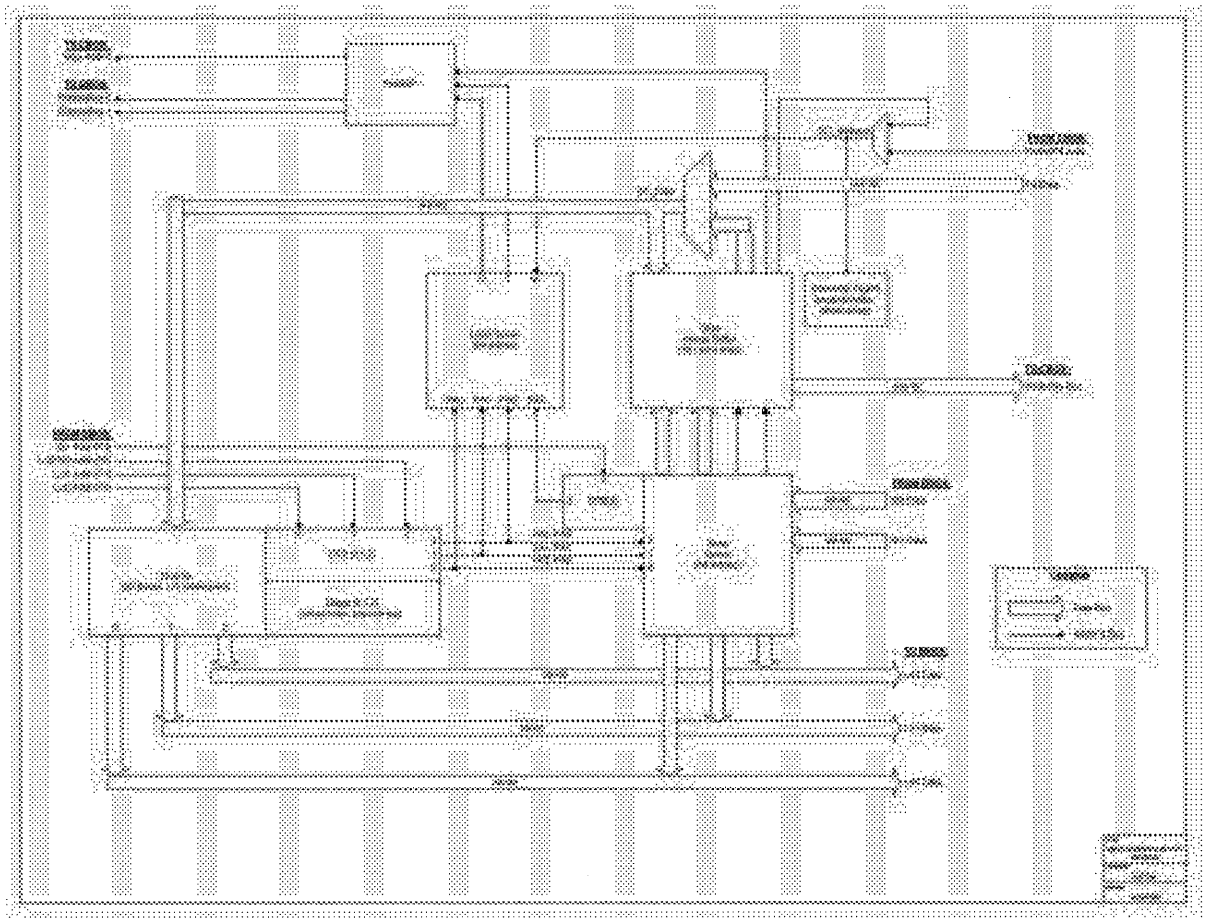
9.1.1 Inputs

- TBS

9.1.2 Outputs

- TBS

Figure 9–1 Address and Data Path



9.2 Dcache

The Data Cache, or Dcache, is a 64K-byte 2-way associative onchip data storage. The data is organized in 64-byte blocks, divided into 32 4-byte banks (address bits 5-2 indicate which bank contains which longword). Each bank can accept one read and one write a cycle. The reads share physical resources for accessing the Dcache. Three Address ports are input to the Dcache from the Ebox; three Data ports are output back to the Ebox.

There are three data ports to the Ebox upon which read data are transferred from the Dcache. Conflicts can arise if Loads request data from the same Dcache bank (address bits 5-2 of each Load address are identical). If this conflict occurs, a load is blocked from completing this cycle, and is retried again as soon as possible.

Stores write into the Dcache during the second half of each cycle.

The virtual address field of bits 14-6 indexes into the Dcache. The two quadwords of data (one from each way) that are addressed by this bit field and by bits 5 through 3 begin driving their data. At the output of the Dcache, a selector drives only one of the two quadwords out onto the Load data bus. This selection is provided by the Tag Store, which uses the rest of the upper address bits to decide which quadword is actually being requested.

The Dcache also has data parity bits stored with the block data.

9.3 Dtags

The Dtags, or tag stores, are address arrays organized similarly to the Dcache. There are four Dtag arrays, one for every load port to the Mbox, and one for the back-end operations (Stores, Fills, Invalidates, Victims). Each Dtag array is maintained as an exact copy of the others. Each holds 1K tag entries; every tag entry corresponds to a physical block of Dcache. Nine bits of virtual address (bits 14-6) are used to index a set of two tag store entries. The Dtag dedicated to the back-end has an extra capability to access eight tag store entries (when bits 14 & 13 are not used) so that Invalidates (which use only physical addresses) can access the Dcache.

Each tag entry has a physical tag field (bits 47-13), control bits that designate the cache state (valid, owned, and so forth), and it will likely have a parity bit across the tag field. Each set of two tag entries will also have an allocation bit. This bit signifies which entry way is the next to be allocated.

The upper part of the physical address (bits 47-13) is compared to the tags being stored at each of the two accessed entries. If a tag matches the address, and the cache entry is determined to be in the proper state, a hit signal is generated for the operation. This hit information is used to generate the hit signal that is sent back to the Qbox. The hit information is also used to control which of the selected quadwords of Dcache data are to drive the Load data bus.

9.4 Load Queue

The Load Queue, abbreviated as LQ, holds unretired Loads that have been issued to the Mbox. The LQ entries are allocated to the Loads in program order, by thread. The LQ is used to maintain ordering of Loads related to Stores and Memory Barriers. It also is used to re-issue Loads from the Mbox itself, when a Load cannot complete successfully when it is first issued from the Qbox.

The LQ has 64 entries and is partitioned equally between threads at run-time. Thus, when a single thread is run ning, all 64 entries are allocated for that thread; if two threads are running, each is allocated 32 entries (the LQ being separated at its mid-point). When four threads are running, each thread is allocated 16 entries.

Each LQ entry contains bits 14–13 of the virtual address, the physical address, opcode, INum, Qbox information, a done bit, and retry bits. The LQ is allocated in program order (by thread) by the Qbox, which assigns load-serial (LNum) numbers for all Loads during the Mapstage.

If the Load completes successfully, it is marked in the LQ during the M1 stage. Otherwise, the Load is marked to be retried. The Load may retry due to a cache miss, a bank conflict or may be a class of Load that can only complete at retirement (i.e., I/O Loads which cannot be done speculatively).

Every cycle, the retry logic in the LQ scans all the entries and finds the oldest ready entry (in a given thread). Readiness is defined differently for each type of retry, but generally refers to when the Load can make further progress. The retry logic then sends the INum of the Load and other stored information to the Qbox. Retry candidates are chosen from different threads in a round-robin fashion.

The LQ facilitates speculative execution of Loads by allowing Stores to check if a Load younger than it, in program order, may have completed (i.e., the Load returned data before the correct Store data had been sent to the Mbox). When the Store address operation dispatches from the Qbox, it checks the LQ. If a match is found, the oldest Load that matches the Store address is forced to trap (i.e., the Load INum is read out and sent to the Qbox). Note that this check is relevant only for Loads and Stores within the same thread.

The LQ also facilitates speculative execution of Loads past Memory Barriers. This is made possible by allowing Stores from the Merge Buffer to check the LQ for possible address matches. In this case, a Store needs to trap a Load from another thread. Note that in this case, up to three Loads can match the Store's address and signal a trap simultaneously.

LQ entries are deallocated once the Load is past the retire point. The Qbox sends the LQ an INum for each thread that corresponds to the youngest operation that is being retired. All LQ entries that are older than this INum are marked as being deallocated.

The LQ drives a signal to the Qbox every cycle that specifies the youngest Load that may issue out of the Qbox. This signal is based on the youngest Load that is being deallocated and the number of available entries in the LQ.

9.5 Merge Buffer

The Merge Buffer, abbreviated as MGB, holds Stores from the store queue after they have retired and before they have updated the Dcache. The MGB helps to accumulate Stores to the same cache block by allowing data from multiple Stores to fill up the same MGB entry. This accumulation reduces the number of unique write operations needed to send to the Dcache, thus reducing the Merge Buffer's bandwidth requirements on the back-end bus.

The Merge Buffer has 16 entries. Two input ports, each providing a data and address path, are driven from the SQA and the SQD, and up to two entries can be allocated each cycle. The addresses compare against the existing MGB entries. Each entry contains a physical address, 64 bytes of data, and a 64-bit mask to indicate which bytes of the block have been written into the MGB.

If a Store address matches one already in the Merge Buffer, the Store's data are loaded into that entry, merging with the data there. The entry's byte mask is updated to reflect the new bytes that are being written.

The Merge Buffer arbitrates with the Cbox for the DC_Data Bus. Once the MGB wins arbitration, it looks up the Dtags. If the cache block is writeable, the data block is updated in the Dcache and the Dtags are updated accordingly.

If the block doesn't exist in the Dcache, a miss request is made to the pre-MAF. If all 64 bytes if the MGB entry have been written, a block ownership request is launched; otherwise a block fill is initiated.

Fills may need to merge data from the Merge Buffer before it updates the Dcache. Hence, Fill addresses search the Merge Buffer for a match at the same time that the Dtags are looked up. If a match is found, the Merge Buffer drives the valid bytes onto the DC_Data Bus, effectively merging with the fill data.

Probes that hit on the Dcache also need to check the MGB. Probes check the Merge Buffer at the same time that they check the Dtags. If a probe hits on a Merge Buffer entry, the Merge Buffer reads out the valid data bytes and drives them to the Cbox.

9.6 Pre-MAF

TBS

9.7 Store Queue (SQA and SQD)

The store queue acts as a 64-entry reorder buffer for all Store instructions and a number of cache movement instructions. The store queue is comprised of two parts: the SQA that stores all information except the store's data, and the SQD that stores the data and a duplicate virtual address.The store queue holds information for a store from the time it issues until that store can be written into either the Dcache or the Merge Buffer. This movement of information and simultaneous store queue deallocation cannot take place until the processor retires (or commits to) the store instruction.

While a store instruction is resident in the store queue, it will attempt to supply data to appropriate younger loads in the same thread. Issuing load addresses compare against STA contents. If all of the requesting data are in one STD entry, the STD will override the Dcache's drive of the Load data bus and drive the data itself.

The store queue input ports can accept two stores per cycle. In addition, the store queue can process up to three loads (providing data) and two deallocations per cycle.

9.8 Translation Buffers

The Translation Buffers, or DTBs, are used to perform fast virtual address-to-physical address translation. There are four copies of the DTB, one for each Mbox input port. The DTBs each have 128 entries and are fully associative. Virtual addresses are sent to the DTBs when Loads and Stores are issued to the Mbox. Each address checks for a comparison against all of the DTB entries to see if the translation for its virtual page number (bits 51-13) has already been loaded into the DTB. If there is a match, the DTB drives back the physical page number (bits 47-13). These address bits are the ones used to compare against the Dtag address. It is also loaded into the appropriate queue (LQ or

SQ) and driven down to the pre-MAF in case the operation requires miss processing. If the virtual address bits do not match anything in the DTB, a TB Miss Trap is indicated. This trap kicks off a PALcode routine that reaches into the operating system's page tables and generates the appropriate physical address for the current operation. The physical page number and the page's associated control bits are all loaded into the DTB via an Internal Processor Register (IPR) write. When the PALcode routine exits, control is returned to the program at the instruction that caused the TB Miss Trap. When the operation makes it to the Mbox next time, the translated address will be in the DTBs.

The allocation policy is round-robin.

9.9 Back End Bus

TBS

9.10 Operations

9.10.1 Read Requests

Loads typically issue out of the Qbox (in the Q stage of the pipeline). After reading the register file, it computes the Load address (in the E stage). The full virtual address is sent to the Mbox at the beginning of M0. The Dtags are looked up in M0, using the index bits (virtual address bits 14-6), while the DTB transates the virtual address to the physical address. The translated (physical) address from the DTB is compared with the tag address from the Dtags. The Dtags are 2-way set associative and the tag comparison is done on the two tags simultaneously. Only one of these tags can compare with a given address. Loads write into their assigned LQ entry starting in the M0 stage.

Both indexed blocks of the Dcache are retrieved at the same time as the Dtags and DTB. If the block is present in the Dcache, the hit indication is used to drive only one of the retrieved quadwords from the Dcache.

The Load compares against SQ addresses in parallel with the Dcache access. If the SQ indicates an address match, the Dcache drive is inhibited and the SQ drives the data. If the Load is not satisfied by either the Dcache or the SQ, a miss request is launched. The Load is retried from the LQ once the missing block is fetched by the Cbox.

If a Load that is dispatched on port 2 is to the same bank as a Load on port 1, the Load on port 2 is marked as having a bank conflict and must be retried.

If a Load is to I/O space (physical address bit 47 equals 1), then the Load cannot dispatch through the memory system until it is known for certain that the Load will not abort (by an exception or a trap). Once a Load address is found to be in I/O space, after the DTB lookup, the Load is retried and completed only when the Qbox signals thatthe Load is next to be retired.

After a Load has completed successfully (e.g., sent data to the Ebox) but before it is retired (and deallocated from the LQ), it may receive a trap signal. In such cases, the Load INum is sent to the Ibox.

9.10.2 Prefetches

TBS

9.10.3 Write Requests

Stores are issued from the Qbox on one of two ports. Stores perform a DTB lookup and load an STA and STD entry. The translated address from the DTB is loaded into the STA, as well as the virtual address and the other related instruction information.

The store's virtual address also compares against the valid entries in the LQ. This is done to ensure that if a Load has been executed out of order to the same address as the Store, it can be replay-trapped.

When a Store retires, the store data are written to the Merge Buffer. When a Merge Buffer entry is selected for writing to the Dcache, the Dtag is accessed to ensure that the block is writeable. When the Merge Buffer entry completes its write to the Dcache, the SQA entry is invalidated.

I/O Stores cannot be completed speculatively; they cannot be cached. There is also a strict restriction of how subsequent I/O Stores can merge into one system request. The Dtags look-ups are irrelevant, because there will never be an I/O address in the Dcache on which an incoming Store can hit. But, since the translation occurs simultaneously to the first Dtag access, the Dtag access will continue until the tag comparison. At that point, when the physical address is determined to be in I/O space, the Dcache miss signal will be inhibited, and no miss request will be sent to the MAF. Still TBD is how I/O Stores that have made it to the Merge Buffer are coordinated with any I/O Loads in the MAF. I/O operations within a thread must beserviced in program order.

9.10.4 Retries

A Load which issues from the Qbox may not be able to complete in the Mbox for various reasons. It can have a Dcache miss (and SQ miss), have a bank conflict, or be a Load to I/O space which is not ready to be retired. In each of these cases, the Load is marked to be retried.

Every cycle, the retry logic in the LQ scans all the entries and finds the two oldest ready entries (in a given thread). Readiness is defined differently for each type of retry, but generally refers to when the Load is again able to make further progress. It then sends the information about the load to the pre-MAF. Retry candidates are chosen from different threads in a round-robin fashion.

For a Dcache miss, a retry is marked ready only after the Cbox has signaled to the Mbox that the data return is imminent. The retry readiness is detected by the LQ when the MAF number for the completing Fill (which is driven by the Cbox) matches the MAF number stored in the particular LQ entry. All Loads that have this MAF number will signal readiness to the retry logic. These Loads will have their retries serviced in INum order.

Bank conflict retries are marked ready immediately after they occur. As soon as the retried Load is the oldest ready retry in the LQ, it will be sent to the Dcache again. Retried Loads are guaranteed not to get a bank conflict.

When the Mbox receives the signal from the Qbox that an I/O Load is the next instruction to retire in a particular thread, the LQ marks the operation as ready to retry. I/O Loads will be retried in correct order due to the fact that they become ready in INum order and they are selected in the retry logic in INum order.

9.10.5 Dcache Misses

If a Load operation is found to miss in the Mbox, based on the Dtag and SQ look-ups, then the data are sought from the Cbox. The Cbox will first look for the data in the Scache. If they aren't found there, it will seek the data from the external memory system. Cbox activity is initiated by loading a MAF entry with the physical address of the missing Load. The physical address is driven to the MAF at the same time it is driven to the LQ. The lower portion of the physical address is driven from the input ports (12:0) and the upper portion comes from the DTB (47:13). The addresses for the issued Loads (a maximum of three per cycle) are held at the input of the MAF until the hit/miss is determined. Loads that have missed are queued. The addresses of the first two entries in this queue are driven to the MAF to compare with the addresses already in the MAF.

If a miss address is found to be in an address block of an entry already in the MAF, the address may be mergeable with the existing MAF entry, based on the merging rules (TBS). If the address doesn't match an already existent MAF entry, a new MAF entry will be allocated based on the MAF allocation policy (TBS). The MAF entry number where the miss is loaded (or merged) is sent back to the LQ and stored there. If there are no free MAF entries, though, the Load miss cannot continue yet; a retry is marked in the LQ entry for the operation.

The Cbox arbitrates between the Loads, Stores and I-stream requests that are in the MAF for its next Scache and system operations. The Cbox forwards the Fill data some time later to the Mbox and indicates which MAF entry the data are for. That MAF number is driven to the LQ. Any Load operations that are waiting for these returning data, will match and signal that one or more retry is required. The data are driven by the Cbox into the Mbox, which steers them into the FRD buffers, via the Back End Bus. When the Load retry (or retries) have made it back through the pipeline, the appropriate FRD buffer is addressed (either by a CAM or by an index, TBD) which drives the Load data bus.

Stores have to check the Dtags as they write to the Dcache. The Dcache state is most relevant to the Store right at the time it can commit its data to the memory system (when its data become system coherent). This look-up is necessary, because the Dcache state could have changed between when the Store first accessed the Dtags and when it is actually committing its data. A dirty Dcache block could have been victimized by the time the Store made it to retirement. If so, the Store needs to initiate a miss request, via the MAF. This time, though, the address is coming from the Merge Buffer. The returning data from the Cbox are sent to the Fill Buffer, and there is some interaction between the Fill Buffer and the Merge Buffer so that the newest bytes of the block make it into the Dcache, superceding the older bytes of the data block.

9.10.6 Load Locked/Store Conditional

Load-Locked and Store-Conditional operations are used by the Alpha architecture to facilitate data sharing by multiple processes in the machine. Generally, a processor attempts to perform the Load-Locked (or LDx_L) and Store-Conditional (or STx_C) as a pair of operations, atomically, both to the same address. The address represents a block of data in memory that is a shared resource. A processor attempts to read from and then write back to the shared resource before any other processor in the machine has written to that resource. The LDx_L issues first, loads data from the shared memory address, and performs an internal operation to denote that the operation has been per-

formed. The STx_C is only allowed to complete successfully if no other processor has written to this block. If it can be successfully completed, the STx_C performs the write, clears any internal state set up by the LDx_L, then returns a value to the source register denoting success. If the STx_C fails, no write is performed, the internal state set up by the LDx_L is cleared, and a return value denoting failure is written into the source register.

The 21464 implementation of LDx_L/STx_C is somewhat complicated by the fact that Loads and Stores can be issued out of order to the Mbox. Thus, the Mbox needs to wait until the retire point of the operations before performing the locking operation, as a way of guaranteeing that the processor has committed to the operation, and to maintain the proper ordering of events.

At its retire point, the LDx_L operation loads a lock register with its address. When the STx_C retires, it compares its address with what is in the lock register. If there is a match, the STx_C can complete successfully if and when the processor has exclusive ownership of the block. The STx_C retire is delayed until this exclusive ownership is acquired. If the processor already has exclusive ownership, this retirement delay is not very long.

When the STx_C is ready to complete, whether successfully or with a failure, it signals a retry to the Qbox, driving the STx_C INum back to the Qbox. In response, the Qbox "bubbles" the pipeline; which means that it allocates a Load port for this operation, but doesn't initiate a new operation to the Mbox. When this bubble reaches the point in the pipeline where cache data are usually driven for a Load operation, the Mbox will drive the success/failure bit onto the least significant data bit. The Qbox then writes these data into the correct register for the STx_C.

There are several situations that can cause an STx_C to fail. One of these situations is if the system is not be able to give exclusive ownership to the ownership request generated by the STx_C. Another case is when a Store operation to the same memory block, from another processor, beats the STx_C to the system memory. This competing Store will enter the processor as an invalidate, which will clear the lock register. A third way of failing a STx_C is if the Ibox receives an interrupt, a trap, any other control flow change (e.g., taken branches) or another memory opera ion between the time it has detected the LDx_L and when it has detected the STx_C, a STx_C failure must occur.

Because this last case is detected in the Ibox, a special bit must travel with the STx_C through the Qbox and then out to the Mbox. The Mbox stores the bit with the STx_C and upon its retiring, will use it as a condition for success or failure. If the bit indicates a failure, no further processing in the Mbox is done.

The STx_C, regardless of whether it can complete successfully, always clears the lock register at its retire time.

9.10.7 Traps

A trap may be signaled while an operation is in its initial dispatch pipeline in the Mbox, or it could be signaled later, after the operation has completed but before it has been retired. Traps clear all processor state relating to that instruction and all other instructions younger than it (in that thread). Traps are classified as either replay traps or exceptions.

Replay traps can be signaled due to the following: a Store address operation (front-end Store) dispatching through the Mbox finds a matching Load that has completed; a Store data dispatch (back-end Store) finds a matching Load in the LQ that belongs to another thread and the Load has completed; a Probe finds a matching Load in the LQ; SQ supplies incorrect data to a Load; Load gets a correctable ECC error. In each of these cases, the Load is restarted from the front-end of the pipeline.

Exceptions may be caused due to the following: a memory operation (Load or Store or Prefetch) finds a TB Miss or a violation (non-existent page, invalid operation); a Load encounters an ECC non-correctable error; a memory operation encounters a non-existent memory error.

A trap indication along with the INum, is sent on a special kill-bus to the Qbox. The Qbox arbitrates all the exceptions it is receiving this cycle and redirects the front-end suitably.

9.10.8 Invalidates/Probes

- Sources/Reasons
- Flow of Events

9.10.9 Memory Barriers

- General Concept
- Issues with Speculation
- Issues with Multi-threading

9.10.10 Multi-threading

- Support in Mbox for MT
- Implications of MT on Mbox Events

9.11 Interfaces

9.11.1 Pipeline Legend

Load/Store Issue Pipeline									
Q3	R0	R1	R2	E0	M0	M1	M2	M3	M4
Issue	Payload Read	Reg. File Read (I)	Reg. File Read (II)	Address calculation	Tag Read	Load Data Drive	Signal cache miss	Store Data arrival	Signal trap

Load/Store Queue Dealloc. Pipeline			
V9	V10	V11	V12
Mark Entries to Dealloc	Pick Retire Block	Update Hi Water Mark	Send HiH2O to Qbox

Store Copy-Out Pipeline					
V9		V10	V11	V12	V13
Retire INum Compare	Pick Retire Block	Read SQ Entry	Send Addr to Merge Buffer	Read SQD Entry	Send Data to Merge Buffer

Merge Buffer pipeline		
V13	V14	V15
Allocate Merge Buffer	Assert NAK to Store Queue	Pick entry to write through

Back End Bus pipeline					
MW	MX	MY	MZ	M0	M1
Bus Grant	Dtag Read	Dtag Write	Drv. DC_data	Dcache Write	Dcache validate

Scache write-thru pipeline					
MY	MZ	M0	M1	M2	M3
Bid for Scache	pick Merge Buffer entry	Drive to Scache	Cbox sends ack	Complete Scache processing	send *WriteThruDone* to Cbox

9.12 Data address Translation buffer (DTB)

Point to Mbox Contract section for introductory material.

Point to Interfaces.

9.12.1 Timing

Table 9–2 Memory Operation (Launch)

E0		M0		M1	
A	B	A	B	A	B
Receive OP Issue from Qbox	Ebox drives LD Addr; CAM ASN and TPU	Launch VA into TB, Tag, Stq	Read PA from TB; determine TB miss	Compare PA's; determine DC_Hit	

Table 9–3 HW_MTPR TB Invalidate, TAG or PTE Issue

E0		M0		M1	
A	B	A	B	A	B
Receive OP Issue from Qbox	Ebox drives LD Addr	Latch VA			

Table 9–4 HW_MTPR TB Invalidate or PTE Retire

V5		V6		V7	
A	B	A	B	A	B
Receive Retire from Qbox	Send Disable_Memop Bubble inum to Mbox retry/trap logic	Wait for bubble grant (about 18 cycles)			

Table 9–5 HW_MTPR TB PTE Retire Bubble

E0 of bubble		M0		M1	
A	B	A	B	A	B
Receive bubble from staging register	IPR drives Tag Addr, write ASN/TPUGRP	Write PTE Tag into TB	Write PA into TB		

Table 9–6 HW_MTPR TB Invalidate Retire Bubble

E0 of bubble		M0		M1	
A	B	A	B	A	B
Receive bubble from staging register	CAM ASN (IASN/IS)	CAM saved VA (IS only)	Clear TPU_Valid bits		

9.12.2 What Data are Compared on a DTB Lookup?

On every DTB lookup, we will need to receive the following inputs:

- Opcode issued
- TPU number (which is translated into a TPU group)
- Address Space Number stored in the ASN IPR for the above TPU
- Virtual Address Tag

Bits decoded from the opcode modify TB behavior in the following ways:

LD_PHYS	bypasses TB and sets PA = VA. (PA appears with the same timing as a normal LD).
LD_VPTE	Retry/trap logic generates DTB double miss fault instead of single miss.
ALT_MODE	Retry/trap logic uses mode stored in DTB_ALTMODE IPR instead of the CM IPR for access checks

A DTB hit produces both a physical translation and a protection mask. The protection mask is ten bits, consisting of read and write permissions for each of kernel, executive, supervisor and user modes, along with fault on read and on write. The retry/trap logic uses these bits, along with the current mode and opcode to determine if an access violation trap is required.

The VA comes from the Ebox in E0B. The TPU ID comes from the Qbox in E0A and is used to locate TPU group, ASN and mode information in the IPR section.

The ASN match is pre-evaluated one phase earlier, in E0B, and added to the VA CAM in M0A to make the DTB VA CAM match lines shorter.

The Ebox will signal a BAD_VA trap if the VA is not correctly sign extended, i.e., VA<63:52> != VA<51>.

For each DTB entry the following information is stored for comparison with the current VA.

ASN<7:0>	Address Space Number / process ID
ASM	Address Space Match
TPUGRP_VALID<3:0>	TPU Group Valid bits
VA<51:13>	Virtual Address Tag
VA_DONTCARE<24:13>	Decoded Granularity hint bits

Every application process has its own virtual address space. Therefore each process has its own Address Space Number. The ASN is used to specify which process the PTE is associated with.

The operating system can allow multiple processes to share PTE's. The Address Space Match bit allows this. If the ASM bit is set the ASN is not compared on a DTB lookup.

To allow the 21464's four TPUs to be used to create 2 or more independent virtual CPUs, a new mechanism to allow each PTE to be specific to a subset of the 4 TPUs will exist. Four TPUGRP_VALID bits will indicate which TPU group each entry is valid for.

9.12.2.1 The TPU Group

The TPU Group is a mechanism to control the sharing of address spaces and address space numbers among TPU's. TPU group membership is defined by the TPUGRP IPR and affects which lines can match on a DTB lookup. These equations will be described later, but for now, here is the high-level description of what all these things mean.

- The TPU Group delimits a space where TPU's in different groups have distinct spaces of ASN values, ASM bits, superpages and address mappings. TPU's in different Groups share the same relation to each other as different processors within a SMP machine. They could even be running different operating systems.
- The ASN delimits a space where TPU's share ASM entries and superpages, but have distinct mappings for non-ASM pages. TPU's in the same Group, but with different ASN, are running different processes within the same instance of an operating system.
- TPU's whose Group and ASN both match share their entire address space. They will be running different threads within one process and oneoperating system.

The Operating Systems people have referred to two different modes of operation, which correspond to different TPU Group assignments. (These mode names are informational only, and do not affect the design of the DTB.)

- Mode 2 (expected to be used by VMS) is the SMP-like mode, where every TPU is in a different group.
- Mode 3 (expected to be used in Unix) is the full multithread mode, where all TPU's are in one group.

To implement the TPU groups, each DTB entry has four valid bits, indicating which group the entry pertains to. At most one valid bit may be set for a given entry. Having all four bits clear indicates that the entry is invalid for all groups.

For a DTB entry to match when doing a compare the following condition must be met:

```
VA<51:13> == current_VA<51:13>              AND
TPUGRP_VALID<current_TPU_group> == '1'      AND
(ASN<7:0> == current_ASN<7:0>               OR    ASM == '1')
```

9.12.2.2 Granularity Hints

This condition become a bit more complicated with the addition of Granularity Hint(GH) bits. GH is a mechanism that allows contiguous pages to be treated as one larger page. GH bits allow recognition of pages of size 8x, 64x, and 512x. The 2 bit GH encoding is interpreted in the following manner:

Table 9–7 Granularity Hint Encoding

GH	Page	Size With 8KB Base	Size with 64KB Base	VA Compare
00	normal	8K page	64K page	compare VA<51:13>
01	8x	64K page	2M page	compare VA<51:16>
10	64x	512K page	64M page	compare VA<51:19>
11	512x	4096K page	512M page	compare VA<51:22>

The condition then becomes:

```
VA<51:22> == current_VA<51:22>          AND
( VA<21:19> == current_VA<21:19>        OR    GH == '11' ) AND
( VA<18:16> == current_VA<18:16>        OR    GH >= '10' ) AND
( VA<15:13> == current_VA<15:13>        OR    GH >= '01' ) AND
TPUGRP_VALID<current_TPU_group> == '1' AND
( ASN<7:0> == current_ASN<7:0>          OR    ASM == '1')
```

If this condition is not met for any of the DTB entries a DTB miss will occur.

9.12.3 64K Pages

The 21464 supports a 52-bit virtual address and 48-bit physical address. These widths lead to several complications when used with the 8K page size standard with the Alpha architecture. Chief among these is that the page table entry stores the physical page frame number as the upper longword of the quadword entry. A 32-bit page frame number and a 13-bit page offset combine to permit only a 45-bit physical address. For this reason, the 213464 has a 64K-page mode, with a 16-bit page offset, permitting the 48-bit physical address space.

A second benefit of the 64K-page mode is that the full 52-bit virtual address space can also be accessed using a 3-level page table, instead of the 4-level table that would be needed with 8K pages.

The 64K page mode is implemented by a bit stored (on a per-TPU basis) in the VA_CTL IPR. When set, this bit has the following effects:

- All granularity hint values are increased by one:

GH	8K	64K
00		VA<15:13>
01	VA<15:13>	VA<18:13>
10	VA<18:13>	VA<21:13>
11	VA<21:13>	VA<24:13>

- The PPFN stored in DTB_PTE<63:32> corresponds to PA<47:16>.
- The VPTE offset in VA_FORM<38:3> corresponds to VA<51:16>.

From a hardware perspective, the 64K mode inserts another set of conditions into the granularity hint logic, and a 3-bit shift into the PTE part of the DTB and into the VA input into VA_FORM.

9.12.4 Hit Determination

The DTB is also in charge of determining whether a particular load access hits in the cache. This is located at the DTB to minimize the load bus timing, even though it is not strictly a matter of address translation.

Hit determination is done by driving both sets' tag addresses from the corresponding tag array and comparing them with the translated physical address. A match sets the hit bit and steers the set select to the matching set. For test purposes, the DC_CTL IPR can

force the cache to always hit in one set. In addition, the Store Logic can force a load to hit, if Store Logic can supply data, or to miss, if it should supply data, but is unable to. The Fill Buffer can force a load to hit, if it is supplying data for an I/O load. These conditions go into the hit logic and override the results of the tag comparison. The hit indication is returned to the Mbox retry/trap logic, which, in the event of a miss, marks the operation for retry, and poisons its load data.

The output of this logic is the DC_HIT signal, which provides the overall indication of whether the operation generated valid data, along with drive enables to the set drivers, Store Logic and Fill Buffer to steer the proper data onto the M%LD_DATA_M2A<63:0> bus.

9.12.5 Returned Status

In the most general terms, either of two things can happen when a memory operation comes in. First, the DTB translation could succeed. In this case, the DTB returns a physical address, along with Dcache hit and set select. Secondly, the DTB translation could trap. In this case, the DTB returns a trap reason in M1A to the Mbox retry/trap logic. After a little while, when this gets back to the Qbox, the operation in question will be killed. Some time thereafter, PALcode will deal with the trap. Among other things, this means that the address and hit indications generated are irrelevant and are permitted to be garbage.

Trap processing is governed by the following rules:

- If the Ebox sent a poisoned address, all traps are inhibited. Poison indicates that the operation in question is the dependent of a load miss, and thus is garbage. The Qbox will reissue such operations after the load retry.
- Otherwise, if the Ebox sent BAD_VA, indicating that the sign extension check failed, we signal a BAD_VA trap. PALcode will need to emulate the instruction (pointed to by the EXC_ADDR IPR) to find out what the failing address was.
- Otherwise, if no entry in the DTB matched the address, and the opcode is a LD_VPTE, we signal a DTB_MISS_DOUBLE trap. PALcode will need to use the double miss flow to find the correct PTE.
- Otherwise, if no entry in the DTB matched the address, and the opcode is not a LD_VPTE, we signal a DTB_MISS_SINGLE trap. PALcode will need to use the single miss flow to find the correct PTE.
- Otherwise, if an entry in the DTB matches the address, but the PTE has protection bits inconsistent with the access requested, the Mbox retry/trap logic signals an ACV trap. The specific equations are as follows:

```
mode =  (OP == HW_LD/Alt || OP == HW_ST/Alt) ? DTB_ALTMODE : IER_CM/CM
prot  =  switch (mode):
            case KERNEL: (KRE, KWE);
            case EXEC: (ERE, EWE);
            case SUPER: (SRE, SWE);
            case USER: (URE, UWE);
ACV  =  (RD & (FOR | ~prot[0])) | (WR & (FOW | ~prot[1]))
```

9.12.6 Effects of a DTB Miss

As mentioned earlier, a DTB miss causes a DTB miss trap. The DTB miss trap is delivered to the Mbox central trap handler. The central trap logic checks whether the associated opcode is a LD_VPTE, which requires double-miss handling, or anything else, which is a single miss, and requests that the Qbox kill the faulting instruction and dispatch to the appropriate PALcode flow.

The PALcode trap handler determines the appropriate PTE from the operating systems page tables. It then fills this PTE into the DTB by writing to IPRs DTB_TAG and DTB_PTE. All communications with the DTB from the PALcode routine is done through writes to IPR registers, which will be discussed later.

The HW_MTPR DTB_TAG0 must issue before the HW_MTPR DTB_PTE0, and, similarly, DTB_TAG1 before DTB_PTE1. This is ensured by having the PALcode restriction that TAG0 must come before PTE0, and be in the same picker, and TAG1 must come before PTE1 and both must be in the opposite picker as TAG0 and PTE0. Having the MTPR to TAG0, TAG1, PTE0, PTE1 immediately adjacent and in that order satisfies this requirement. In addition, the MFPR VA must be before and in the same picker as the LD_VPTE. (Being before and in the same picker as the MFPR VA_FORM is also acceptable.)

When the PTE has been written to the IPRs, it is not copied from the IPR into the DTB until the instructions that write the IPRs have retired. To ensure consistency, in any flow containing writes to the DTB TAG or PTE IPRs, either the complete set of four MTPRs must retire, or none of the writers may retire.

So that all four TPU's can be working on TB miss flows at once without colliding, there must be four sets of physical IPR's, with one set visible to each TPU.

A thread must not complete another DTB fill flow while a prior fill flow in the same thread is unretired. This is done by scoreboarding the HW_MTPR TAG0 at the beginning of the DTB fill flow against the retire of the HW_MTPR PTE1 at the end of the previous DTB fill in the Qbox, as documented by the Qbox.

A new DTB entry is written from the IPR set into the DTB array when the HW_MTPR DTB_TAG1 retires. When this retire occurs, the DTB requests that the Mbox retry logic bubble back the HW_MTPR inum to the Qbox, which inserts a bubble into all four load/store pipes, and also releases any waiting DTB writes for that TPU. The DTB entry is written when the bubble arrives. In the meantime, memory operations in the same TPU as the writer of a retired but unwritten DTB entry can continue to use the copy of the DTB entry stored in the IPR.

9.12.6.1 Speculative and Duplicate DTB entries

Because the single DTB miss flow is performance-critical, DTB entries must be usable even before the DTB writer retires. At the same time, if the DTB writer turns out to be on a bad path, it must not have affected any good path instructions. In addition, since the DTB is a shared resource, this restriction also applies to other TPUs.

To permit all this, a speculative DTB entry stored in a DTB_TAG/DTB_PTE IPR set may be used for a memory operation if all of the following conditions are met:

- The DTB entry is not the result of poisoned data.
- The DTB entry has not been invalidated as a duplicate.

- The DTB entry was written by the same TPU as the memory operation.
- The DTB entry is older than the memory operation.

Because, among other reasons, two TPUs could execute a DTB fill almost simultaneously, it is possible for the speculative PTEs of two TPUs to translate the same address. These will never be used simultaneously, by the rules above. However, if one or both PTEs retire and are written to the DTB array, multiple DTB entries could be activated on a single operation. This has unpleasant electrical and logical consequences. The following rules ensure that an entry in the DTB array can never duplicate another entry in the array or a speculative PTE.

- A HW_MTPR TAG performs a CAM of the DTB when issued, and invalidates itself if a hit occurs.
- A HW_MTPR PTE performs a CAM of all of the speculative TAGs when it retires, and invalidates all matching speculative TAG/PTEs.

9.12.7 Data Storage in the PTE

The DTB uses a RAM array to store all data to be retrieved on a DTB read:

PA<47:13>	Physical address bits of 8K page.
UWE,SWE,EWE,KWE	User,System,Executive and Kernel Write Enable bits.
URE,SRE,ERE,KRE	User,System,Executive and Kernel Read Enable bits.
FOW,FOR	Fault On Write, Fault On Read.

When a PA is written to the DTB, each bit is either XORed with the respective VA bit or it is forced low. When the PTE is read from the DTB each PA bit is again XORed with the VA. This will restore the PA bit for any bit that was XORed on the way in. If a bit was forced low on the write then it will insert the VA bit on the read. This provides a mechanism for supporting GH bits and superpages(see later).

The PA bits are written in the following manner:

```
PA_write<47:22>   XORed with VA_write<47:22>
PA_write<21:19>   XORed with VA_write<21:19>   if (GH < '11')   otherwise write "000"
PA_write<18:16>   XORed with VA_write<18:16>   if (GH < '10')   otherwise write "000"
PA_write<15:13>   XORed with VA_write<15:13>   if (GH < '01')   otherwise write "000"
```

The PA bits are read in the following manner:

```
PA_read<47:13> is XORed with VA_matched<47:13>
```

The protection and fault bits are compared with the access requested, and the Mbox retry/trap logic sends back an ACV trap if inconsistent. The IPR section will also latch the VA and access bits in the MM_STAT and EXC_ADDR IPR's. The IPR logic tracks Mbox faults as they happen, together with the P%RK_KILL_V5A bus, to ensure that these IPRs reflect the last good path faulting instruction.

9.12.8 IPRs That Affect the Contents or Behavior of the DTB

DTB_TAG0, DTB_TAG1

The VA is written to the DTB_TAG register by use of the MTPR PALcode instruction. Four registers of each type exist (1 per TPU) although only one is visible to the user at any given time.

DTB_PTE0, DTB_PTE1

The PA and protection bits are written to the DTB_PTE register by use of the MTPR PALcode instruction. Four registers of each type exist (1 per TPU) although only one is visible to the user at any given time.

DTB_IA (Invalidate All)

When a write to this IPR retires all PTEs in the DTB are invalidated for TPUs in the writer's TPU group. PALcode must include an IFETCHB after this IPR is written and before any memory operation.

DTB_IAP (Invalidate All Process Specific PTEs)

When a write to this IPR retires all process specific(ASM==0) PTEs are invalidated for the writer's TPU group. PALcode must include an IFETCHB after this IPR is written and before any memory operation.

DTB_IASN (Invalidate Address Space Number) (Proposed)

When a write to this IPR retires all process specific(ASM==0) PTEs with the ASN of the current TPU are invalidated for the writer's TPU group. This has been requested by Unix, and may be implemented if they provide a justification and SRM change. PALcode must include an IFETCHB after this IPR is written and before any memory operation.

DTB_IS (Invalidate Single)

When a write to this IPR retires any entry in the DTB which matches the VA provided in the IPR(and current ASN of the TPU if ASM==0) is invalidated for the writer's TPU group. PALcode must include a IFETCHB after this IPR is written and before any memory operation.

M_CTL (Mbox Control Status Register)

When a write to this IPR retires, bits SPE<2:0> in this IPR enable the 3 superpage modes. Superpage enables only affect the group of TPUs belonging to the writer of M_CTL.

TPUGRP (Thread Processing Unit Grouping definition)

When a write to this IPR retires its contents define how TPUs are grouped. When a new PTE is written to the DTB, the PTE can be made valid for only the group of TPUs to which the writer belongs. Whenever a TPUGRP is reused the DTB must be flushed for that TPUGRP.

DTB_ALTMODE (Alternate Access Check Mode)

This is the mode used when a memory reference comes in by way of a HW_LD or HW_ST instruction with ALTMODE set. This is used to implement various probe operations, and must exist on a per-thread basis. In such cases, the Mbox retry/trap logic uses the access mode (kernel, executive, supervisor or user) stored here, rather than the one stored in the CM IPR.

9.12.9 Superpages

Superpages are an extension to VA –>PA translations. These mappings provide a translation outside the DTB for three regions of VA space. The translations are all set to permit access in kernel mode only.

Superpage0 is used to direct map one quarter of WindowsNT's 32bit address space. The kernel code is kept in this area of memory. It is believed that 64-bit NT will use the Unix superpage mode.

Superpage1 is used to direct map the least significant 41 bits of the physical address space (bits <47:41> sign extended) to support older versions of UNIX and VMS. This superpage is consistent with the 43 bit virtual address supported by EV4, EV5 and the size of the 3 level VPTEs used in Digital UNIX. (see SRM Digital UNIX II-B section 3.1.1).

Superpage2 is used to direct map the whole of the physical address space for more recent versions of UNIX and VMS which may use four level PTEs.

In hardware, we simply need a set of special match lines in the CAM which detect the following conditions:

```
If M_CTL[SPE<2>] = 1 AND VA<51:50> =   then  PA<47:13>=VA<47:13>, USEK="0001"
"10"
If M_CTL[SPE<1>] = 1 AND VA<51:40> =   then  PA<47:13>= "1111111",VA<40:13>,
"111111111101"                               USEK="0001"
If M_CTL[SPE<1>] = 1 AND VA<51:40> =   then  PA<47:13>= "0000000",VA<40:13>,
"111111111100"                               USEK="0001"
If M_CTL[SPE<0>] = 1 AND VA<51:30> =   then  PA<47:13>= #00000,VA<29:13>, USEK="0001"
"11111111111111111111"
```

M_CTL is an IPR. SPE<2:0> are the 3 superpage enable bits within the IPR. The operating system must ensure that no valid PTE in the DTB array will ever conflict with an active superpage, by never including the superpage region in the regular page tables.

When one of these conditions is detected - a special, hardwired PA entry is read from the PA array. Since all PA bits are XORed with VA bits as they are read from the array - the hardwired PA entries are as follows:

	PA<47:42>	PA<41>	PA<40:31>	PA<30:13>
Superpage 2:	All 0s	0	All 0s	All 0s
Superpage 1 & VA<40>=1:	All 0s	1	All 0s	All 0s
Superpage 1 & VA<40>=0:	All 1s	0	All 0s	All 0s
Superpage 0:	All 1s	1	All 1s	All 0s

Since superpages are enabled separately for different TPU groups; for each superpage mode - a full, four bit mask of TPUGRP_VALID bits will be stored. For a particular superpage comparator - only the TPUGRP_VALID bits corresponding to the writer's TPU group will be affected when M_CTL is written (bits outside the group should be left as they are and not cleared). The superpage TPUGRP_VALID bits are not affected by IA, IAP or IS. They do have a reset mechanism so that the whole DTB can be flushed during Power on Reset and when TPU groups are modified by a write to the TPUGRP IPR.

9.12.10 Possible Support for Generic Superpages

Bruce mentioned the idea of using generic superpages. Instead of hardcoding the superpages as additional DTB entries, there could be several generic superpage entries. This certainly seems possible from an implementation standpoint with the following implications:

9.12.10.1 Page Table Array(PTA) Implementation

Since the superpages would no longer be hardcoded the translations would need to be filled into the DTB instead of simply enabled using the SPE<3:0> bits. The XOR scheme talked about above would still allow generic superpage translation for the existing superpage modes.

9.12.10.2 Virtual Address Array(VAA) Implementation

The DTB uses cam cells to compare the incoming VA with the VA stored for each entry. For the superpages the cam cells are removed and the superpage code is hardcoded into the array. The superpages vary in size and VA space mappings. Generic superpages will require the addition of a 3 state cam cell into the cam array. The third state would be 'don't care'. This would allow the cam structure to compare only a subset of VA bits. The EV6 DTB used 2 cam cells to build a 3 state cam cell for the GH bits. Something similar could be done for the generic superpage entries.

9.12.11 Replacement Policy

Least Recently Used(LRU) vs Not Last Used(NLU) vs Round Robin?

The 21264 used a round-robin replacement policy. What will the 21464 use?

Some quick experiments indicate that (on the SQL benchmark), compared to Round Robin, NLU improved miss rates by 1 / 1k inst, and LRU by 8/ 1k inst.

Given the performance differences, the complexity of implementation, and the impact of anything other than round robin on verification, the DTB will

use round robin replacement.

9.12.12 DTB Size

It has been observed that a 128-entry DTB is incapable of mapping a 2MB SCache using default 8K pages. One could build a 2-set 256-entry DTB to do this. This would essentially be two copies of the current TB, with the choice of set done by looking at VA<13>.

A preliminary experiment (using SQL again) indicates a 13 / 1k inst reduction in TB miss rates with this design. Based on the above, the DTB will have 128 entries.

9.12.13 ITB Usage

The core of the DTB design will also be the core of the ITB design. Notable differences include the following:

- There is only one ITB, but there are four DTBs.
- The ITB only stores 5 protection bits, vs. 10 in the DTB.

- The ITB does not have speculative entries and will use that logic to handle the micro-ITB instead.
- The ITB does not have hit determination logic.

9.12.14 Reset and Testability

All IPRs, DTB Valid bits and the DTB Write Pointer must be cleared on reset, except that the DCache set enable bits in the DC_CTL IPR reset to one. If this proves impractical, a fallback position is to require that reset PALcode execute a DTB_IAG to reset the write pointer and a DTP_IA to invalidate the array for each TPU group and to write M_CTL and VA_CTL appropriately, before any virtual loads and stores are issued.

For test purposes, the test logic requires access to the virtual address (lookup), fill virtual address, PTE read data, PTE fill data and write pointer. The virtual address lookup path is critical; the test port should probably be included as a leg of the retry address mux.

Some caution is in order regarding nomenclature. The DTB is a CAM/RAM structure whose data elements are addresses. Thus, the virtual address is the data part of the CAM, the physical address is the data part of the RAM, and the write pointer is the address part of the CAM, when viewed from the perspective of the test logic.

9.12.15 Issues

1. Unix changes PTEs without doing TB invalidates. Unix toggles the protection bits without invalidating the PTE. The duplicate prevention logic must prevent ill effects from resulting.
2. The combination of GH=11 and 64K pages means that GH cells have to go up to VA<24>. EV7 seems to be doing this.
3. Enforcing issue ordering between MTPR TAG and PTE on DTB fill. This can probably be done by slotting the PALflow correctly. If not, it could be done by using another IPR reader class.
4. Is EBox doing the BAD_VA trap? Chris has volunteered that EBox could signal the BAD_VA trap, since they are already signalling Unaligned.
5. Is there a separate IPR bus? Ebox feels that a separate bus for MFPR which acts like a multimedia result bus is the cleanest way.
6. Should we make all explicitly written IPRs readable? This is for testability.
7. Should generic superpages be added? What is the mechanism? Concern is the number of 3-state CAM cells.
8. Bit assignments and encodings for Trap Reason and IPRs.
9. Will Pbox ensure that we get Kill INum before freshly issued post-kill instructions. (They have to do this for all boxes, not just us.) The current timing is V7=I2 of the good path flow.
10. Duplicate suppression on bad paths requires handling the following conditions: Spurious writer with GH greater than that of the overlapping entry in the main array won't hit main array unless we bump up the GH value. If we CAM array at MT TAG issue, GH bits and ASM are not available then. CAM at retire could hit a half-issued (TAG only) speculative entry, which doesn't have any GH bits or ASM. We could make speculative entries work onlywith GH=00.

11. Approval to add DTB_IASN. Having it present still maintains backward compatibility.
12. PALcode restrictions.
13. EV6 and EV7 compatibility. EV7, in particular, is considering changes to 64K mode and GH semantics.
14. Any issues relating to use of the DTB as the ITB.

9.13 Store Logic

9.13.1 Overview

The SQ, SQD, and SQC (collectively known as the Store Logic or STL) work together to form a 64-entry reorder buffer for all store instructions and a number of "cache movement instructions". These instructions are listed below:

Store Instructions	STB, STW, STL, STQ, STQ_U, HW_ST STF, STG, STS, STT STL_C, STQ_C QUIESCE
Cache Movement Instructions	WH64, ECB, CCB, WMB

The SQ buffers nearly all information for a STL instruction (except an actual store's data), and contains the allocation and deallocation functions. The SQD buffers the data portion of stores, supplies that data to loads that require it, and supplies that data to the back-end process that copies stores into the cache/memory system. In order to perform these tasks with the required timing, the SQD duplicates some of the functions of the SQ. The SQC contains additional control and sequencing for more complicated store instructions and MB instructions.

The STL supports three major instruction pipeline flows: Store Issue, Load Issue, and Store Copy-Out. Each of these flows takes place on a different port, although there may be some sharing of wires between the Store Issue and Load Issue ports. The STL supports two Store issues, and two Store Copy-Outs per cycle. In addition the STL can process up to 3 Load Issues per cycle, although only 2 loads are permitted if there are two Store Issues in the same cycle.

The STL divides the 64 entries into 16 blocks of 4 entries, where each block will contain four consecutive stores (in program order) for a specific TPU. The number of blocks allocated to each TPU is dependent on the number of active (non-quiesced TPUs). A fifth pipeline flow controls the reallocation of a block to the same or a new TPU.

In order to avoid the Qbox issue logic overflowing the STL buffering capability, the Pbox assigns a store number (SNUM) to each instruction that will be processed by the STL. The SNUM is a sequentially increasing identifier for STL instructions in program order from a single TPU. The SNUM is unique among instructions for a given TPU, but may be duplicated for instructions in another TPU. The STL specifies to the Qbox issue logic a SNUM high-water mark, which is the largest SNUM that the STL has buffering for. The Qbox will not issue STL instructions whose SNUM is greater than the high-

water mark, which prevents STL overflow. When a STL block is added to a TPU, or one is ready for reuse, the STL will increase the high-water mark for that TPU, and the Qbox will enable the next four STL instructions to issue.

The STL holds information for an STL instruction from the time it issues until it retires and other structures are updated. If the instruction refers to an address that is represented in the DCACHE, then the STL entry will not be deallocated until the DCACHE is updated. If the instruction refers to an address that is not in the DCACHE, then it can be deallocated as soon as the instruction has been copied into the MGB.

While a store instruction is buffered in the STL, it will attempt to supply data to appropriate younger loads in the same TPU. The STL will supply data, SNUM, and status for each load specifying that the load should:

Status	Action
STL_MISS	Use DCACHE data
STL_HIT	Use SQ data
STL_RETRY	Use SQ data, but SQ can't supply it now

If STL_RETRY is signaled, the LQ simply retries as soon as possible.

In order to facilitate the STL finding the appropriate store instruction for a given load, the STL computes an active range of INums for each STL entry. This range is a conservative estimate of the load INums that should use this entry, and is initialized by the Store Issue for the STL entry. Subsequent Store Address Issues may reduce the end of the range if the incoming STL instruction is in the current range and the SQD address comparison would match the original entry.

An incoming Load Address Issue, causes the SQD to read entries whose range includes the INum of the load, and whose SQD address overlaps the load. The STL logic guarantees that there will be either zero or one entries that meet these criteria. At the same time, the SQ is computing the appropriate status for the load access. In most cases, STL_MISS or STL_HIT will be returned enabling the load to complete as far as the STL is concerned. In a very few cases, the STL logic will be incapable of supplying the correct data, and the SQ will return STL_RETRY. The load is then retried continuously until the STL condition is cleared (usually by some entry being deallocated).

When a STL instruction retires its SQ status is changed to retired, which causes it start requesting service from the Store Copy-Out logic. The Copy-Out logic processes retired STL instructions in program order, and can handle two STL instructions from a given TPU in a single cycle. The Copy-Out process sends the STL instructions to the MGB (merge buffer) which responds with a positive or negative acknowledgement. If the MGB supplies a negative acknowledge, then the STL instructions will be sent again, when that TPU is selected. Eventually the MGB sends a deallocate message to the STL, which changes the SQ status to deallocate, which enables the block containing the entry to be processed by the reallocation pipeline.

9.13.2 Store Issue Flow

The Store Issue loads the information about an STL instruction into the SQ and SQD. The entry that is to be loaded is determined by the SQ by matching the incoming instructions TPU and SNUM against the TPU and SNUM stored in each block of the SQ. The block TPU and SNUM are initialized by the reallocation process and described later.

Each STL entry computes an active range of INums within which this STL entry will be visible to Load Issues. This range is computed in a conservative fashion so that for a given Load Issue, only zero or one entries of the SQD will be read. This range is required because a number of STL instructions to the same address can be resident in the STL at a given time. The SQ entry stores the virtual address and physical address for STL instructions, however only the virtual address is duplicated in the SQD. Two STL instructions are defined to "overlap" if the they are on the same TPU, the quadword SQD index is the same, and that some addressed bytes are shared by the two instructions.

The lower end of the active range is simply the INum of the STL instruction until it retires, at which point it is changed to be "negative infinity" (older than any instruction to be issued). The end INum of the range (or EINum) is initialized to be "positive infinity" (younger than any instruction to be issued). When the EINum of an entry is retired, it is changed to be negative infinity, which effectively disables this entry of the STL from supplying data to loads. For each subsequent Store Issue, each entry in the SQ updates its EINum if appropriate. If the incoming Store Issue's INum is younger than the entry's INum and older than the entry's current EINum, then the entry's EINum will be set to be the incoming Store Address Issue's INum.

9.13.3 Load Issue Flow

The three load address ports are used to look up the STL to determine if some data for this load should come from the STL. The TPU, INum and virtual address of the load are used to find a single entry that is enabled to supply data to this load. The TPU, INum and physical address of the load are then used to determine if the correct entry was selected using the virtual address. If this second comparison determines that either the wrong entry was selected, or that there were more than one entry required for the load to complete, it signals this fact using STL_RETRY.

9.13.4 Store Copy-Out Flow

When a STL instruction retires its SQ status is changed to retired, which causes it start bidding to be copied into the MGB (merge buffer) and DCACHE. The Store Copy-Out process selects a TPU that has some STL instruction bidding to be copied-out. It then selects the oldest block that contains a bidding STL instruction, and copies out the two oldest bidders from that block (if there is more than one). The SQ commands the SQD to read the data for those STL instructions, and examines their opcodes and physical addresses. If there is no overlap in the physical addresses and the operations are compatible, both are sent along with the data from the SQD to the MGB. After a fixed delay, the SQ gets an acknowledgement from the MGB telling the SQ that each STL instruction was either accepted or rejected, and the MGB index it was merged into if accepted. The SQ stores this information to control deallocation of the SQ/SQD entry. If the entry was not accepted, it becomes enabled to bid again. In some instances, the STL instruc-

tion requires special handling by the Copy-Out process (WMB, STx_C, I/O Stores). In this case, the Copy-Out process stops copying out STL instructions for that TPU until the operation is done.

Because there is a window of a couple of cycles between the sending of an instruction and the acknowledgment by the MGB, it would be possible to have a number of instructions inflight toward the MGB. In order to process WMB instructions, the retire ports will not send any instructions after the WMB until the MGB sends acknowledgment for instructions preceeding the WMB. In addition the retire ports wait until all merge buffers for this TPU are made coherent before sending any new instructions. The retire ports process STx_C like ordinary stores, except that STx_C holds up the retire point. This means that no other stores can be processed after the STx_C in this TPU. When the STx_C is sent to the MGB, it will reply with a STx_C_success signal in addition to the not_accepted signal.

9.13.5 Block Allocate Flow (TBD)

9.13.6 Things Not Done

Store conditional processing (in control section) IO processing signals

9.14 Merge Buffer

9.14.1 Overview

The Merge Buffer holds Stores from the SQ after they have retired and before they have updated the Dcache and the Scache. The MGB helps to accumulate Stores to the same cache block by allowing data from multiple Stores to fill up the same MGB entry. Accumulating store data for a whole cache block, conserves system bandwidth by avoiding data transfers for cache blocks that are completely dirtied; the CBox does a CtoD transaction on the system. Furthermore, since the Scache can be written at a minimum granularity of a quad-word, the MGB acts as a holding place for stores which write less than a quad-word thus avoiding Scache fills (in order to merge byte/word/lword write from the processor).

The Merge Buffer has 16 entries. Two input ports, each providing a data and address path, are driven from the SQ, and up to two entries can be allocated each cycle. The addresses compare against the existing MGB entries. Each entry contains a physical address, 64 bytes of data, a 64-bit mask to indicate which bytes of the block need to be written into the Dcache and another TBS mask to indicate which data needs to be written to the Scache.

If a Store address matches one already in the Merge Buffer, the Store's data are loaded into that entry, merging with the data there. The entry's byte mask is updated to reflect the new bytes that are being written.

The Merge Buffer arbitrates with the Fill Buffer (Cbox fill returns) for the Back End Bus. Once the MGB wins arbitration, it looks up the Dtags. If the cache block is writeable, the data block is updated in the Dcache and the Dtags are updated accordingly.

If the block doesn't exist in the Dcache, a miss request is made to the MAF. If all 64 bytes of the MGB entry have been written, a block ownership request is launched; otherwise a block fill is initiated.

Fills may need to merge data from the Merge Buffer before it updates the Dcache. Hence, Fill addresses search the Merge Buffer for a match at the same time that the Btags are looked up. If a match is found, the Merge Buffer drives the valid bytes onto the DC_Data Bus, effectively merging with the fill data.

Probes too need to check the MGB. Probes check the Merge Buffer at the same time that they check the Dtags. If a probe hits on a Merge Buffer entry, the Merge Buffer reads out the valid data bytes and sends them to the Cbox.

9.14.2 Merge Buffer Allocation

Every clock, the Store Queue sends up-to-two store addresses over to the Merge Buffer. The incoming store may merge with an existing Merge Buffer entry if :

- Physical address (PA<47:6>) and VA<14:13> matches
- Both are pure store ops (not IO, WMB, STC etc.)
- If they are to different TPUs, then they may merge only if the block is owned

If the new address cannot be merged or allocated in the Merge Buffer, then a NAK is sent to the SQA. The Store Queue needs to resend this store later. In this case, the merge_buffer also sets write_to_Dcache (unless it is already set) on the entry with which it could not be merged.

It is also possible that an address matches and the state is owned, but neither Dcache nor Scache processing is pending (this is signified by the free_to_allocate bit being set). In this case, the thread_id[1:0] is not valid and need not match. Merging is legal in this case and the thread_id[1:0] is set to the new thread.

PA Match && VA Match	TPU Match	State	Merge
Yes	Yes	X	Yes
Yes	No	State==Owned	Yes
Yes	No	State!=Owned	No
No	X	X	No

If the store is accepted from the Store Queue, the merge buffer index that the store merges with or is freshly allocated to, is sent over to the Store Queue. If a store does not merge with an existing entry, a new entry is allocated only if one is available (has its free_to_allocate bit set).

If there is no free entry around (and the incoming store does not merge), the merge buffer NAKs the store copy-out. The Store Queue stalls (and continues to make the same request), until either the Store Queue asserts Purge_mgb (Store Queue starts filling up) or if an entry becomes available (because its age counter times out).

9.14.2.1 Boundary Case

If the SQ sends a request such that it tries to set the dcache_dirty[64] bits at the same time it is being cleared from a previous write dispatch to the Dcache (in DC_data stage of the Back End bus pipe), then the setting of dcache_dirty takes precedence over clearing.

9.14.3 Merge Buffer Writes to Dcache

A Merge Buffer entry will be marked for write_to_Dcache once it satisfies either of the conditions:

- entry is not an IO-store

  and if

- Line is fully dirty (all dcache_dirty bits are set)

  or if

- Its timer has timed out

  or if

- Purge_mgb is active and the entry's TPU matches Purge_mgb_TPU[3:0]

  or if

- A STx_C op is allocated to this entry

Purge_mgb_TPU[3:0] is sent by the Store Queue indicating that a given TPU needs to have its Merge Buffer entries flushed, in order to make room in the Store Queue. When Purge_mgb_TPU[3:0] is active, a request is made to the Back End Bus controller, which then disallows the PreMAF from sending a request to the CBox in the following clock. If in the Scache tag launch stage, the Back End Bus receives an ACK implying that the Scache is not running a cycle this clock, then the Back End Bus may be granted to the Merge Buffer. The Store Queue also asserts Purge_mgb_TPU[3:0] when it starts to process a WMB.

A picker picks a Merge Buffer entry (to write to the Dcache) when the Back End Bus controller indicates that the following cycle may be used by the Merge Buffer (to make a request on the Back End Bus controller). The picker may pick an entry only if its write_to_Dcache status is set. The picker may be simply a priority encode of the write_to_dcache bits. The picked entry is read and the address sent to the Back End Bus (in the same clock).

During the Dtag Read stage of the Back End Bus transaction, the Btag responds with the Dcache state

(cache_state={valid,shared,owned}). If the line is owned, the Scache set (scache_set[1:0]) is sent to the MGB.

If the Dcache does not have write permission (dcache_state!=owned), it aborts the data portion of the transaction by de-asserting DC_data_valid. Else, the Merge Buffer assumes that the data is going to be successfully written to the Dcache and therefore it is safe to ask the Store Queue to deallocate the entry(ies) that corresponded to the Merge Buffer entry.

The Merge Buffer data is accompanied by the 64 dcache_dirty bits.

After the data is sent over the Back End Bus, the dcache_dirty bits are reset (if DC_data_valid is asserted and the Cbox indicated that there was no ECC error).

If the Btag read revealed that the line is not valid in the Dcache (state!=valid), the data phase of the Back End Bus is aborted (DC_data_valid=0). However, in this case, the Merge Buffer assert MGB_id_dealloc<3:0>, to allow the Store Queue to deallocate. If

the line does not have the correct state (state is invalid or line is not owned by the processor) or if line_fill_needed bit is set, a request is made to the MAF via the PreMAF. The Merge Buffer may send one of 3 types of requests to the MAF (PreMAF):

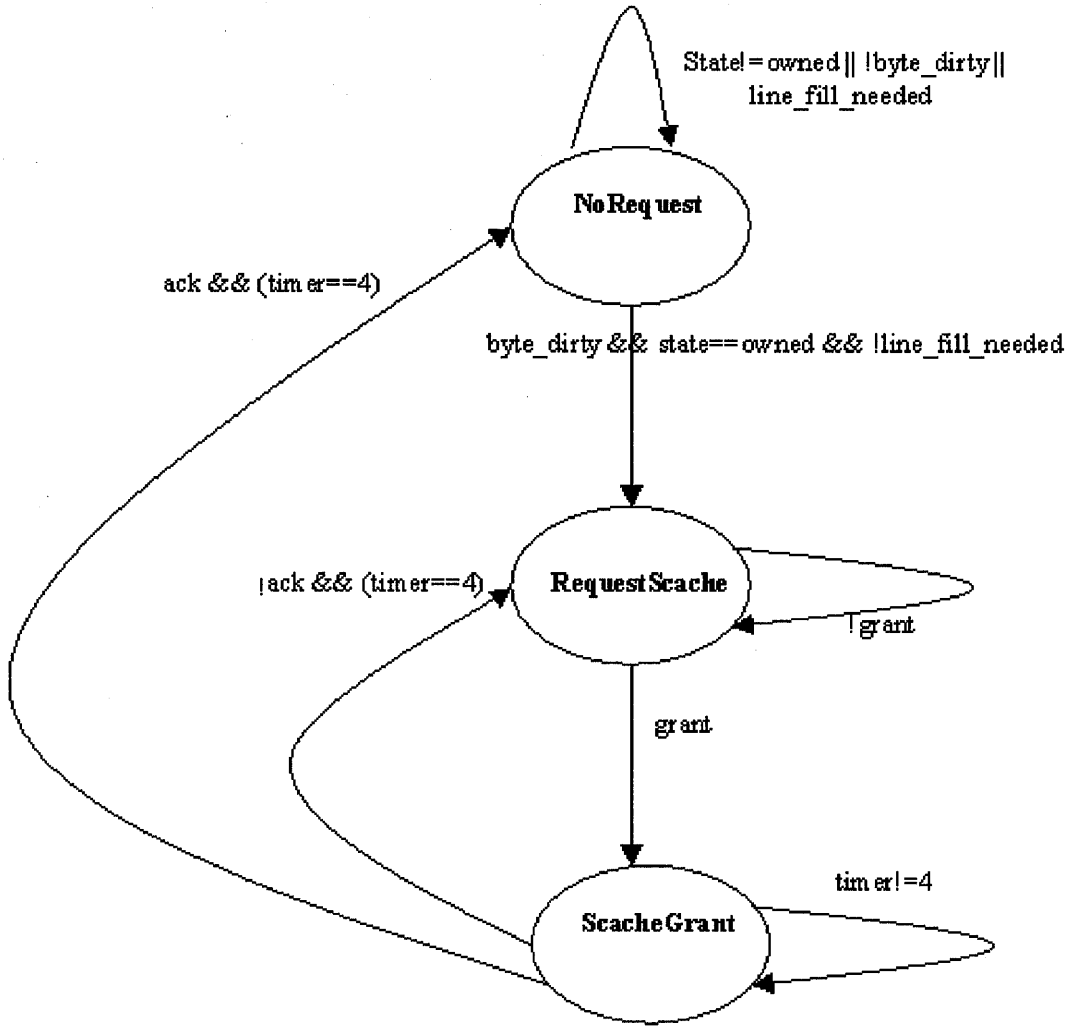
- ItoD: if all bytes are dirty, but the state~=valid
- CtoD: if all bytes are dirty, but the state==shared
- ReadMod: if line_fill_needed is set

It is possible, that the Cbox may refuse this request (send NAK in Scache tag launch stage). In general it is expected that the probability of this happening is low (bank conflicts due to internal Cbox cycles); hence, if a request is NAK'ed, the write_to_dcache bit is once again set and the request is routed via the Back End Bus to the MAF. Redoing the entire Back End Bus cycle may seem wasteful, but this avoids SILO'ing the request and re-issuing it to the MAF, when the CBox detects an occasional collision with an internal (Cbox) request. Note that the bytes that were successfully written into the Dcache, the first time around, are not re-written again; the MGB_id_deallocate[3:0] that is sent out corresponds to the new bytes that are written this cycle (if any). If no bytes are written into the Dcache during this repeat cycle, STQ_dealloc (which is used to qualify MGB_id_deallocate[3:0]) is set to 0.

9.14.4 Scache Writes

An entry needs to be written through to the Scache if there are qwords ready to be written. The following figure illustrates the various states of the Scache write through process.

Figure 9–2 Scache Write-Through Process



Each entry has the state machine shown above. Thus, until an entry is ready to be written out to the Scache, it stays in the NoRequest state. An entry may request the Scache picker to write through, to the Scache only after the entry reaches the Btag Write stage of the Back End Bus pipeline. Past the Btag Write stage, the Merge Buffer entry checks the TBS bit and transitions to RequestScache state, provided line_fill_needed bit is not set (line_fill_needed is set in the Btag Write pipe stage, if the cache block needs to be fetched, because all the byte_dirty bits in a quadword are not set). TBS is set if any byte_dirty bits are set in the octaword.

Once an entry is allowed to drive its data to the Scache, it transitions to the ScacheGrant state; the grant signal is used to read the (PA) address (to be sent to the CBox). It is presently expected that it will take 4 cycles to receive the ack signal from the Cbox (implying that the data was accepted). Hence a 2-bit timer needs to be loaded to count down the time it takes to receive the ack signal (alternately, one could stage this state for 4 cycles). At the end of the delay, the ack is sampled. If ack is asserted, then it transitions to NoRequest state else it transitions to the RequestScache state, in order to repeat the request (to the Cbox).

9.14.5 Probe handling in the Merge Buffer

Probes arbitrate for the Back End Bus (just like fills) and check the Btags once granted. The merge buffer is checked at the same time and if the address matches and if dcache_state == owned, merge buffer responds with mgb_hit . If there is a hit, the Merge Buffer writes the VDB_idx[5:0] which is sent by the Back End Bus (from the CBox). If a probe (with invalidate) hits the Merge Buffer, line_fill_needed is not allowed to be set (unless it is already set); this is different from other Back End Bus transactions (in all cases, line_fill_needed is set if needed).

Note that data evicted by probes (unlike Scache write-throughs), are not full quad words. Hence 16 byte_dirty bits are read (along with the octaword data) and sent to the Victim Data buffer (in the CBox).

A separate signal WriteThruDone is sent, once the ack for each of the octawords that were sent (to the CBox) are received back (no octaword_dirty bits are left set). The CBox closes the Victim Data buffer entry once it receives WriteThruDone.

9.14.6 Line fill and Merge Buffer

After the Btag Read stage of the Back End Bus pipeline, the Merge Buffer entry may decide to launch a FetchLineMod request to the MAF (via the PreMAF) if any quad-word does not have all the byte_dirty bits set (although atleast one byte_dirty is set) and if the line_valid bit is not set (implying that the complete line is not valid). This status is latched in the line_fill_needed bit for the entry.

During the Btag Read stage (of the Back End Bus pipeline), the Merge Buffer is CAMed (along with the Load Queue). If there is an address match, the Merge Buffer asserts mgb_hit.

Based on cache_state (cache_state=Btag_state|MGB_state|Cbox_state), following are the actions taken by the Merge Buffer.

- cache_state==owned.

  This is the case either when we initiated a fill from the Scache since we didn't have a qword full of data to send to the Scache (line_fill_needed is set), or a fill initiated by a load request, is returning from the Scache or system. The Merge Buffer indicates that it will drive the DC_data bus corresponding to the byte_dirty bits that are set. The merge buffer reads out the bytes corresponding to byte_dirty and sources the data onto the DC_data bus. The merge buffer receives the entire DC_data bus and in the cycle that the DC_data bus is being driven (towards the Dcache) it also writes itself (effectively merging the fill data with the existing dirty data).

  The dcache_dirty bits are read out and sent along with the DC_data (to enable writing to the Dcache, only the bytes that need to be updated).

  2 cycles following the data return (from the Cbox), the ECC status is returned. If no errors are reported, the Back End Bus is responsible for setting the valid bit on the Btags; else if a correctable ECC error is reported, the valid bit is left unset.

  If no (ECC) errors are reported, the line_fill_needed bit is reset, all the byte_dirty bits are reset and the line_valid bit is set. If a correctable ECC error is reported, the line_fill_needed and byte_dirty bits are left unchanged; the line_valid bit not set .

  If the Dcache accepts the Back End Bus request (Back End Bus indicates DC_data_valid), the dcache_dirty bits will be reset.

The mgb_state of the Merge Buffer entry is set to cache_state .

Note that if the cache block does not exist in the Dcache (Btag state==invalid), the Back End Bus controller is responsible for

asserting all of the dcache_dirty bits.

- (cache_state==valid||shared)

  This is the case when a fill (initiated by a load) returns on the Back End Bus but the final state is shared (not owned).

  The data is merged (as above) and written into the merge buffer.

  The Merge Buffer indicates to the Back End Bus that the transaction needs to be aborted (abort_fill) i.e the Dcache will not be written (DC_data_valid is set).

  Load retries triggered due to the data fill will be satisfied, since the merged data will still be driven onto the load data bus.

- Line comes with ownership, but no data.

  This is the case when a CtoD was sent in the past. The Back End Bus transaction proceeds as normal (using the dcache_dirty bits as set by the merge buffer).

9.14.7 IO Stores

IO stores do not set the write_to_dcache bit; instead, when the CBox is ready to send the IO store to the system, it sends the IO store address on the Back End Bus. Once the Merge Buffer entry (corresponding to the IO store) matches the address, it is forced to write through to the Scache. The entry is deallocated after the Scache write through is complete.

For a complete description of IO handling refer to the IO handling document.

9.14.8 Store Conditional Support

STx_C ops may not merge with existing entries in the Merge Buffer. A fresh entry needs to be allocated. However, unlike other store ops, STx_C does not send the ack to the Store Queue until it has obtained ownership (or been refused ownership) of the cache block. If ownership is obtained, the Merge Buffer acknowledges the Store Queue request and at the same time writes the STx_C into the Dcache. If on the other hand ownership is not obtained, the Merge Buffer acknowledges the Store Queue, but fails the STx_C and consequently does not update the Dcache.

Thus, write_to_dcache bit is set as soon as the STx_C is allocated into the Merge Buffer.

Past the Btag Read stage of the Back End Bus pipeline, the STx_C is ack'ed if the block is owned. If not, a CtoD STx_C request is routed to the PreMAF. Once the CtoDStXC request completes, the STx_C is known to pass or fail (in the Btag Read stage).

The Merge Buffer receives lock_TPU<3:0> from the Load Queue. The lock_TPU<3:0> signifies that the TPU (indicated by the corresponding bit) owns the lock register (set by a previous LoadLock instruction); if at any time, the TPU deasserts the lock, the Merge Buffer is obliged to fail the STx_C.

If a previously initiated CtoDStXC operation returns (on the Back End Bus) and does not find a matching Merge Buffer entry, the CtoDStXC is aborted (DC_data_valid=0) on the Back End Bus.

9.14.9 MB and WMB Processing

Every clock, the merge buffer will send out a 4-bit vector TPU{0,1,2,3}_coherent . Each bit in the vector is set if all entries relating to that TPU (bit 0 signifies thread 0) in the merge buffer have dcache_state = owned. Thus each Merge Buffer entry has the TPU{0,1,2,3}_owned vector; this vector is derived from mgb_state[2:0]. TPU{0,1,2,3}_coherent is essentially a wired-AND of all the TPU{0,1,2,3}_owned bits.

When a MB is in flight, the Store Queue should assert Purge_mgb along with Purge_mgb_thd[3:0] to assure prompt purging of merge buffer entries for that thread.

The Store Queue will use TPU{0,1,2,3}_coherent to decide to proceed with retiring an MB. WMB is retired at issue time, but the Store Queue needs to ensure that no stores (for the thread in which the WMB is present) are sent to the Merge Buffer until Thd{0,1,2,3}_coherent is active (for the thread).

9.14.10 MAF request

The Merge Buffer may make one of the following requests to load the MAF (via the pre-MAF):

ItoD: the line is completely dirty in the Merge Buffer, but the line is not present in the dcache (dcache_state != valid).

CtoD: line is completely dirty, but does not have ownership (dcache_state != owned)

CtoDSTxC: this is sent out for Store Conditional operations

FetchLineMod: line does not exist in cache and the line is not completely dirtied in the Merge Buffer

Inval: this is caused due to the Evict instruction. The effect is to do an internal probe with invalidate. The PreMAF gives the highest priority to Merge Buffer requests; however, the CBox may reject the request 3 cycles later. Therefore, the write_to_dcache status is staged and may be set back, if the CBox rejected the request.

9.14.11 Cache Movement ops (WH64, Evict)

TBS

Evict is sent from the Store Queue. Neither the byte_dirty nor the dcache_dirty bits are set. Evict sends an Inval request to the MAF.

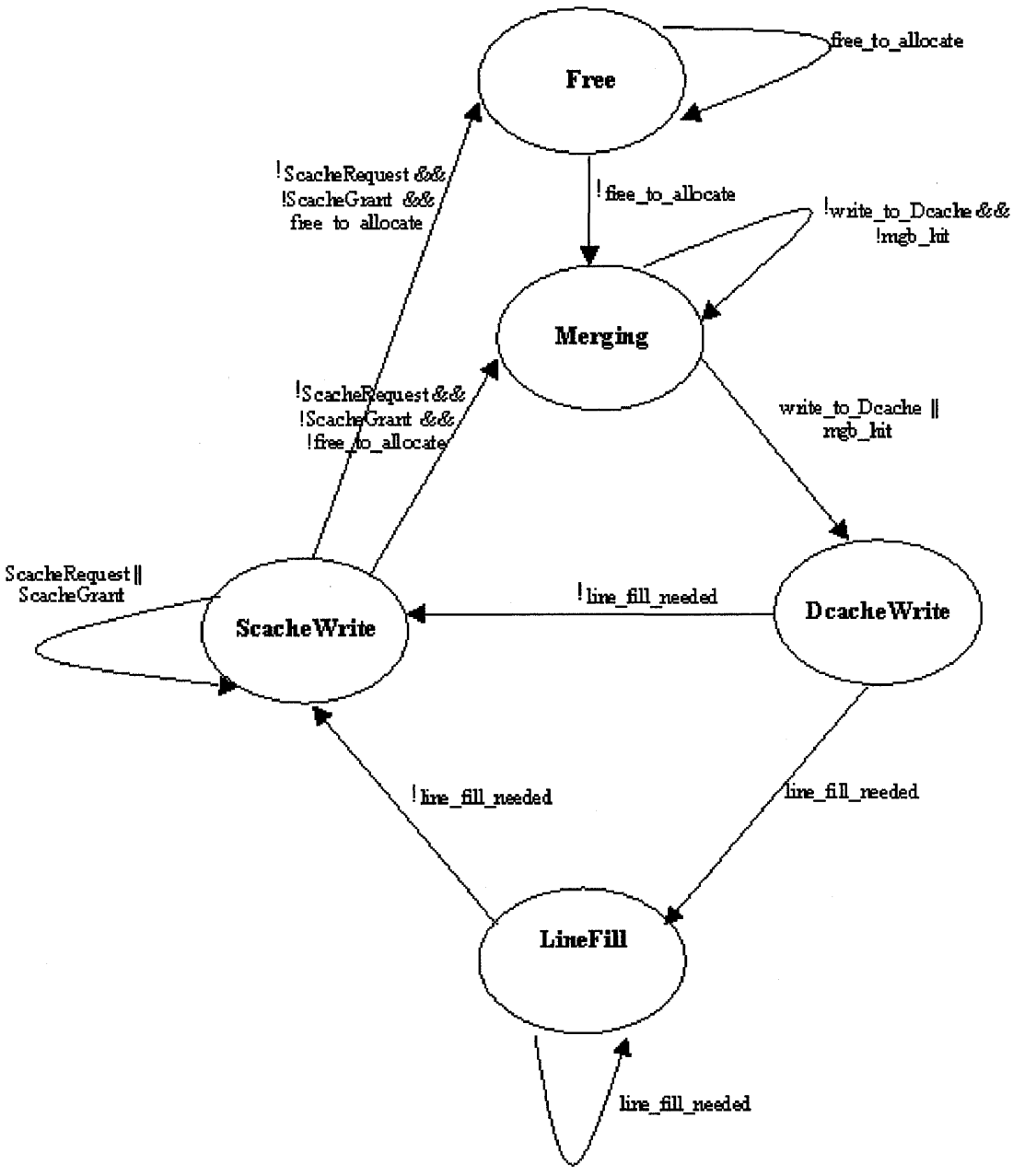
9.14.12 Merge Buffer States

Figure 9–3 is a logical state diagram that illustrates the states through which a Merge Buffer entry progresses. Each of the states are represented as follows:

```
Free      ::=  free_to_allocate;
Merging   :=   ~free_to_allocate && ~write_to_Dcache && ~mgb_hit;
```

```
Dcache_write    :=   mgb_hit;
LineFill        :=   line_fill_needed;
ScacheWrite     :=   ScacheRequest || ScacheGrant;
```

Figure 9–3 Merge Buffer Entry States



9.14.13 Data Array

The data array slice consists of :

- 512 bits of data

- 64 byte_dirty bits for each byte of data

byte_dirty is set when:

store data (from the Store Queue) is being written per P{0,1}be[7:0] and PA[5:3]

It is reset when:

the write through to the Scache (CBox) is complete
or if
a line fill merges with an existing entry

The data is written as shown by the Store Queue. It is also written by the Back End Bus. The data array has 3 write ports (2 write ports from the Store Queue and another write port for the fill data from the Back End Bus). There are 2 read ports (1 for Dcache writes and another for Scache writes).

9.14.14 Address Array

Following are the fields in the Address array:

TBS

9.14.15 Control Section

TBS

9.15 Load Queue

VIRTUALLY IDENTICAL TO Section 9.4.....

The Load Queue, abbreviated as LQ, holds unretired Loads that have been issued to the Mbox. The LQ entries are allocated to the Loads in program order, by thread. The LQ is used to maintain ordering of Loads related to Stores and Memory Barriers. It also is used to re-issue Loads from the Mbox itself, when a Load cannot complete successfully when it is issued from the Qbox.

The LQ has 64 entries and is partitioned equally between threads at run-time. Thus, when a single thread is running, all 64 entries are allocated for that thread; if two threads are running, each is allocated 32 entries . When four threads are running, each thread is allocated 16 entries. When a thread quiesces, it gives up its load queue entries to the other active threads.

Each LQ entry contains the physical address, opcode, INum, a done bit, retry bits and a TBS. The LQ is allocated in program order (by thread) by the Qbox, which assigns LQ numbers (LNums) for all Loads during the Map stage.

If the Load completes successfully, it is marked as done in the LQ. Otherwise, the Load is marked to be retried. The Load may retry due to a cache miss, a bank conflict or may be a class of Load that can only complete at retirement (i.e., I/O Loads which cannot be done speculatively).

Every cycle, the retry logic in the LQ scans all the entries and finds the oldest ready entry (in a given thread). Readiness is defined differently for each type of retry, but generally refers to when the Load can make further progress. The retry logic then sends the Load to the (DIFFERENT) pre-MAF. Retry candidates are chosen from different threads in a round-robin fashion.

The LQ facilitates speculative execution of Loads by allowing Stores to check if a Load younger than it, in program order, may have completed (i.e., the Load returned data before the correct Store data had been sent to the Mbox). When the Store address operation dispatches from the Qbox, it checks the LQ. If a match is found, the oldest Load that matches the Store address is forced to trap. Note that this check is relevant only for Loads and Stores within the same thread.

The LQ also facilitates speculative execution of Loads past Memory Barriers. This is made possible by allowing Stores from the Merge Buffer to check the LQ for possible address matches. In this case, a Store needs to trap a Load from another thread. Note that in this case, up to three Loads can match the Store's address and signal a trap simultaneously. Probes (invalidates) also may force a previously completed load to trap; in this case, upto 4 loads (belonging to different threads) may trap.

LQ entries are deallocated once the Load is past the retire point. The Pbox sends the LQ an INum for each thread that corresponds to the youngest operation that is being retired. All LQ entries that are older than this INum are marked as being deallocated.

The LQ drives a signal to the Qbox every cycle that specifies the youngest Load that may issue out of the Qbox. This signal is based on the youngest Load that is being deallocated and the number of available entries in the LQ.

9.15.1 Load Queue Allocation

The Load Queue has 64 entries that are shared between the currently active threads. Load Queue entries are arranged in blocks of 4 entries (referred to as load blocks). Entries within a block are physically contiguous and are in strict INum order (the oldest is allocated to entry 0 while the youngest to entry 3). Blocks may be dynamically reassigned from one thread to another in order to achieve a balanced sharing. In normal mode of operation, the number of load queue entries allocated to a given thread is shown.

During retries and traps, we need to know the age of the load with respect to other loads in order to pick optimally. Each load block has an young_vector associated with itself, which gives the location (bits correspond to the physical location of other entries) of loads that are older than itself. When a thread is activated (or if a previously quiesced thread wakes up), it starts to take away load blocks from other threads as those threads release their entries (via retirement). Similarly, when a thread quiesces, it releases its entries which are then assigned equally among the threads that are active.

9.15.2 (Age) Young Vector generation

TBS

9.15.3 Load Queue Limit and Block Allocation

TBS

Case 1: New Thread in Machine (or a Previously Quiesced Thread Waking Up)

TBS

Case 2: A Thread Quiesces

TBS

9.15.4 Thread Choosing

TBS

9.15.5 Block Assignment

TBS

9.15.6 Load Issue

TBS

9.15.7 Load Retries

Loads may not complete (return data) after they have been issued. They end up retrying due to the following reasons:

- The block does not exist in the Dcache (dcache miss).
- The block does not exist in the Scache (scache miss).
- Dcache bank conflict
- The data exists in the Store Queue, but the data has not arrived yet.
- The load is to IO space and thus must issue (to the system) only when the load has retired (at retire).

When the load encounters any of the above conditions, it is marked for retry. The load stalls in the Load Queue until its stall condition (retrying condition) has gone away i.e a load stalled on dcache miss, is allowed to retry once the data from the Scache is imminent. However, since there may be more loads ready to retry than there are retry ports (currently ports 0 and 1 are reserved for retries), a picker picks the oldest 2 loads in a TPU (and goes round robin between TPUs).

It is possible that a load may get multiple reasons to retry (bank conflict as well as Store Queue data not available at the same time). However, only 1 retry reason is recorded in the Load Queue. The retry reasons are prioritized in the following order:

1. At retire
2. Scache miss
3. Dcache miss
4. Bank conflict
5. SQA immediate retry

When a load is asked to retry, it sets the retry_status register to record the reason for retry.

9.15.8 Dcache Miss

When a load issuing from the Qbox, misses in the Dcache, it sets its retry_status=dcache_miss. It also sets its retry_ready bit, thus preparing to retry immediately. A picker in the Load Queue picks the oldest 2 loads from the Load Queue and sends the physical address (PA[47:6]) to the Pre MAF. The Pre MAF arbitrates between requests arriving from the Load Queue, the IBox as well the Merge Buffer; once, the CBox accepts the retry request, a bubble request is sent over to the Qbox.

Using the Load Queue to determine the picking in the MAF helps as follows :

- The Load Queue already has a picker that picks the oldest load in a TPU goes round-robin between the different TPUs. The MAF looses any notion of program order and thus the MAF can never pick the oldest waiting load. Furthermore, we need have only 1 picker (not 1 in the Load Queue and another in the MAF).
- We cannot allocate 3 load misses per cycle in the MAF. Therefore the Load Queue acts as a queue into the MAF.
- The Scache loop is shorter than the retry loop. Hence if the Load Queue waited until the MAF picked an address to send through the Scache and then initiated the retry process, it would unduly increase the latency of the load (as shown in the pipelines below).

9.15.8.1 MAF Pick

TBS

9.15.8.2 Load Queue Pick

TBS

9.15.9 Scache Line Miss

The Load Queue schedules the dcache-miss retry, assuming it will hit the Scache. If it turns out that the line does not exist in the Scache, then it sets the retry_status=scache_miss and waits in the Load Queue.

When the CBox receives the data from the system, it sends the fill address and subsequently the fill data to the Mbox. The fill address is allocated in the Fill Buffer. Once the Back End Bus grants arbitration to the Fill Buffer, the fill address is sent to the Dtags, the Merge Buffer, the Load Queue as well as the Fill Buffer.

If a fill dispatching on the Back End Bus matches a Load Queue entry that is stalled on a scache miss (retry_status==scache_miss), then its corresponding retry_ready bit is set.

9.15.10 Load Queue retry - Bank Conflict

Bank conflicts are detected in the A phase of M0. The retry status is written in M1, phase B.

E-stage	M0-stage	M1-stage			
	Bank conflict detected	Write Load Queue Set retry_status=bank_conflict retry_code==bank_conflict	Pick oldest retry	Read Load Queue	Send Bubble Request to QBox

The load may be picked (to retry) as early as M2 and may start to retry (send bubble request) 2 cycles later.

Add SQA immediate retry.

9.15.11 Retry at retirement

TBS

9.15.12 Retry Block

TBS

9.15.12.1 Pick Oldest Retry

TBS

9.15.12.2 Oldest and Next Oldest Retry Chooser

Each bank (of the Load Queue) drives out the silo_id and the lnum of the oldest block as well as an indication

(more_than_1_retry_rdy) that the block has more than 1 entry that is ready for retry. The 2 lnums are then compared. If the older block (depending upon the lnum comparison) has more_than_1_retry_rdy asserted, then the oldest and next_oldest are assigned to the same block.

Once the oldest and next_oldest entries are selected, drive retry_grant_bank to each of the banks. Thus if the oldest and next_oldest are from separate banks then drive retry_grant_bank to both banks; else drive only to one (bank that is selected).

The selection of the oldest and next_oldest and sending back the grant (to disable the load queue entry from bidding again) needs to happen in 1 phase (as shown above).

In the next phase, we read out the inum of the oldest_retry and next_oldest_retry (using the silo_id read out in the previous phase).

9.15.12.3 Thread Chooser

A 4-bit vector last_thd_chosen[3:0] records the last thread chosen (to retry). Priority encode thread_mask[3:0] starting at last_thd_chosen[3:0] to generate thread_choose[3:0].

The Pre-MAF may send block_rty_TPU[3:0] to disable retries from being chosen in a given TPU. This is used in conjunction with TPU_mask[3:0] to choose the next TPU to retry.

9.15.13 Prefetches

Following are the forms of prefetch instructions which arrive on the load port(s):

- LDL r31,(r):if Dcache miss, fetch from Scache/system and install in Dcache (Pref)
- LDQ r31,(r):if Dcache miss, fetch from Scache/system and set LRU to evict (PrefEvict)
- LDB r31,(r):if Dcache miss, fetch from Scache/system in shared state (PrefShared)
- LDW r31,(r)
- LDF r31,(r)
- LDG r31,(r):if Dcache miss, fetch from Scache/system but don't cache in Bcache (PrefDC)
- LDS r31,(r):if Dcache miss, fetch from Scache/system in owned state (PrefMod)
- LDT r31,(r):if Dcache miss, fetch from Scache/system but do not cache in either Bcache or Dcache

Pref	Send out a FetchLine command to the Cbox.
PrefEvict	Send out a FetchLineEvict command. The Cbox treats it the same as a FetchLine however, when the block returns to the MBox, the Dtag sets the LRU bit to evict.
PrefOnce	Send out a PrefOnce command to the Cbox. The Cbox sends out a FetchLine command but does not cache the block in Scache (invalidates
PrefDC	Send out a PrefDC command to the Cbox. The

9.16 Load Traps

TBS

9.16.1 DTB trap

9.16.1.1 Load/store Order Trap

TBS

9.16.1.2 Inval Trap (Traps Due to Probe-invalidates)

TBS

9.16.1.3 MGB Trap (Traps Due To Merge Buffer Dispatches On Back End Bus)

TBS

9.16.1.4 Trap Summary

Table 9–8 shows the trap summary.

Table 9–8 Trap Summary

Trap	Status Bit (Trap_status)	Signalled in Pipe Stage	Trap Condition
DTB trap	dtb_trap	M1	TB_not_present || TB_access_vio
Load-store order conflict	order_trap	M2	TBS
Parity/non-correctable error	Machine_check	M3	(P{0,1,2}ECC_check_result == error_corrected) || P{0,1,2}dcache_parity_error
ECC correctable error	correctable_error	M3	—
Inval trap (also MGB trap)	inval_trap	Back End Bus Tag Read stage	(DC_addr[47:6]==LdQ.PA[47:6])&& ((DC_op==store) && (DC_thread_id != Ldq.thread_id) || (DC_op==inval))
Load Queue not available	—	M0	load_queue_not_avail

9.16.2 Trap Resolution

Up to 4 threads may have their trap bits set in thread_trap[3:0] but we can process only 1 thread at a time. In order to prevent the Completion Unit from advancing the retire point past the trap point, we assert stall_retire_thd[3:0] for each of the threads that have a possible trap.

The Load Queue examines the trap status of the Load Queue blocks belonging to the thread chosen. It finds the oldest Load Queue entry that has its trap bit set. The block is oldest if NAND(younger_vector[i],block_trap) is true.

In M4 phase B, the lnum for the oldest block in each bank is compared. Drive grant_trap_bank in phase B, to the bank that is older. The oldest entry in the block that is granted resets its trap bit (so as not to bid again). The inum of the oldest load is read out of the Load Queue and sent to the Retire/Kill unit.

9.16.3 Thread chooser

Each port records the thread ID in M3 (a 4-bit vector thread_trap[3:0]) if it has a potential trap. thread_trap[3:0] may also be loaded directly by the Load Queue (when an probe invalidation or store dispatch from the Merge Buffer finds a hit in the Load Queue).

The Store Queue sends its trap status (for each of the ports). If either of the threads (on the 2 store ports) match a thread in the thread_trap register, then choose the matching thread. If both threads match, choose one.

If there is no thread match between the store ports and the thread_trap register, then choose (in a round robin fashion) a thread from among the bits set in the thread_trap register. Send the thread ID to the Trap Resolution block in M3 (i.e send the chosen thread at the same time the ECC error is being latched at the Load Queue).

If the Store Queue has traps on 2 different threads, while the Load Queue has a trap on yet another thread, then allow the Store Queue to report its traps (the Load Queue loses its bid to report its trap that cycle). All this is because, the Mbox can report only 2 trap inums per clock.

9.16.4 Kill Bus

The Kill bus (kill_valid,kill_inum[7:0], kill_thread_id[1:0], trap_type[??:0]) is sent to the Retire/Kill unit.

9.16.5 Litmus 1 Handling

If a probe-inval or a store from the Merge Buffer sets the inval_trap, it asserts stall_retire . Once the Retire/Kill unit acknowledges (for that thread) that there are no more pending retired instructions in the pipe, the Load Queue initiates trap processing on that thread. This ensures that a load queue entry may trap only if it hasn't advanced past the retire point. Exact interface signal names are not known at this time.

9.17 Dcache Tags

The MBox contains four tag arrays. These tags describe the contents and state of every line in the DCache. Three of the arrays are at the front end, and are connected to the load ports. One array is at the back end, and is tied to the Back-End Bus.

9.17.1 Front End Tags

The front end tags serve one and only one purpose, namely to indicate whether a load coming in on its associated port is a hit. Tag launch is slated for M0A, and hit determination for M1A.

In accordance with the DCache structure, the tag is 2-way set associative, with 512 entries per set indexed by VA<14:13> and PA<12:6> (or, equivalently, by VA<14:6>).

The data stored in the front end tags are the physical tag, PA<47:13> and valid and fill-in-progress bits. One parity bit protects the tag entry. Parity errors have the effect of forcing a trap. The fill-in-progress bit indicates that the associated tag entry is not yet valid, but will soon be. This allows the Load Queue to initiate a retry for that entry immediately.

The front-end tags have three read ports and one write port. Physically, there are three copies of the front-end tag to provide the necessary number of ports.

To support multiple synonym invalidation, valid bits for all four combinations of VA<14:13> for a given physical index PA<12:6> need to be able to be rewritten simultaneously. It may be convenient to pull the valid bits out of the main tag array to permit this operation.

9.17.1.1 Timing

Table 9–9 show the Dcache front-end tag timing.

Table 9–9 Dcache Front-End Tag Timing

E0		M0		M1	
A	B	A	B	A	B
Receive OP Issue from Qbox	Ebox drives LD Addr	Launch VA into TB, Tag, Stq	Read PA's from tags	Compare PA's with TB; determine DC_Hit	

9.17.1.2 Tag Operations

- Incoming loads on the three load ports are looked up in the tag, using VA<14:6> as the index, compared with the output of the DTB, and return a hit indication, a set selection and a fill-in-progress indication.
- The back end tag may send a new tag entry to be written. Some flight time delay is acceptable, as long as the new tag arrives before the entry is set valid.
- The back end tag may send a new set of valid bits to be written.

The tag needs to support both a read and a write in the same cycle.

9.17.2 Back End Tag

This tag handles stores, synonyms, probes and fills. Because this uses physical addresses, this tag is 8-way set associative, indexed by PA<12:6>. VA<14:13> are concatenated to the set number.

The back end tag must contain, in addition to PA<47:13>, valid, shared and owned bits, and the SCache set in which the cache line resides. Each pair of entries also must contain a set allocation bit, indicating the destination set of the next DCache fill. The tag entry is parity protected.

9.17.2.0.1 Tag Operations

- Start fill.

  When the Cbox MAF accepts a PreMAF miss request, the PreMAF requests a back end tag launch, using VA<14:6>, and reads the set allocation bit. The entry so indexed is then written with the new tag value, including the appropriate ownership and SCache set values. The valid bit is cleared and the fill-in-progress bit set. The set allocation bit is flipped, unless directed otherwise (Prefetch Evict Next). With the exception of the valid bit, these data need not be written immediately, as long as the write occurs before the End Fill operation.

- End fill.

  When all fill data are transferred from the Cbox to the DCache array, the PreMAF requests that the valid bit in the tag be set and the fill-in-progress bit cleared.

- Probe.

  The CBox requests a tag launch with PA<12:6>. If any of the 8 entries so indexed hit, a write cycle is initiated clearing the matching valid and fill-in-progress bits, of which there may be as many as four.

- Store.

  The Merge buffer requests a tag launch with PA<12:6> and also supplies VA<14:13> at the time that the PreMAF accepts a merge buffer evict request. For the two entries indexed by VA<14:13>, return a hit indication. For the other six entries, invalidate entries that hit.

9.17.3 IPRs

Tag operation is controlled by the DC_CTL IPR. Relevant fields include the following:

DC_CTL Field	Description
SET_EN[1:0]	Gates the match lines for the respective sets.
F_HIT	Forces the DC_HIT line.
FLUSH	Clears all the valid bits.
F_BAD_TPAR	Forces bad parity on tag writes.
DCTAG_PAR_EN	Gates parity checking.

9.18 Dcache Array

The Data Cache, or Dcache, is a 64K-byte on-chip data storage. The data are organized in 64-byte blocks, divided into 32 4-byte (longword) banks (virtual address bits 6-2 are used to address the banks). Each bank can accept a read and a write per cycle. Three Address ports are input to the Dcache from the Ebox; three Data ports are output back to the Ebox.

There are three data ports to the Ebox upon which read data are transferred from the Dcache. Since only 1 read is permitted per clock, in the event of a bank conflict, the older load on port 0 and 1 is given priority, followed by the younger load on port 0 and 1, followed by the load on port 2. In the event of a bank conflict, the load is retried out of the Load Queue.

Cache block fills originating either from the CBox or stores in the Merge Buffer, may write upto an entire cache-block (512 bits) per cycle into the Dcache. The write data is accompanied by 64-parity bits, which are stored in the array, as well as 64 dcache_dirty bits, which control the write to the appropriate byte-bank. During a write, the cache index bits are sent from the Back-End Bus, while the set bit is sent from the Dtags.

During a read operation, 3 indexes are presented to the Dcache. After prioritizing between the 3 ports, each bank is selected to drive the load data bus on the corresponding port. Both sets are read out of the Dcache, formatted and sign-extended. Once the Dtag compare as well as the Store Queue check (to see if the Store Queue may drive the data, instead of the Dcache) is done, the appropriate set is selected and sourced onto the Load Data bus.

The parity bits (for each byte) are read out and sent along with the Load Data bus. The EBox is responsible for signalling parity errors.

9.18.1 Read Dcache

The Dcache row index (Indx<12:0>) comprises of VA<14:13> and PA<12:2>. The row index arrives at the MBox in early M0 phase A (or late MY phase B). During phase A, the row address is decoded and possible bank conflicts checked. The array is read in phase B of M0.

The data is formatted in early phase A of M1; the set select (as well as Store Queue hit) is known in phase A of M1. The data is selected and driven to the EBox.

Back-End Bus Pipeline/Dcache Write Pipeline/Load Pipeline TBS.

9.18.2 Write Dcache

Write pipeline is shown above. Write data may originate either from the the Merge Buffer, directly from the CBox (via a latch in the Mbox) or from one of the 4 Fill Buffer entries (for IO and partial fill data).

The fill data and the fill address (VA[14:13], PA[46:6]) busses are sent to the Dcache during the Drv. DC_data phase of the Back End Bus pipeline.

The data is written during phase A of the Dcache Write phase of the Back End Bus pipeline.

9.18.3 Bypass Fill Data

Presently it is thought that the fill data bus (write data) will mux onto the read lines in order to bypass the data to the Load Data bus. This needs to be examined.

9.18.4 Structure

The array is physically structured in 2 halves: left and right. The right half provides the even long words (0,2,4 etc.) while the left half provides the odd long words (1,3, etc). Each half contains both sets A and B. The bits for each set are interspersed (bit0 of set A and set B are adjascent). The LSBs for each of the 4 bytes are grouped together (0,8,16,24) and the corresponding bit position of each of the 8 longwords are kept together as follows:

```
(0,32),(8,40),(16,48),(24,56),(1,33),(9,41) .....
```

The 32 banks are arranged as 8 sets: each set comprises of 4 banks which are interleaved between the right half and the left half. Thus bank 0 (and bank2) is on the right array, while bank 1 (and bank3) is on the left array.

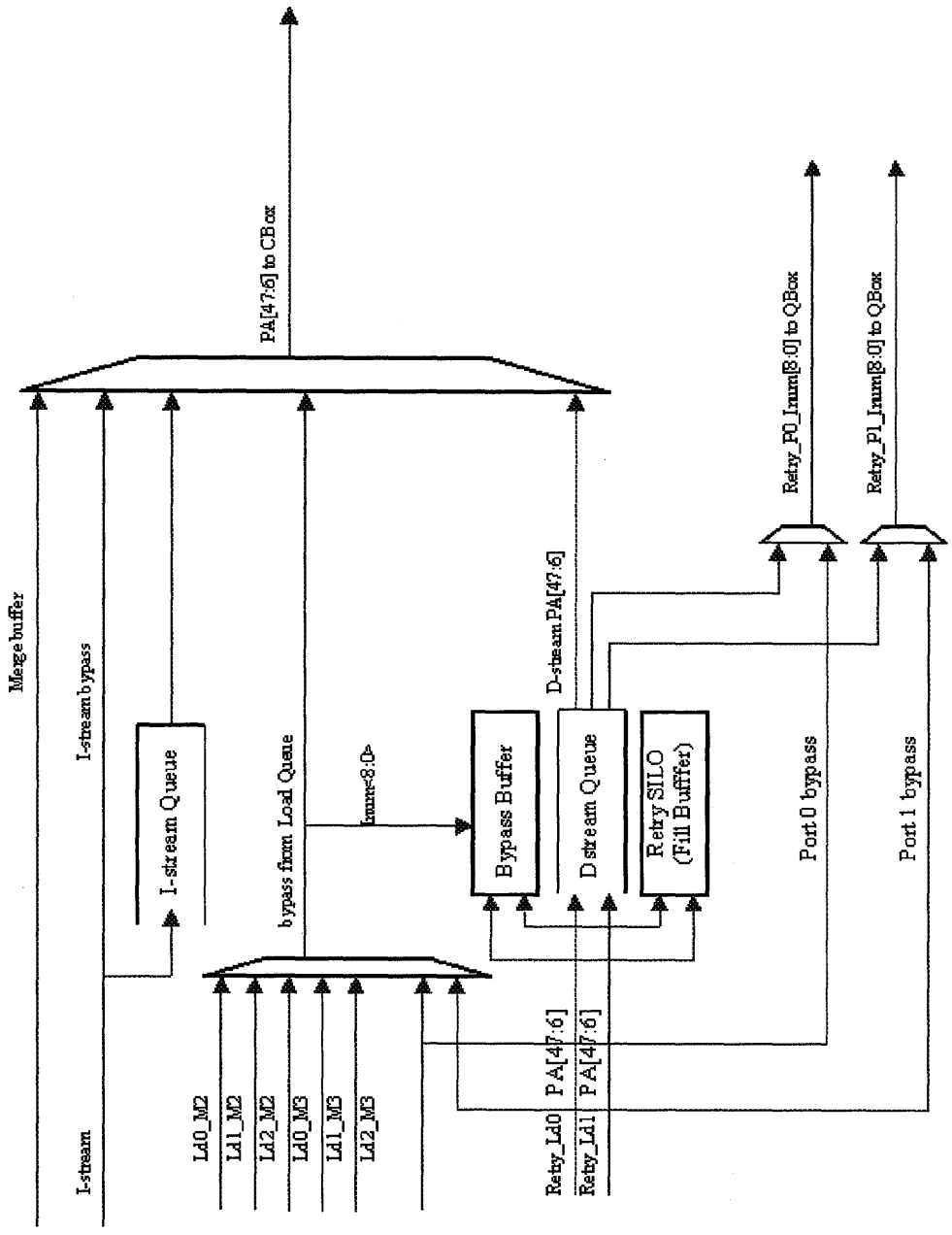
6 sets (3 ports * 2 ways) of differential bit wires (12 wires) or global bit lines enter the DRV section (sense amp). Following the DRV section is the SWP section which does format conversion (byte, word, lword, qword) as well as sign extension. The formatted data for each of the 2 sets (ways) are then muxed with the Store Queue data path and onto the Load Data bus.

The channel in between the 2 arrays (left and right halves) is used to route Index<12:0>.

9.19 Pre-MAF

Figure 9–4 shows the pre-MAF queue.

Figure 9–4 Pre-MAF Queue



The Pre-MAF queue can accept requests every clock from the following sources:

- I-stream
- Merge buffer
- 2 load retries from the load queue

The Merge Buffer requests have the highest priority and are always forwarded to the Cbox (MAF) in the cycle that the request arrives. The load retries are written into the D-stream Queue while the I-stream requests are written into the I-stream Queue.

If the D-stream request queue is empty, newly issued loads (from either ports 0,1 or 2) may be bypassed directly to the CBox, without going through the extra stages of writing into the Load Queue and then being scheduled from it. The bubble requests are conditioned upon the CBox acknowledging that the request is indeed being sent to the Scache. Retry requests from the Load Queue which do not need Scache access, bypass the Dstream Queue and are allowed to send their bypass request directly. The bubble request is arbitrated 3 cycles prior to being sent.

9.19.1 Merge Buffer Requests

Merge Buffer requests are sent via the Back End Bus. Merge Buffer requests are TBS.

Merge Buffer requests are sent to the MAF, without any queuing delays in the Pre MAF. The acknowledgement (ack) is sent directly to the Merge Buffer. If the request was not accepted by the MAF, the Merge Buffer needs to resend the request.

9.19.2 D-stream Queue

The purpose of the Dstream Queue is to buffer requests from the Load Queue enroute to the CBox. Since it takes many cycles to re-read the Load Queue in case the request doesn't get access to the MAF, the Dstream Queue buffers 16 retry requests destined for the CBox MAF.

Requests emanating from the Load Queue have a status bit send_to_scache, implying that the request needs to be sent to the CBox. These are the requests that are enqueued in the Dstream Queue. All others proceed directly via the bypass path shown (Port 0 & 1 bypass), to send the bubble request to the Qbox.

Retry requests coming in, CAM the Fill Buffer (Retry SILO) as well as the Bypass Buffer. If a match is found, the send_to_Scache bit is reset (implying that the request should not be forwarded to the CBox). Requests entered into the Dstream Queue are allowed to send their bubble request only after all preceding entries in the Dstream Queue have sent their bubble request. By disallowing requests which hit in the Retry SILO (Fill Buffer), from proceeding to the MAF, allows the request to send its bubble request paired with another. It also preserves Scache bandwidth. One exception to this are IO requests, which keeps its send_to_Scache bit set, even if a hit is found (in the Retry SILO).

Three cycles after a request is sent to the CBox, an ack is received which implies that the Cbox accepted the request; if the ack is not received, the request needs to be resent. At the time, the ack is received, the PA of the request that was just sent is used to CAM the Dstream Queue; if any entry finds a match, it's send_to_Scache bit is reset (except if its IO).

9.19.3 Killing Retries

The D-stream queue also acts as a staging latch (silo) for retries waiting for Scache access. The kill bus is routed to the D-stream queue and all retry entries need to compare their inum to check if the retry needs to be aborted; if so the entry sets retry_abort. Once the tail_ptr comes to an entry whose retry_abort is set, it suppresses the bubble request. However, the MAF request will still be made.

Entries whose retry_abort bit is set, do not assert block_retry.

9.19.4 I-stream Queue

The I-stream Queue is constructed as a separate queue primarily because of the following reasons:

- Istream requests do not need to CAM previous requests in order to suppress requests to the Scache
- There is no bubble widget for Istream requests.

The I-stream queue is a 16-entry-deep FIFO. When the Istream queue is almost full, the pre-MAF asserts pmf_full to prevent additional entries from being sent.

9.20 Mbox Back End Bus

TBS

9.21 Internal Processor Registers

The Mbox Internal Processor Registers (IPRs) provide visibility and control for processor-specific operations in the Mbox. These include handling Translation Buffer misses, along with various kinds of Dstream faults, and enabling and disabling various parts of the box, generally for test and reset use.

Mbox IPRs are described in Section 16.3.

All IPRs, with the exception of DC_CTL, exist on a per-TPU basis. That is, any read to a readable IPR returns data specific to that TPU. Any explicit write to a writable IPR takes effect when the write retires in that TPU. However, many IPRs control chip-wide state. Thus, retiring an IPR write can affect state that is visible to another TPU. For example, the DCache and DTB are shared resources. DC_CTL is a chip-wide IPR, it is used to control the shared DCache.

In addition to the general 21464 treatment of IPRs, the Mbox applies the following specific rules to its IPRs;

- TB entries must be usable speculatively. For further information, see the Translation Buffer document. IPRs that write the DTB, including invalidates, exist as a pair of IPRs (such as DTB_PTE0 and DTB_PTE1). To perform a write, both members of the pair must be written in adjacent slots in the same map block, slotted to the strong load ports, with the IPR1 following the IPR0. The IPR1 operation is a long latency operation which causes a bubble back, allowing the QBox to release subsequent DTB writers.

- All other MBox IPRs only take effect on retire, and must be protected with an IFETCHB instruction. The MBox IPR write port is connected to the LD_ADDR[0]<63:0> bus. In order to write an MBox IPR, two identical HW_MTPR instructions must be issued in adjacent slots in the same map block, slotted to the strong load ports. This ensures that one of the two writes will travel down the LD_ADDR[0] bus. The other will go down LD_ADDR[1] and will be ignored.
- The MBox IPR read port is connected to the Ebox IPR read bus, which operates as a 5-cycle multimedia instruction slotted through a weak load picker. Thus, all MBox HW_MFPR instructions must be issued on the weak load port.

In addition to the MBox IPRs, several other box IPRs also interact with the MBox. They are handled as follows:

- The CBox IPRs are mapped into physical memory. Access to them is via IO Loads and Stores.
- The IBox IPRs are accessed through the load address and data busses they same way JSR call and return addresses are. In particular, the JSR call and HW_MTPR path is connected to the LD_ADDR[2]<63:0> bus. This connection is made at the Ebox end of the bus, adjacent to the drivers. These instructions must issue on the weak load port. The JSR return and HW_MFPR path is connected to the Ebox IPR read bus. These operations must be slotted as 5-cycle multimedia operations on the weak load port.

9.21.1 Implicitly Written IPRs

There are two groups of Mbox IPRs that are written implicitly, that is, by other than an HW_MTPR instruction. Implicitly written IPRs require special and careful handling, as documented by the Qbox. The first group consists of the DC_STAT IPR, which is written when any of several asynchronous events happen on a DCache fill. DC_STAT is an implicitly event-written IPR, in that we do not attempt to associate its writing with any particular instruction. Also, events set bits in DC_STAT which are not cleared even if the instruction (indirectly) leading to that event turns out to have been killed.

The second, and more complex, group consists of the VA, VA_FORM and MM_STAT IPRs, which, for simplicity, will be referred to as the MM_STAT IPR set, as all three share the same update criteria. The MM_STAT IPRs are implicitly written by any instruction causing a Dstream fault leading to any DTB_MISS or DFAULT PALcode entry point. Furthermore, the set of MM_STAT IPRs read by a particular entry to PALcode must correspond to the instruction that generated the disruption leading to that PALcode entry. This means that if an older disruption overshadows a younger one, the older disruption must overwrite the MM_STAT IPRs. Note that poisoned instructions must never generate faults.

This may be expressed, in a TPU-centric view, as assigning an INum to each (TPU-specific) MM_STAT IPR set, and only allowing a faulting instruction to update the set if all of the following conditions hold:

- The instruction is older than all faulting instructions for the same TPU issued in the same cycle in other memory pipes.

- The instruction is older than any instruction in this TPU that decided in the previous cycle that it will write MM_STAT.
- The instruction is older than the INum, if any, currently associated with this TPU's MM_STAT IPR set.

If this is the case, the instruction's particulars and INum are written to the MM_STAT IPR set. The MM_STAT IPR set INum is cleared whenever that INum or an older INum is killed.

A special case condition is that the LD_VPTE instruction, which is only executed within the DTB single miss flow, writes only its INum, and not its particulars, to the MM_STAT IPR set. The reason is that the LD_VPTE disruption handler deals with correctly fixing up the underlying memory operation that caused the DTB single miss immediately preceding the LD_VPTE, rather than fixing up the LD_VPTE itself, which was merely the first attempt to deal with the DTB miss, and not anything interesting of itself. Thus, the MM_STAT particulars from the original disruption must be preserved.

The INum of the LD_VPTE must be written to the MM_STAT set to ensure that we do not speculate all the way through a DTB single miss and into another Dstream fault while a LD_VPTE double miss or trap is pending. The particulars of the original single-miss entry will still be preserved at the time the LD_VPTE traps, as all subsequent memory operations are dependent on the DTB writer block issue, which is in turn data dependent on the LD_VPTE. The Mbox must detect LD_VPTE faults before this protective window expires to avoid having younger memory operations overwrite the MM_STAT particulars before the LD_VPTE disruption has a chance to write its INum to the MM_STAT set.

10
Internal Ring Bus

This chapter is to connect the Cbox, Rbox, and Zbox chapters.

11
Second-Level Cache and Controller (Cbox)

The Sbox and the Cbox contain the onchip 3 MB six-way set-associative second-level cache (the Sbox) and the control of this cache (the Cbox). Additionally, the Cbox, in conjunction with the Mbox and Zbox, implements the cache coherent, distributed-shared memory system.

11.1 Cbox Overview

The Cbox is divided into two logical and physical partitions:

- CS — the "Scache controller" partition. The CS manages the Scache pipeline.
- CF — the "fill datapath" partition. CF handles data flowing to and from the Cbox

The following sections comprise the CS partition.

Name	Mnemonic	Description
Internal probe queue	IPQ	Sixty-four entry FIFO for holding MAF indexes that require internal probe processing
Miss address file	MAF	Holds requests from the local processor until satisfied, and holds probes and forwards from remote processors.
Probe queue	PRQ	Thirty-two entry FIFO for holding probes and non-block responses that are waiting for access to the Scache pipeline.
Response queue	RSQ	FIFO to hold VAF indices that have not yet been delivered to system.
Retry queue	RTQ	Sixty-four entry FIFO for holding MAF indexes of Scache transactions that must execute through the Scache pipe again due to an error or bank conflict.
System interface	SYS	Connects the Cbox to the Rbox and Zbox via 21464's internal ring bus.
System request queue	SRQ	Stores MAF indices requiring a system request. Tracks number of outstanding system requests for a given Scache index.
Test structures	TTQ	
Victim address file	VAF	Holds responses being sent back to the system either as displacement victims or in response to system probes.
Config and status registers	CSR	Holds the Cbox CSRs.

The following sections comprise the CF partition.

Name	Mnemonic	Description
Data buffer muxes	CF_DBM	
Fill data buffer	CF_FDB	Buffer that holds fill data from the Zbox, destined for the Rbox or Cbox.
Fill data logic, ecc	CF_FBE	
Rambus input	CF_RBI	
Rambus output	CF_RBO	
Victim data buffer	CF_VDB	Buffer for victim data from the Cbox, destined for the Zbox or Rbox.

When the processor (Ibox or Mbox) needs access to a block of data that it does not have, it makes a request to the Cbox. If there is a copy of the requested block in the second level cache (Scache) Cbox returns that copy. Otherwise Cbox will make a request to the system to get a copy of the requested block. In multi-processor systems, to keep memory coherent, Mbox must be sure that this processor is the only processor with a copy of that block (exclusive) before it writes to it. If the block Mbox wants to write resides in the Dcache or the Scache but is marked as shared then there may be other processors with copies of that same block. Cbox must make a request to the system to obtain an exclusive copy.

When a stores retires it is first written into the Mbox merge buffer. The merge buffer merges stores to the same block before requesting an exclusive copy of the cache block from the Dcache, Scache or system. Once an exclusive copy has been obtained, the store data in the merge buffer merges with fill data returned from Dcache, Scache or system. The complete cache block is then written-through to Dcache and Scache simultaneously.

The system responds to requests for cache blocks by returning a copy of the block with state indicating if you have a shared or exclusive copy and if the block is dirty or not. The returned block is filled into the Scache and also sent back to the requester (Ibox or Mbox). Filling a block can cause an existing Scache block to be displaced and sent back to the system as a victim.

To process a request for a copy of a cache block, the system must be able to determine where copies exist. A directory is used to hold this information (see ?). The system will ensure that the requester gets the most up-to-date copy of the block, and if requested exclusive then the system will initiate the invalidation of copies residing in processors. To do this the system sends probes to processors holding copies of the block. The probes ask the processor to forward its copy to the requester and mark its copy as shared and/or invalidate its copy.

In summary, the Cbox major features are:

- Up to six outstanding requests to the same Scache index. (By comparison, the 21364 can have one outstanding request to a given Scache index .)
- 64-entry MAF
- 64-entry VAF
- Fills entire cache line (512 bits) to the Mbox and Ibox per cycle. (By comparison, the 21364 fills 128 bits per cycle.)

11.2 Sbox Overview

The Scache has the following features:

- 3 MB, six-way set associative
- Physically indexed, physically tagged
- 16 banks, with one read/write port per bank
- QUAD-word (64 bits) writeable, single-bit ECC correction on tags and data. Double-bit ECC detection on tags and data
- LRU replacement

11.3 Scache Control — the CS Partition

The following sections describe the Scache control logic — the CS partition.

11.3.1 Overall Pipeline Flow

The overall pipeline diagram for the Cbox is shown in Table 11–1 and described in Sections 11.3.2.3.1 through 11.3.2.3.7.

Table 11–1 Cbox Pipeline Stages

Internal Probes (IPQ)	Retries (RTQ)	LRUevict/BLK*[1] (SYS)	Transaction Type (Source)	Scache Pipeline States															
				Arb	MAF	Scache Tag Pipe						Scache Data Pipe							
				C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	
			Arb	MAF	Tag Launch	Read Tag	Tag Compare	Read Data Write Tag	Set Select	Drive Data	Fill Bus	WR 1	WR 2	WR 3	WR 4	WR 5	WR 6	C15	C16
Read PA & MAF state. Bank conflict check.	Arbitrate for Scache pipe		Read PA & MAF state. Bank conflict check.	Read PA & MAF state. Alloc VAF entry.	Compute new MAF state.	Write new MAF state. Read LRU/tags	Victim set select.	Read data. Victim tag ECC.		Send Victim addrr and VDB. idx to Mbox. Write VAF.			Victim data on the fill bus. Victim data correction. CAM MAF with victim PA.	Write VDB.		Wr ite VAF. Write RSQ.			
Compute new MAF state. int_probe_ack			Compute new MAF state[2]. retry_ack	Read PA & MAF state.	Compute new MAF state.	Write new MAF state.			Write LRU/tag.		Send fill addr/ tag to Mbox.			Fill data on the fill bus.			Decrement inflight count		
Write new MAF state. Read tag.			Write new MAF state. Rd tag.																Write data.
Tag ECC detect. Set select.			Tag ECC detect. Set select.																
Tag ECC correct. Writetag. Read data.			Taf ECC correct. Write tag. Read[3] data.																
Send probe addr/tag/cmd to Mbox.			Send addr/tag/ cmd to Mbox.										Fill data on the fill bus.						

Table 11–1 Cbox Pipeline Stages (Continued)

Transaction Type (Source)	Scache Pipeline States																	
	Arb	MAF	Scache Tag Pipe						Scache Data Pipe									
	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14			
Arb	Arbitrate for Scache pipe																	
MAF		Read PA & MAF state. Bank conflict check. Alloc VAF entry.	Tag Launch	Read Tag	Tag Compare	Read Data Write Tag	Set Select	Drive Data	Fill Bus	WR 1	WR 2	WR 3	WR 4	WR 5	WR 6	C15	C16	
S2D* (PDQ)			Compute new MAF state. probe_ack	Write new MAF state. Read tag.	Tag ECC detect. Write tag.	Tag ECC correct. Write tag. Read data.		Send add/tag/cmd to Mbox.			Fill data on the fill bus.			Decrement inflight count.				
Forwards (PRQ)		Alloc/merge MAF entry. Read MAF states Bank conflict check. Alloc VAF entry.	Compute new MAF state. probe_ack.			Tag ECC correct. Write tag. Read data.		Send probe addr/tag/cmd to Mbox			Fill data on the fill bus[4]. ECC correction.	Write VDB.						
SharedInval (PRQ)						Tag ECC correct. Write tag.												
Early Warn[5] (SYS)		Read PA.						Send addr/tag to Mbox.										

Table 11–1 Cbox Pipeline Stages (Continued)

Transaction Type (Source)	Scache Pipeline States																
	Arb	MAF	Scache Tag Pipe						Scache Data Pipe								
	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14		
	Arb	MAF	Tag Launch	Read Tag	Tag Compare	Read Data Write Tag	Set Select	Drive Data	Fill Bus	WR 1	WR 2	WR 3	WR 4	WR 5	WR 6	C15	C16
Mbox Req (Mbox req hits Scache)	Arbitrate for Scache pipe	Allocate/merge MAF entry. Read MAF state. Bank conflict check.	Compute new MAF state. miss_ack.	Write new MAF state. Read tag.	Error detection. Set select.	Tag ECC correct. UPdate LRU. Read data.		Fill addr/tag/cmd to Mbox.			Fill data on fill bus.						
Mbox Req (Mbox req miss Scache)							Send MAF index to SRQ.	Miss addr/cmd to Mbox. Write SRQ.	Read PA from MAF.	Send to SYS.	Drive miss request on ring.						
MGB write_thru (Mbox)										Mbox sends wr_thru.		Write VDB. Index conflict check.	wr_thru ack			Write data.	

1 Block responses and I2DResponses require two Scache transaction. The first transaction is the LRUEvict, which extracts the victim data, and the second transaction is the fill. These happen atomically. Timing shown here assumes that the extracted LRUvictim is a) coherent, and b) we don't need MGB to write-thru.

2 The only retry action that updates the MAF state is STODFAIL. A retrying STODFAIL can set MAF.need_sys_req.

3 If the retry is due to a data ECC error, the Scache tag has already been updated and the retry must not change the tag state again.

4 Cbox sends the cache block to Mbox if Scache is to keep a copy of the cache block *Shared* after the probe.

5 EarlyWarn does not set MAF.sc_inflight.

11.3.2 Miss Address File — the MAF

11.3.2.1 Overview

The miss address file or MAF, is the major control structure in the Cbox and is responsible for tracking outstanding miss requests to the system. The MAF is a 64-entry associative memory, with control logic for managing the Scache pipeline. Note that the number of MAF entries (64) corresponds to the number of misses (where a miss may be a d-stream cache-block request, i-stream cache-block request, ownership-only request,

etc) that a processor may have in-flight at any one time. Allowing several simultaneous outstanding misses is critical to keeping a wide-issue superscalar machine like the 21464 fed. By way of comparison, the previous-generation Alpha processor (the 21364) has 16 MAF entries.

11.3.2.2 Principle of Operation

<block diagram needed here>

MAF operation is initiated by 3 major classes of operations:

1. Requests from the core
2. Fills/responses from the system
3. Probes from other processors

11.3.2.2.1 Requests from the Core

The Mbox PreMAF and Mbox MGB may deliver a total of 1 request per cycle to the MAF. The Mbox delivers the physical address and other request state to the MAF. The MAF firstCAMs the PA against existing MAF entries to check if there already is a MAF entry with thisphysical address. If there is, the MAF attempts to merge the new request into the existing MAF entry. If there is no CAM match, a new MAF entry is created. Note that along with merging or creating a new MAF entry, the request is launched into the Scache pipeline.

11.3.2.2.2 Fills/Responses from the System

If a miss request from the core does not find the data (or the required cache state) in the Scache, a request is launched to the system. The system will respond with the data and the new cache state. This response from the system carries with it the MAF index number of the corresponding request. When the fill arrives from the system, the MAF index is used to access the correctMAF entry, extract the relevent information, and perform the fill.

11.3.2.2.3 Probes From Other Processors

Probes (Forwards, Invals)

Probes, like Mbox miss requests, may or may not find that the MAF already has an entry with the probe physical address. Therefore, an incoming probe first CAMs the MAF. If a MAF entry already exists, the probe uses this MAF entry, and launchs into the Scache pipe. If a MAF entry does not exist for this physical address, one is created, and the probe launches into the Scache pipe.

11.3.2.3 MAF Pipeline Timing Diagram and Pipeline Overview

Table 11–2 shows the MAF pipeline timing diagram.

Table 11–2 MAF Pipeline Timing Diagram

CZ	C0	C1	C2	C3	C4	C5	C6	C7
PreArb	Arb	RD/CAM PA	Scache pipe CTRL	Write new MAF state				Fill CTRL

11.3.2.3.1 CZ, C0: MAF Arbitration Logic

The MAF arbitration logic selects one transaction each cycle to launch into the Scache pipe. This arbitration is split across two stages: CZ and C0. CZ is called the PreArb stage and C0 is the Arb stage. Transactions arbitrate in the PreArb stage with the following priority:

1. Retries from RTQ
2. Internal probes from IPQ
3. Probes/non-block responses from PRQ
4. Mgb requests from Mbox

The winner from this stage is latched and sent to the C0 arbitration stage, where the Cbox arbitrates with the following priority:

1. System fill from SYS
2. PreArb winner
3. Mbox PreMAF request from Mbox

Note that the logic is designed to give the Mbox PreMAF requests (such as the L1 Load Miss) as low a latency as possible.

11.3.2.3.2 C1: MAF Bank Conflict Detection Logic / MAF CAM / MAF RD

The Scache tag array (STAG) and Scache data array are large memory structures. The pipeline diagram in Table 11–1 shows that the arrays are read and written in different cycles, thereby requiring multiple ports. Multiporting an array as large as the tag or data array would result in a prohibitively large structure. To avoid this constraint, the arrays are banked into individual, smaller arrays.

Specifically, there are 16 banks for the Scache tags and 16 banks for the Scache data, with each bank having a single read/write port. The low four bits of the physical address (PA<9:6>) give the bank number. Although single transactions are launched into the Scache pipe each cycle, bank conflicts can still arise, and are managed with the following logic.

Consider the Scache tag array first (STAG). The STAGs are read in cycle C3 of the Cbox pipe and are written in cycle C5. Recall that the STAG array has only a single merged read/write port per bank. Table 11–3 illustrates how a bank conflict can occur:

Table 11–3 Scache Tag Array Bank Conflicts

Cycle Number → Transaction ↓	0	1	2
SharedInval	RD tag		WR tag
MissReq1		RD tag	
MissReq2			RD tag

In cycle 3, the SharedInval transaction is trying to write the STAGs at the same time the MissReq2 transaction is trying to read them. If the SharedInval and the MissReq2 have the same PA<9:6>, then we have a bank conflict. This situation also applies to the Scache data array.

The bank conflict detection logic in the CS_MAF is responsible for detecting situations like the one above, and rejecting the later arriving transaction to prevent the bank conflict. (Exceptions to this are described in Section 11.3.2.3.3.) This is accomplished as follows:

For every transaction that enters the pipe, we note when it wants to access the STAG (C3, C5) and/or SDATA (C5, C15) arrays.

We check this against what is already in the pipe. For instance, if the particular transaction wants to read the STAGs (C3), we check if we currently have a transaction in the C3 stage that is going to write the STAGs in C5. If so, the incoming transaction is either NACKed (Mbox) or placed in the RTQ (RTQ, IPQ, PRQ).

{ SHOULD A TABLE BE PLACED HERE OF EACH TRANSACTION TYPE ALONG WITH WHEN THEY READ/WRITE TAGS/DATA ? }

11.3.2.3.3 Exceptions

Fills (and LRUevicts) are special transactions. They are never stalled, rejected, or retried (but see section on hiccup). If the BCL detects a bank conflict for a fill or LRUevict in C1, the transaction which IS ALREADY IN THE PIPE AND IS CAUSING THE BANK CONFLICT is "preempted".

The unlucky transaction is placed in the RTQ for later execution, and the fill procedes merrily along.

11.3.2.3.4 C1: MAF CAM / MAF RD

In the C1 pipe stage of the Cbox, we either CAM the MAF with the incoming transaction's physical address or we read the MAF entry specified by the incoming transaction. The state at the corresponding CAMed or read entry is read out and is delivered to the C2 stage of the pipe. There is a single CAM port on the PA<47:6> stored in the MAF, which is shared between:

- Mbox requests

  Incoming Mbox requests CAM against the entries in the MAF. If there is a MAF entry with a matching PA, we attempt to merge the incoming Mbox transaction with the matching MAF entry.

- Probes (such as Forwards and SharedInvals)

  These CAM the MAF to discover if there is an outstanding request to the same address.

- Victims

  Victims on their way to the VAF CAM the MAF to determine if they are coherent.

11.3.2.3.5 c2: MAF logic

The C2 stage of the MAF is the most complex. Here, based on the incoming transaction and the state read from the corresponding MAF entry, if any, we compute the "next state" of this MAF entry, and required outputs for subsequent stages. The logic involved is too complex to be discussed in detail here; it will be covered later in this chapter when we discuss the flows for each transaction type.

11.3.2.3.6 C3–C6: Scache Tag Access

In these cycles, the physical address is delivered to the Sbox and the tag state is looked up. These cycles technically belong to the Sbox. No MAF logic runs here.

11.3.2.3.7 C7: Fill Pipe Control

In cycle C7, the result of the STAG lookup is latched in the MAF, and, based on the tag state and the transaction state, various commands are delivered to the Mbox and to the VAF. Again, the details of this logic will be discussed later.

11.3.2.4 Contents of MAF Entries

Table 11–4 shows the contents of each MAF entry.

Table 11–4 Contents of Each MAF Entry

Contents	# Bits	Ports	Description
I. Physical address bits			
miss_pa<47:6>	42	1 RD/WR, 1 RD, 1 CAM	I/O space:PA<47> =1.
miss_tpu<3:0>	4	1 WR, 2 RD	Thread processing unit, merged
miss_ifill_ptr<4:0>	5	1 WR, 1 RD	I fill buffer index.
II. I/O request fields			
mgw_closed	1	1CAM, 1WR	I/O merge window is closed.
io_mask<7:0>	0	Stored in a separate table indexed by the TPU	I/O byte mask.
io_size<1:0>	0		I/O size.
QW_addr	0		RdIO or WrIO
III. Request type			
i_fill_ena	1	2RD, 2WR	I-stream fetches (I-demand/I-prefetch).
dc_req_type<4:0>	5	2RD, 1WR	Request type (LD, ST, STx_C, Prefetch Scache, prefetch mod)
IV. Request state			
valid	1	Multi-port	MAF entry is valid.
sc_inflight	1	2Rd, 2WR	Has inflight miss/fill/probe in the Scache pipeline.
sys_cmd<2:0>	3	3 RD, 1~2 WR	System request command.
need_sys_rqst	1	2 RD, 1 WR	Waiting for the system request launch.
sys_inflight	1	2 RD, 1 WR	Has an inflight system request.
IV. Coherence state bits			

Table 11–4 Contents of Each MAF Entry

Contents	# Bits	Ports	Description
coherent	1	1RD, 1WR	Coherence state
cohr_cnt<5:0>	5		
timer_on	1		
has_int_probe	1		
VA<15:14>	2		
Inval_seen	1		
victimize	1		
vic2shr	1		
Invalidate	1		
evict_next	1		
MB_retired<3:0>	4		
Total	~83		

Notes:

Physical Address

- The physical address field PA<47:6> of the MAF needs:

  CAM ports (1):

  – Allocation and merging of Miss from Mbox.

  – Probe processing CAM.

  – Victim processing CAM.

  – Since probes and Scache victims are infrequent, we share 1 CAM port for all three functionality with the priority:

    Victim CAM.

    Scache pipe (Probe CAM and Miss CAM).

- Write port (1): MAF allocation for new Misses.
- Read ports (2):

  – Scache pipe (Blk*, Retries, ShrToDirty*, *Req, and *Forward).

  – System request.

- Write and read are mutually exclusive and we share a single port for Read & Write.
- The current proposal is to have one RD/WR port and one RD-only port.
- A MAF entry may be originated from more than one thread due to merging. When we merge MissReqs from Mbox, we must merge the thread processing unit of the first requester. Mbox does not merge retires and I/O requests across threads. Cbox does merge MissReqs across thread and preventing Cbox from mergeing I/O request across threads must be done in Mbox (i.e. close the merge window first).

When a Shr2Dirty[STC]Req fails:

- If the MAF entry has a I-miss or LD, then send a ReadReq.
- If the MAF entry has a ST, then send a ReadModReq.
- If the MAF entry has only Stx_C, then send no system request.

Since we can have only one I/O request bidding for the system request pipe at any given time, we will have a small structure to store I/O specific fields.

The MAF.sys_cmd<2:0> is set if a system request is needed after looking up the Scache tag.

- If a new Miss gets merged before the system request is sent out (i.e. MAF.need_sys_rqst = 1), then we change the system command.
- If a SharedInval hits a ShrToDirty, we do not need to change the system command to a ReadMod since the ShrToDirty gets forwarded.
- If a SharedInval hits a ShrToDirtySTC, we may reset the Stx_C bit.

Coherent = (have_max_coh = 1 & timer_running = 0).

11.3.2.5 MAF Allocation/Merge/Retry

- Overview and working assumptions
  - The MAF accepts one Miss request per cycle from the Pre-MAF (Pre-MAF).
  - There is ONE MAF entry for a cache block.
  - There is one MAF entry that has the merging window open for an I/O block.
  - No-cache pre-fetches from Mbox will follow the same path as for the regular loads miss but the pre-fetch block doesn't get written to the Scache. We must make sure the block is not a ExclCln or Dirty.
  - Miss Request Inputs
  - I/O requests never ask for the write permission.
  - Ibox never asks for the write permission of a cache block.
- Upon receiving a new Miss request from the PMF, the MAF determines whether to
  - Reject the Miss request.
  - Allocate a MAF entry for the Miss request.
  - Merge the Miss request.

MAF Full	Scache Pipe Available	MAF CAM Available	Address Match	sc_inflight	Action
1	X	X	X	X	Reject
X	0	X	X	X	
X	X	0	X	X	
0	1	1	1	1	
0	1	1	1	0	Merge
0	1	1	0	X	Allocate

  - Eviction requests and ChgToShared requests from Mbox need a VAF/VDB slot. Can we just set the MAF.victimize* bit for the request rather than see if VAF slot is avaialble before deciding whether to accept the request. Then take the same flow as MAF.victimize path?

- Reject: The MAF asks the PMF to retry the Miss request
  - If Cbox can't service the Miss request.
  - Since Mbox filters multiple Miss requests to the same cache block, the merits of merging Miss request when the MAF is full seems small. Hence the MAF will rejects new Miss requests when the MAF is full.
  - The Pre-MAF continuously retry the failing requests which minimizes possible thread starvation for access to MAF entries.
- Allocate: The MAF allocates a new MAF entry for a Miss request
  - The MAF returns the ACK along with the MAF index to Mbox so that Mbox can pull a Qbox bubble.
  - The new Miss request enters the Scache pipe except for I/O requests. I/O requests enters the Scache pipe (or goes directly to the CRQ?) only if the merging window is closed (i.e. m%io_ok_to_send = 1).
- Merge (I/O space)
  - I/O requests from the same thread gets merged in Mbox but Mbox uses the MAF CAM port for the PA compare.

    I/O requests to the same block get merged in the MAF if the merge window is open.

    The MAF may have more than one MAF entry for the same I/O block (e.g. I/O byte reads) but only one has its merge window open.

    I/O request whose merge window is closed (m%io_ok_to_send = 1) enters the Scache pipe (or system request pipe?).

    The merging windows for I/O requests are managed by Mbox.
  - We may have only one I/O request per thread waiting for the system launch.

    The first four VDB entries are reserved for WrIO data block.
  - Mbox does not send multiple I/O requests to the same block from different threads unless the merge window is closed to prevent I/O merging across threads.
  - All I/O merging rules conforming the SRM are handled in Mbox.

MAF States				
sc_inflight	**need_sys_rqst**	**sys_inflight**	**Action**	**Notes**
1	0	0	Reject	Has in-flight miss/probe/fill in the Scache.
0	0	0	Merge Request enters the Scache.	No outstanding request.
0	1	0	Merge Request does not enter the Scache.	Bidding for the system request pipe.
0	0	1		nflight request in the system.

MAF States			
1	0	1	Merge[1]
1	1	0	Merge[1]]
0	1	1	Must not happen
1	1	1	Must not happen

[1] This can happen since we delay clearing of the sc_inflight bit to give Mbox sufficient time to consume fill blocks. But to minimize the system fill latency we may send a system request if necessary before resetting the sc_inflight bit.

- Merge (Memory space)
  - Mbox filters the most of multiple Miss requests to the same cache block by CAMing the fill buffer.

    Filtering of multiple Misses to a cache block is to conserve the Scache bandwidth.

    The MAF merges multiple Miss requests to a cache block if they didn't get filtered in Mbox.
  - For merged Miss requests, the MAF

    Returns the merged MAF index to the PMF.

    Sets the appropriate control flags for the merged MAF entry.

    If a non-no-cache prefetch request merges onto a no-cache prefetch, we clear the bit.
  - We merge STx_C across different threads but only one thread will succeed depending on the order of the retry in Mbox.
  - Since the I-fill buffer index may get reassigned to a new miss in the Ibox, we must not fill the Ibox more than once with the same cache block. Resetting the miss_icache bit after the block has been delivered to the Ibox prevents the filling the same block twice.

    After Scache tag return.

    After System fill.

11.3.2.6 MAF Deallocation

- I/O Read can be deallocated when the requested block is returned or NXMResp is received.
- I/O Write can be deallocated when the WrIoAck is received. When we receive the WrIoAck, we also need to notify the Mbox so Mbox can retire the MB.

pa<47>	*_inflight_in_sc	need_sys_rqst	inflight_in_sys	coherent	victimize	vic2shr	Notes
1	0	0	0	X	X	X	IO request
0	0	0	0	1	0	0	

- If we receive NXMResp, we save the PA & ... and Mbox will trap. We can de-allocated the MAF index once we notify the Mbox and save necessary information into the error status register.
- If the MAF entry has a victim waiting to become coherent in the victim buffer, we must clear the blockage when the cache block becomes coherent even before we de-allocate the MAF entry.
- If MAF.victimize, MAF.vct2shr, MAF.cvt2inval, MAF.cvt_inv_if_shr bit is set, then we need to perform the Scache tag update before we de-allocate the MAF entry.

11.3.3 RSQ

11.3.4 Internal Probe Queue — the IPQ

The internal probe queue or IPQ is a 64-entry FIFO for holding "internal probes". An internal probe is a special Scache transaction that either invalidates or victimizes an Scache block.

Principle of Operations

Internal probes can be created by the following three transactions:

1. Cache manipulation instructions from the Mbox (CCB, ECB instructions).
2. Block response from system arrives and one of the following is set:
   - MAF.victimize
   - MAF.vict2shared
   - MAF.invalidate
3. Non-block response from system arrives and one of the following is set:
   - MAF.victimize
   - MAF.vict2shared
   - MAF.inval_seen

Other than the cache manipulation case, internal probes arise because the network is not ordered. The following sequence illustrates the generation of an internal probe:

1. Processor A sends an ownership request to the home node.
2. The home responds by sending an exclusive copy of the block to processor A.
3. Before the exclusive copy arrives at processor A, another processor requests the same block, and the home sends a FWD message to processor A.
4. The FWD message arrives at processor A before the exclusive copy arrives.

Processor A records the fact that another processor has requested ownership of this block and sets the MAF.victimize bit. When the exclusive copy finally arrives, processor A cannot simply throw the block out, because that would cause a livelock. Instead, processor A fills the block to the Mbox and the core, ensuring forward progress, and loads the IPQ with a Victim command. Thus, we satisfy the forward progress requirement, as well as the requirement that we reliquish ownership of the block.

If an internal probe wins arbitration but is rejected because there is another transaction in the scache pipe to the same address, the internal probe is placed at the back of the FIFO queue.

If an internal probe wins arbitration but is rejected due to a bank conflict, the internal probe is placed in the retry queue.

11.3.5 Probe Queue — the PRQ

The Probe Queue (PRQ) is a 32-entry FIFO, similar in design to the RTQ and IPQ. The PRQ holds probes and non-block responses from the system before being processed in the Scache pipe. The PRQ accepts one probe or one non-block response from the SYS section every cycle. Additionally, a NACKed probe or non-block response from the MAF may also need to be written into the PRQ; therefore, the PRQ has two write ports.

11.3.5.0.1 Principle of Operation

The 21464 probe queue, unlike the 21363 design, holds two different classes of messages from the system: probes and non-block responses.

The probes are:

- FetchFwd
- ReadShrFwd
- ReadFwd
- ReadModFwd
- InvalToDirtyFwd
- SharedInval

The non-block responses are:

- ShrToDirtySuccessCnt
- ShrToDirtyFail
- ShrToDirtyProbCnt
- InvalAck
- WrIOACK
- NXMResponse
- ERRResponse

The other key difference between the 21464 and 21364 designs is that in the 21364, the PRQ is strictly ordered. Requests must be processed in FIFO order. In the 21464, no such restriction applies.

Each cycle, the head of the FIFO is read out and delivered to the MAF. Should this transaction be NACKed by the MAF (in C2), the transaction is placed at the tail of the FIFO and will be reissued again when it reaches the head. To prevent a transaction from being continuously NACKed, we record, at each PRQ entry, whether this probe has been rejected before. If a probe that has been rejected before is rejected again, a counter is incremented. When this counter saturates, a signal to the MAF arbitration logic asserts, giving the PRQ priority.

Because we place responses and forwards in the same queue, and because forwards generate responses, we have the possibility for deadlock. If the PRQ was full of forwards, and the response channel was full of responses, we wouldn't be able to take a

forward out of the PRQ because to process the forward requires a response buffer. But we can't sink any responses because the PRQ is full. To allieviate this, we reserve one PRQ entry for non-block responses. This ensures forward progress.

The PRQ receives responses and probes from the Rbox and Zbox. Should the probe queue become full, it must prevent the Rbox and Zbox from sending it more transactions. To accomplish this, the probe queue asserts backpressure signals to the Rbox and Zbox. It must assert these backpressure signals early enough so that all transactions already in flight to the PRQ can be sunk.

Use the following calculation to determine when to assert the back pressure signals:

- 4 (entries in pipe which may be NACKed)
- 2 (probes on the way to PRQ from SYS)
- 6 (probes in ring or that will be injected onto ring before backpressure signal arrives).

  Thirty-two minus twelve equals 20, therefore:

  Throttle probes when 19 entries in use.

  Throttle non-block responses when 20 entries in use.

11.3.5.1 Probe Address File (MAF) Contents per Entry

Table 11–5 PRQ Contents for Each Entry

Contents	# Bits	Ports	Description
valid	1		
probe_paddr<47:6>	42	1WR, 1RD	
probe_cmd<4:0>	5	2WR, 1RD	
transaction_id<16:0>	17	2WR, 1RD	Processor ID + Requester's MAF index
TOTAL	~65		

11.3.6 Victim Address File — the VAF

The Scache retains most blocks until the space (i.e. Scache set) they occupy is needed for another block. If a block is not held exclusively in the Scache at the time it is evicted, it is simply overwritten. But, if a block is in exclusive state, the directory must be notified that this cache is releasing exclusive access and, if the block is dirty, it must be written back to memory. Rather than delaying the fill that overwrites this block, the Scache moves the old contents to the victim data buffer (VDB), where the block waits for coherence before being sent to the home node. The victim address file (VAF) is a 64-entry buffer that stores addresses of Scache victims or probe responses to be sent to the memory system.

In the VAF 64-entry buffer, four entries (one for each TPU) are reserved for WrIo requests from the Mbox and four entries are reserved for probe responses.

Table 11–6 VAF Commands

Class	Encoding	Network Message	Destination	Data	Directory State
Non-block Responses	CS_VAF_CMD_SPCL_INV_ACK	InvalAck	Requester	No	RemoteExcl
	CS_VAF_CMD_INV_ACK	InvalAck	Requester	No	RemoteExcl
	CS_VAF_CMD_INV_TO_DIRTY_RESP	InvalToDirtyRespCnt(0)	Requester	No	RemoteExcl
Release Responses	CS_VAF_CMD_VICTIM_CLEAN	VictimClean	Home Diectory	No	InMemory
	CS_VAF_CMD_VICTIM_CLEAN_TO_SHR	VictimCleanToShared	Home Diectory	No	Shared
	CS_VAF_CMD_FORWARD_ACK_EXCL	ForwardAckExcl	Home Diectory	No	RemoteExcl
	CS_VAF_CMD_FORWARD_ACK_SHR	ForwardAckShared	Home Diectory	No	Shared
	CS_VAF_CMD_FORWARD_MISS	ForwardMiss	Home Diectory	No	RemoteExcl/Shared
	CS_VAF_CMD_SHR_TO_DIRTY_COMPL	SharedToDirtyComplete	Home Diectory	No	RemoteExcl
	CS_VAF_CMD_SHR_TO_DIRTY_RELEAS	SharedToDirtyRelease	Home Diectory	No	InMemory/Shared
Block Responses	CS_VAF_CMD_BLK_SHR	BlockShared	Requester	No	Shared
	CS_VAF_CMD_BLK_INV	BlockInvalid	Requester	Yes	Shared
	CS_VAF_CMD_BLK_DIRTY	BlockDirty	Requester	Yes	RemoteExcl
	CS_VAF_CMD_BLK_EXCL	BlockExclCnt(0)	Requester	Yes	RemoteExcl
Victim Block Responses	CS_VAF_CMD_VICTIM	Victim	Home Directory	Yes	InMemory
	CS_VAF_CMD_VICTIM_TO_SHR	VictimToShared	Home Directory	Yes	Shared
	CS_VAF_CMD_VICTIM_ACK_SHR	VictimAckShared	Home Directory	Yes	Shared
	CS_VAF_CMD_VICTIM_ACK_EXCL	VictimAckExcl	Home Directory	Yes	RemoteExcl

11.3.6.1 Victim Address File (VAF) Contents per Entry

Table 11–7 shows the VAF contents for each entry.

Table 11–7 VAF Contents For Each Entry

Contents	# Bits	Ports	Description
valid	1		The VAF entry has the valid response.
allocated	1		The VAF entry is speculatively allocated and not available for allocation.
victim_pa<46:6>	42	1WR, 1RD	
victim_cmd0<4:0>	5	1WR, 1RD	
victim_cmd1<4:0>	5	1WR, 1RD	
maf_idx<5:0>	6	1WR, 1RD, 1CAM	Requester's MAF index

Table 11–7 VAF Contents For Each Entry

Contents	# Bits	Ports	Description
req_node<9:0>	10	1WR, 1RD	Requester's node ID
has_full_blk	1	1WR, 1RD	Has the full victim block (i.e. ECC corrected and merge buffer data has been extracted).
TOTAL	~81		

11.3.6.2 Principle of Operation

The following table outlines the main victim flow for each Cbox pipe stage.

Table 11–8 Main Victim Flow for Each Cbox Pipeline Stage

Stage	Main Victim Flow
C0	—
C1	Speculatively allocate a VAF entry based on the transaction that won the arbitration. The following transactions allocate a VAF entry in C1: • LRUevict • BlkExclusiveProbable • Internal probes • Forwards (SharedInvals, etc) • ShrToDirtySuccess, ShrToDirtyProbable
C2	Deallocate VAF entry if transaction is rejected — if the incoming transaction got NACKed.
C3	—
C4	—
C5	—
C6	MAF logic computes initial VAF command based on tag from the Stag array and transaction type. This logic is covered in a later section on Cbox Flows.
C7	Send initial victim command from MAF to VAF.
C8	Write victim physical address to VAF, deallocate VAF entry if no victim.
C9	—
C10	CAM MAF with victim PA. If we hit a MAF entry with the victim PA, and the MAF state indicates that the victim PA is not coherent, then we must wait for coherence before sending the victim.
C11	Write victim data read from the scache into the victim data buffer (VDB).
C12	Receive probe_hit from Mbox and compute final victim command. This signal indicates if the Mbox MGB has modified data for the victim cache block. If this is the case, we must allow the Mbox MGB to "write-thru" to the VDB before sending the victim.
C13	Write VAF cmd, write VAF index to RSQ, if the victim is ready to be sent. The following two conditions can prevent the victim from being sent: • Victim is not coherent • Mbox MGB has modified data for this cache block, and has not yet written thru to the VDB.

Table 11–8 Main Victim Flow for Each Cbox Pipeline Stage

Stage	Main Victim Flow
C14	Read head of RSQ, read VAF index specified by RSQ.
C15	Send the VAF command to SYS, deallocate VAF entry if no VDB read required. However, if the corresponding VDB entry needs to be read by either the Zbox or Rbox, the VAF entry is not deallocated until this read occurs.
C16	Send second victim command, if applicable. A victim flow can require two messages to be delivered: one to the requesting node and one to the home node. If this is the case, then we send the second victim command at stage C16.

11.3.6.3 Secondary VAF Flows

As noted above, if the victim is not coherent, or if the Mbox MGB has not yet written the modified data thru to the VDB, we must not send the victim, as follows:

1. When an incoherent MAF entry receives the final InvalAck, making it coherent, the MAF sends the MAF index that became coherent to the VAF. We CAM the VAF with this MAF index, and the hit entry is written to the RSQ (note the retiming as we go from C5 to C15):

C5	MAF sends coherent MAF index to VAF.
C15/C6	CAM VAF with MAF index
C16	Send VAF index to RSQ
C17	Write RSQ

2. When the Mbox MGB writes thru to the VDB, the VAF entry (if coherent) is ready to be sent to the system (note retiming as we go from C12 to C16):

C9	Mbox MGB sends write-thru VDB
C10	—
C11	—
C15/C12	MAF sends wr_thru_vdb_done to VAF
C16	—
C17	Send VAF index to RSQ
C18	Write RSQ

11.3.6.4 Reserved VAF Entries

Four VAF entries, one per TPU, are available only for WRIO requests. The Mbox uses these four entries along with the four corresponding VDB entries, to store write IO data.

Additionally, four VAF entries are reserved for handling responses to forwards.

11.3.7 System Interface (SYS)

The System interface section (SYS) connects the Cbox to the Zbox and Rbox via the internal packet ring network. The SYS contains two 8-entry FIFOs: one for requests from the Cbox destined for either the Zbox or the Rbox, and one for responses from the Cbox destined for either the Zbox or the Rbox. Incoming packets from the Rbox are

either passed along to the Zbox, placed into the PRQ (probes and forwards), or driven into the scache pipe (block responses and I2DResponse). The SYS section also contains the Cbox CSRs.

In the 21464, unlike the 21364, we victimize Scache blocks at fill time. When the new fill arrives with data, we extract the victim, write the new data to the Scache, and write the victim to the VDB. Since we never stall fills, we must be assured that when the fill arrives, we have a VAF entry into which to put the victim. The MAF maintains a counter of the number of available VAF entries. Every time the SYS sends a request, we send a signal to the MAF counter to decrement the number of free VAF entries. Furthermore, the MAF sends back to the SYS a signal that indicates whether there is an available VAF entry. If there is not an available VAF entry, then the SYS must not send a request.

Note: The BlkExclusiveProb message might require two VAF slots. Therefore, when we send a ReadModSTC request from the SYS, we decrement the VAF free count by 2.

11.3.7.1 Principle of Operation:

Every cycle, the SYS section must arbitrate among the 3 possible sources that want to drive the ring. It does so with the following priority:

1. Rbox packet destined for the Zbox
2. System response packet from Cbox
3. System request packet from Cbox

The SYS section receives "back pressure" signals from both the Zbox and Rbox. These signals tell the Cbox whether the Zbox or Rbox can accept new packets for a particular class (responses or requests). If *either* the Rbox or the Zbox cannot accept responses, the CBox does not place anything on the ring. If *either* the Rbox or the Zbox cannot accept requests, the Cbox does not send a new request packet. Finally, to prevent Zbox starvation, if the Cbox has driven new packets out onto the ring in 15 consecutive cycles, it stalls for one cycle and does not place a new packet on the ring.

When the 8-entry FIFO in SYS for Cbox requests is filled, the SYS section signals back to the SRQ, which then stops sending further system requests to SYS.

Similarly, when the 8-entry FIFO in SYS for Cbox responses is filled, the SYS section signals back to the RSQ section, which then stops sending further responses to SYS.

11.3.7.1.1 Response FIFO Entry Fields

A response FIFO entry contains the following fields:

Table 11–9 System Interface Section Response FIFO Entry Fields

Field Name	Size
Block Address	31 bits
Home_Owner_Node	10 bits
Stripe_bit	1 bit
Cmd	8 bits

Table 11–9 System Interface Section Response FIFO Entry Fields

Field Name	Size
Requester_Node	10 bits
Requester_MAF_Idx	6 bits
Requester_VAF_Idx	6 bits
Request_destined_for_zbox	1 bit

11.3.7.1.2 Request FIFO Entry Fields

A request FIFO entry contains the following fields:

Table 11–10 System Interface Section Response FIFO Entry Fields

Field Name	Size
Block Address	31 bits
Req_Home_Node	10 bits
stripe_bit	1 bit
cmd	8 bits
req_maf_idx	(6 bits
request_destined_for_zbox	1 bit
iomask	8 bits
qwadd	3 bits
tpu_idx	2 bits

11.3.8 System Request Queue (SRQ)

The SRQ is a 60-entry [1] FIFO queue that buffers requests which miss in the Scache and require a system request. The SRQ serves two main functions:

- Since we can generate requests more rapidly than the paths to memory can accept them, the SRQ serves as a buffer between the MAF and the system interface:

  We can generate one request per cycle, but the ring interface is shared with the Rbox and Zbox, so sometimes our request will not win arbitration for the ring.

- The SRQ limits the number of outstanding requests to a given Scache index.

11.3.8.1 Principle of Operation

The Scache is six-way set associative, which means that at any given Scache index, we have six sets of storage, and six tags. If we were to allow more than six outstanding requests to the same Scache index, it is possible that the first six responses (fills) would arrive, write into the Scache, and then the seventh fill would arrive and victimize the first before we have actually written the fill data to the Scache.[2]

1 Four MAF entries are reserved for forward probes, so the SRQ needs to be only 60 entries, not 64.

When the MAF generates a system request — either due to a miss from the Mbox or a system response that does not fully satisfy the original request — we put the MAF index of the miss into the SRQ, along with the Scache index. Each cycle, we take the entry from the head of the SRQ and check how many outstanding requests we already have to the stored Scache index. If the number of outstanding requests is less than six (the number of Scache sets at any index), we do the following:

- We use the stored MAF index to read the PA from the MAF, and we deliver the request to the C-U-Z interface unit (CS_SYS).
- We increment the count of the number of outstanding requests to this Scache index.

However, if the number of outstanding requests is equal to six, we leave the SRQ entry on the queue and advance to the next SRQ entry; no request is sent.

When we receive a response from the system, the proper counter is decremented.

As noted earlier, the Scache index consists of bits PA<18:6>. Using counters to keep track of the number of outstanding requests at each index would therefore require 8K counters. To reduce this storage requirement, we instead allow a *total* of six requests to a *group* of 128 indixes. Physical address bits <11:6> are used to specify one of 64 counters. As an example, we would allow a total of six requests to be outstanding to the following group of Scache indices at any one time:

0, 128, 256, 384, ...

This could be a performance issue for strided code.

There is a debug mode in the Cbox that allows only one outstanding request per PA<11:6>.

11.3.9 Retry Queue (RTQ)

Transactions in the Scache pipeline can encounter conditions that prevent them from completing normally. In those cases, the transaction must be *retried*, that is, reexecuted through the Scache pipeline. The retry queue is a 64-entry FIFO that holds transactions which must be retried.

11.3.9.1 Principle of Operation

The following conditions can cause an Scache transaction to be placed in the retry queue and retried:

- Scache tag array bank conflict
- Scache data array bank conflict
- Single-bit ECC Scache tag error
- Single-bit ECC Scache data error
- Ifetch request from Mbox hits data in Mbox MGB
- System response arrives and finds MAF entry with MAF.i_fill_ena asserted

---

2 The current Scache pipeline is such that with six sets, this situation can not occur. Since the pipeline and the number of sets may change between now and tapeout, the SRQ remains in the design.

An entry in the RTQ consists of the following:

- MAF index associated with the transaction (6 bits)
- VAF index associated with the transaction (6 bits)
- Retry type (2 bits)
- Retry command (4 bits)
- Retry due to data error (1 bit)
- This retry has been nacked (1 bit)
- This transaction originated with an IO processor (1 bit)

If a transaction is to be retried, the MAF.sc_inflight bit for the MAF entry associated with the retry is kept asserted until the retry successfully retries and clears the Scache pipe. This ensures that no other transaction to the same address (with fills being the only exception) may enter the Scache pipeline until the retry has been processed.

The RTQ must have the same number of entries as the MAF (64) because we could have a situation where every system fill returns an exclusive block to the Cbox, and each MAF entry associated with these fills has MAF.i_fill_ena asserted (Istream request). This requires that each Ifetch be retried from the RTQ.

Each cycle, the RTQ reads the entry at the head of the FIFO, delivers it to the MAF, and deallocates the RTQ entry. If the MAF signals an ACK 3 cycles later, all is well. If instead, the MAF signals a NACK back to the RTQ (transaction was rejected), the RTQ allocates a new entry and pushes it into the back of the FIFO.

Situations can arise whereby a transaction in the retry queue is continuously denied access to the Scache pipeline. The RTQ has logic to detect this situation. When the RTQ detects that an entry is being denied access, a signal is asserted to the MAF arbitration logic. This signal forces the MAF to reject all requests that have a lower priority than the RTQ. Please refer to the section on Cbox livelock/starvation avoidance for more details.

11.3.10 TTQ

11.4 Fill Datapath — the CF Partition

11.4.1 FBE

11.4.2 VDB

11.4.3 FDB

11.4.4 DBM

11.4.5 RBI

11.4.6 RBO

11.5 Scache Tag Array — the ST Partition

The Scache tag array (STAG) stores cache states of blocks in the secondary cache (Scache).

11.5.0.1 Principle of Operation

In response to a Scache tag request command from the Scache control, the Scache tag array is responsible for the following:

- Look up the Scache tag and send the CURRENT cache block state to the Scache control.
- Update the cache state and LRU if necessary.
- In case of a single bit tag ECC error, correct and store the corrected tag in the tag ECC register and signal it to the Cbox. The Cbox (i.e. Scache control) retries the request.
- The tag ECC register gets cleared
  - After the retry reads the corrected tag from the register.
  - A probe or a system fill to the same Scache index since they may displace or victmize the cache block which is in the tag ECC register.

11.5.0.2 Pipeline Stages

Table 11–11 shows the pipeline stages for the Scache tag array.

Table 11–11 Scache Tag Array Pipeline Stages

S2	S3	S4	S5	S6
1. Decode the Scache index for RD. 2. Generate the tag ECC [1].	1. Read the Scache tag/LRU. 2. CAM the ECC tag register with the MAF index.	1. Tag compare and Set select. 2. Syndrom generation and Error detection. 3. Decode the Scache index for WR. 4. Decode the LRU and look up the stale fill table if LRU_RD_ENA is asserted. 5. Fix the tag ECC bits if necessary [1].	1. Send the set number to the Cbox and SC data array. 2. Write the Scache tag and/or LRU if necessary. 3. Single bit error correction and load the corrected tag into the ECC register if a retry is required [2]. 4. Clear the tag ECC register if necessary. 5. Write the stale fill table if C_ST_CMD_LRUEVICT.	1. Send the response (VSD) to the Cbox.

Notes:

- [1]: The ECC bits are function of the physical address and the cache state. Yet we only need tag ECC bits which depend only on the physical address for the tag compare. Tag ECC bits which depend on the cache state are generated before the tag is written back.
- [2]: A retry is required if
  - Scache miss and a single bit tag ECC error in any set OR
  - Scache hit and a single bit ECC error in the same set.

11.5.0.3 State Transition

Table 11–12 shows the Scache tag state transition table.

Table 11–12 Scache Tag State Transition Table

Current Tag State Command (cs%st_cmd_c3a)	Invalid	Shared	Excl	Dirty	LRU RD	LRU WR	Tag RD	Tag WR	Write the Stale Fill Table	Inval. ECC[1]
C_ST_CMD_NOOP	Invalid	Shared	ExclCln	Dirty	No	No	No	No	No	No
C_ST_CMD_MISS	Invalid	Shared	ExclCln	Dirty	No	Yes[2]	Yes	No	No	No
C_ST_CMD_SETDIRTY	Invalid	Shared	Dirty	Dirty	No	Yes[2]	Yes	Yes	No	No

Table 11–12 Scache Tag State Transition Table (Continued)

Current Tag State Command (cs%st_cmd_c3a)	Invalid	Shared	Excl	Dirty	LRU RD	LRU WR	Tag RD	Tag WR	Write the Stale Fill Table	Inval. ECC[1]
C_ST_CMD_LRUEVICT[3]	Invalid				Yes	No	Yes	No	Yes	Yes
C_ST_CMD_INVAL	Invalid	Invalid	Invalid	Invalid	No	Yes[4]	Yes	Yes	No	
C_ST_CMD_CTOS	Invalid	Shared	Shared	Shared	No	Yes[2]	Yes	Yes		
C_ST_CMD_STOE	JInvalid	ExclCln	Must not happen.		No	Yes[2]	Yes	Yes		
C_ST_CMD_STOD	Invalid	Dirty			No	Yes[2]	Yes	Yes		
C_ST_CMD_BLKINV	Invalid	Must not happen			No	Yes[4]	No	Yes		[Yes]
C_ST_CMD_BLKSHR	Shared				No	Yes[2]	No	Yes		
C_ST_CMD_BLKEXCL	ExclCln				No	Yes[2]	No	Yes		
C_ST_CMD_BLKDIRTY	Dirty				No	Yes[2]	No	Yes		
C_ST_CMD_RCVRY	Invalid	Shared	ExclCln	Dirty	No	No	No	No	Yes	
C_ST_CMD_STOI										
C_ST_CMD_ETOS_DTOI										

1 Invalidate ECC register in the index match

2 Make the set most recently used.

3 To prevent the LRUEvict from evicting a stale fill block, we check the stale fill block table at the same time as we read out the LRU. The stale fill block table stores the set numbers that may have stale fill blocks. Those sets in the stale fill table must not get evicted. The stale fill block table is a simple FIFO that stores the last N bits (4 ~ 8 bits) of Scache index and the set number of system fills in-flight in the Scache.

4 Make the set least recently used.

- Stag Read/Write conflict
  - Because the Scache bank conflict can't be detected early enough it is possible to have a read/write conflict to the same Scache bank.
  - For a Scache bank conflict, the earlier transaction takes precedence over the later one, which means the WRITE must proceed. One exception is when we try to evict a block to make a room for the following fill (i.e. LRU_RD_ENA is asserted), the WRITE is discarded in favor of the READ. The Scache control (Cbox) is responsible for replaying the rejected transaction.

ST_RD_ENA_C2A	LRU_RD_ENA_C2A	ST_WR_ENA_C4A	LRU_WR_ENA_C4A	ACTION
0	0	0	0	NoOp/Fill
X	X	1	0	Must not happen
0	0	0	1	Write LRU
0	0	1	1	Write tag and LRU
0	1	X	X	Must not happen
1	0	0	0	Read tag

ST_RD_ENA_C2A	LRU_RD_ENA_C2A	ST_WR_ENA_C4A	LRU_WR_ENA_C4A	ACTION
1	0	0	1	Read tag and write LRU
1	0	1	1	Bank conflict: Write tag and write LRU
1	1	X	X	Bank conflict: Read tag and read LRU

11.5.0.4 Stale Fill Table

The SFT (Stale Fill Table) stores the set number and the Scache index of the system fill in progess in the Scache to prevent from victimizing the set which has the stale data in the Scache.

Table 11–13 Stale Fill Table (SFT)

	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	Notes
C_ST_CMD_LRUEVICT(0)	[1]	[2]												
C_ST_CMD_BLK*(0)			[3]										[4]	
C_ST_CMD_LRUEVICT(5)											[1]	[2]		Victimizing the block before the block is written into the Scache.

Notes:

- [1] LRU decode and set select
- [2] Scache data array read
- [3] Scache tag array write
- [4] Scache data array write
- If we have less than 6 sets, we must not send more than (N-1) system requests to the same Scache index, where N is the number of sets.

11.5.0.5 The 21464 Scache Least Recently Used (LRU) Scheme

The 21464 Scache is 6-way set associative cache. To minimize the probability of the Scache bank conflict, the 21464 LRU scheme is designed such that only a LRU victim needs to read the LRU. All other Scache transaction does not require reading of the LRU array. The proposed 21464 Scache LRU uses an approximate tree-like structure.

Table 11–14 Scache Least Recently Used (LRU) State Bits

States	Notes
Arb	The set A is more recent than the set B
CrD	The set C is more recent than the set D
ErF	The set E is more recent than the set F
ABrCD	The set AB is more recent than the set CD
ABrEF	The set AB is more recent than the set EF
CDrEF	The set CD is more recent than the set EF

- Making a set the most recently used (MRU) set:

	LRU states					
	ArB	**CrD**	**ErF**	**ABrCD**	**ABrEF**	**CDrEF**
Make the set A MRU	Set	No change	No change	Set	Set	No change
Make the set B MRU	Clear	No change	No change	Set	Set	No change
Make the set C MRU	No change	Set	No change	Clear	No change	Set
Make the set D MRU	No change	Clear	No change	Clear	No change	Set
Make the set E MRU	No change	No change	Set	No change	Clear	Clear
Make the set F MRU	No change	No change	Clear	No change	Clear	Clear

- Making a set the least recently used (LRU) set

	LRU States					
	ArB	**CrD**	**ErF**	**ABrCD**	**ABrEF**	**CDrEF**
Make the set A LRU	Clear	No change	No change	Clear	Clear	No change
Make the set B LRU	Set	No change	No change	Clear	Clear	No change
Make the set C LRU	No change	Clear	No change	Set	No change	Clear
Make the set D LRU	No change	Set	No change	Set	No change	Clear
Make the set E LRU	No change	No change	clear	No change	Set	Set
Make the set F LRU	No change	No change	set	No change	Set	Set

- Determining an LRU set from the LRU states

LRU States						
ArB	CrD	ErF	ABrCD	ABrEF	CDrEF	LRU Set
0	X	X	0	0	X	Set A is the LRU
1	X	X	0	0	X	Set B is the LRU
X	0	X	1	X	0	Set C is the LRU
X	1	X	1	X	0	Set D is the LRU
X	X	0	X	1	1	Set E is the LRU
X	X	1	X	1	1	Set F is the LRU
X	X	X	0	1	0	Illegal state
X	X	X	1	0	1	Illegal state

Notes:

- We assign an arbitrary set when an illegal state is detected.

11.5.0.6 Scache Tag ECC Code

The 21464 Scache tag (paddr<46:19> + vsd<2:0>) array is protected by a 32b ECC code.

The C''s in the table below indicate which bits contribute to the corresponding check bit. The corresponding check bit is the parity of the contributing bits and the syndrom bit is the parity of the corresponding check bit and the contributing bits.

Check bits 6 and 4-0 are calculated with odd parity, meaning that the total number of ones in the contributing data bits and the check bit will be odd when stored in memory. Check bit 5 is calculated with even parity, to ensure that the cases of all-zero and all-ones returned from the memory are reported as uncorrectable errors.

When the tag is invalidated, a complete entry will be written, with zeros in all the data bits; this will require that the check bits be written with 0x5F so that it can be read without error. All valid syndromes (in the sense that they represent correctable errors) have either 1 or 3 ones. A syndrome with only 1 one represents an error in a check bit; a syndrome with 3 ones represents an error in a data bit. Any syndrome with an even number of ones, or with 4 or more ones, represents an uncorrectable error. Customary practice is to ignore the cases of 5 or 7 ones, and report an uncorrectable error when there is an even number of ones in the syndrome.

Table 11–15 Scache TAG Syndrome Bits

	VSD			Physical Address																										
C/S[1]	Shared	Dirty	Valid	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19
6	C	C	C	C		C			C	C		C			C				C	C		C			C				C		

Table 11–15 Scache TAG Syndrome Bits

5																		C	C	C	C	C	C	C	C	C	C	C	C	C
4									C	C	C	C	C	C	C	C	C	C											C	C	C
3			C	C	C	C	C	C							C	C	C	C							C	C	C	C			
2		C				C	C	C				C	C	C				C				C	C	C				C			
1	C			C	C			C		C	C			C			C			C	C			C			C			C
0	C	C	C		C		C		C		C		C			C			C		C		C			C			C	
S[2]	43	45	49	4A	0B	4C	0D	0E	51	52	13	54	15	16	58	19	1A	1C	61	62	23	64	25	26	68	29	2A	2C	70	31	32

[1] C/S means Check/Syndrome

[2] S means Syndrome

- Masks to select data participating in each check bit:
  - check 6 mask= 0x7A691A44
  - check 5 mask= 0x00001FFF
  - check 4 mask= 0x007FE007
  - check 3 mask= 0x1F81E078
  - check 2 mask= 0x238E2388
  - check 1 mask= 0x4CB24C91
  - check 0 mask= 0x75549522
- Tag compare
  - valid
  - paddr<46:19>
  - CB<5:3>
- Error detection

		OR (SYN<5:0>)	
		Zero	**Non-zero**
EVEN PARITY(SYN<6:0>)	**Even**	Good data	Double or even number of errors
	Odd	Triple or odd number of errors	Single bit error

If the memory returns all zeros in the data and check bits, the resulting syndrome will be 0x5F. All zeros will result in a syndrome of 0x12. Both syndromes have an even number of ones, so will be reported as uncorrectable errors.

- Error correction
  - The SYN<5:0> indicates the location of the bit error. Then the error bit simply gets flipped.
  - Corrected bit = (Uncorrected bit) XOR (DECODE(SYN<5:0>))

11.6 Scache Data Array — the SG Partition

This is mentioned in Tables 10-2 and 10-3.

11.7 Flows

11.7.1 Overall Pipeline Flow

Fills, Probes, and Misses all access the Scache using the same pipeline. Each cycle one operation is picked for launch:

- FillExcl, FillShared from system; displace a victim (2 cycle operation, granted on even cycles. Highest priority)
- Transfers between Router and Rambus
- Replays (Tag ECC error, Data ECC error, resource or ordering conflict)
- ChgToExclMark, ReadMark, Probes (Inval, Forward, ForwardInval) and VictimAcks
- Local memory accesses
- Misses and Evict requests from the MAF
- Early returns from system

(To avoid deadlocks the system priorities are: Fills -> Forwards -> Probes -> Requests)

11.7.1.1 Pipe Operation

Operations are launched into the pipe and either complete successfully or fail because of an error in the Tag or Data ECC code, or fail because of a resource conflict with a previously launched operation. Failed operations queue to be replayed. Corrected tag or data for ECC fails is temporarily saved in a bypass register. When the operation is retried it accesses the bypass register. The bypass register is cleared by Tag or Data writes with the same index.

Table 11–16 Scache Control Pipeline Diagram

	Fill_1	Fill_2	CTE_1 (change)	CTE_2 (change)	CTE_1 (fill from victim)
Arbitrate	Grant if cycle is even.				
Read MAF	Read PA from MAF. Check MAF: CTE [3]. Update MAF state.	Read PA from MAF.	Read PA from MAF. Check MAF: CTE [3]. Update MAF state.	Read PA from MAF.	Read PA from MAF. Check MAF: CTE [3]. Update MAF state.
Tag Launch	Detect conflicts [1]. Row decode Tag Ram.		Detect conflicts [1]. Row decode Tag Ram. Check VAF. If find shared block follow 'fill from victim' below.	Row decode tag ram.	Detect conflicts [1]. Row decode Tag Ram. Check VAF. If don't find shared block follow 'change' above.
Read Tags	Read tag ram.	(Don't read tags).	Read tag ram.	Read tag ram.	Read tag ram.
Com-pare	Select victim set (LRU). Generate victim tag syndrome.	Bypass victim set number from Fill_1		Compare PA to tags. Determine set number. Generate tag syndromes.	Select victim set (LRU). Generate victim tag syndrome.
Read Data	(Don't write tags) . Correct victim tag. Read data ram.	Write tag, VSD. Invalidate Tag_ECC register if index matches fill index.		Set Excl, LRU (if match). Correct tags. Tag ECC replay [4]. Invalidate Tag_ECC register if index matches CTE index. Read data ram.	(Don't write tags) . Correct victim tag. Read data ram.
Set Select	Send victim PA to MBox. Write vitctim PA to VAF. Set select victim data.	Send fill PA to MBox.		Send CTE PA to MBox. Set select data.	Send victim PA to MBox. Write victim PA to VAF Set select victim data.
Read VDB				Generate data syndrome.	
Fill Bus		Send fill data to MBox.		Send fill data to MBox. Correct data. Data ECC replay [3].	Write victim data to VDB.
Write VDB	Write vitctim data to VDB.				
Write 1					
Write 2		Write data ram.			

Table 11–16 Scache Control Pipeline Diagram (Continued)

	Arbitrate	Read MAF	Tag Launch	Read Tags	Com-pare	Read Data	Set Select	Read VDB	Fill Bus	Write VDB	Write 1	Write 2
CTE_2 (fill from victim)		Read PA from MAF.	Row decode tag ram.	Read tag ram. (Not used)	Bypass victim set number from Fill_1	Write tag, VSD. Invalidate Tag_ECC register if index matches fill index.	Send fill PA to MBox.	Read VDB	Send fill data from VDB to MBox.			Write data ram.
Probe (Inval)		CAM MAF; merge or allocate entry. Detect hold conditions [2]. Hold in MAF.	Detect conflicts [1]. Row decode Tag Ram. Check and conditionally Invalidate VAF [5].	Read tag ram.	Compare PA to tags. Determine set number. Generate tag syndromes.	Write tag Invalid (if match). Correct tags. Tag ECC replay [4]. Invalidate Tag_ECC register if index matches fill index.	Send probe to MBox. Send InvalAck.					
Probe (Forward)		CAM MAF; merge or allocate entry. Detect hold conditions [2]. Hold in MAF.	Detect conflicts [1]. Row decode Tag Ram. Check VAF. I f victim is there mark and don't try reloading. Still need to update tags.	Read tag ram.	Compare PA to tags. Determine set number. Generate tag syndromes.	Update tag. (Excl -> DirtyShared, DirtyShared -> Shared) Correct tags. Tag ECC replay [4]. Invalidate Tag_ECC register if index matches fill index. Read data ram.	If was Excl send probe to MBox. MBox write through to SData and VDB. Set select forward data.	Generate data syndrome.	Send fill data to MBox. Correct data. Data ECC replay [3].	If not already there, write forward data to VDB.		

Table 11–16 Scache Control Pipeline Diagram (Continued)

	Arbitrate	Read MAF	Tag Launch	Read Tags	Com-pare	Read Data	Set Select	Read VDB	Fill Bus	Write VDB	Write 1	Write 2
Probe (ForwardInval)		CAM MAF; merge or allocate entry. Detect hold conditions [2]. Hold in MAF.	Detect conflicts [1]. Row decode Tag Ram. Check VAF. If victim is there mark and don't try reloading. Still need to update tags.	Read tag ram.	Compare PA to tags. Determine set number. Generate tag syndromes.	Write tag Invalid. Correct tags. Tag ECC replay [4]. Invalidate Tag_ECC register if index matches fill index. Read data ram.	If was Excl send probe to MBox. Set select forward data.	Generate data syndrome.	Correct data.	If not already there, write forward data to VDB.		
Local Probes												
Miss		Read PA from MAF. Update MAF in-flight bit.	Row decode tag ram.	Read tag ram.	Compare PA to tags. Determine set number. Generate tag syndromes.	Write tag LRU (if match). Correct tags. Tag ECC replay [4]. Read data ram.	Send fill PA to	MBBox. Set select data.	Generate data syndrome.	Send fill data to MBox. Correct data. Data ECC replay [3].		

Notes:

[1] Resource and Order Conflicts

Table 11–17 Resource and Order Conflicts

	Resource conflicts	Order conflicts
Fill	Same tag bank as probe or miss 2 cycles earlier. Replay Probe or Miss. Same tag bank as CTE 3 cycles earlier. Replay CTE.	In-flight CTE or fill to the same index about to write the data that the fill is victimising.
CTE	Same tag bank as CTE 2 or 3 cycles earlier. Same tag bank as Probe or Miss 1 or 2 cycles earlier.	In-flight Fill, CTE, Forward or ForwardInval to the same index which may extract victim. Victim buffer entry awaiting modified bytes from MBox.

Table 11–17 Resource and Order Conflicts

	Resource conflicts	Order conflicts
Probe	Same tag bank as Fill or CTE 3 cycles earlier. Same data bank as Fill or CTE 7 cycles earlier. (not Inval) Same tag bank as Probe or Miss 2 cycles earlier.	In-flight Fill, CTE, Forward or ForwardInval to the same index which may extract victim. In-flight Fill or CTE to the same index about to write the data that the Probe is reading.
Miss	As Probe	In-flight Fill, CTE, Forward or ForwardInval to the same index which may extract victim. In-flight Fill or CTE to the same block about to write the data that the Miss is reading.
Write Through	Same tag bank as .. Same data bank as ..	

[2] MAF holds CTE or Probe when:

- Match MAF entry for the same block which has received a ReadMark or CTEMark but is not yet coherent.
- Probe to the same block in-flight which might replay. (For CTE, In-flight Forward for case FillExcl; Forward; CTEMark)

[3] CTE cases:

- FillExcl checks MAF. If already have Shared block, Fill is dropped and ReadMark is converted into a CTE. (or vice versa?)
- CTEMark checks MAF. If already have Excl block, don't need to do anything.

[4] ECC replays:

- CTE (change), Inval, Forward, ForwardInval or Miss may replay because of Tag ECC error.
- CTE (change) or Miss may replay because of a Data ECC error.

[5] Invalidating VAF entries:

- Probe Inval invalidates and deallocates shared blocks in the VAF.
- Probe Inval marks excl blocks which have not sent victim request as 'no victim req needed'.

11.7.1.2 Pipeline Timing Diagrams

11.7.1.2.1 Scache Control Pipeline Stages

Table 11–18 Scache Control Pipeline Stages

Mbox Retry Pipe	Scache Pipeline Stages		Blk* I2DResp [4]		Retry	S2D*
Send Req	Arb	C0 Arb	Arbitrate for the Scache pipe	Arbitrate for the Scache pipe	Arbitrate for the Scache pipe	Arbitrate for the Scache pipe
	MAF	C1 MAF	Read PA & MAF states. Allocate a VAF entry		Read PA. Bank conflict check.	Read the PA & MAF states. Bank conflict check.
Ack	Scache Tag Pipe	C2 Tag launch	Compute the new MAF states			Compute the new MAF state.
Bub ble		C3 Rd Tag	Write the new MAF state Read LRU/tags		Read tag.	Write the new MAF state. Read tag.
		C4 Tag Compare	Victim set select		Error detection. Set select.	Error detection. Set select.
Q		C5 Rd Data	Rd Scache data array. Victim tag ECC.		Tag ECC correct Update the SC tag [3]. Read SC data.	Tag ECC correct Update the Stag. SC data rd.
R0		C6 Set Select		Write LRU/tag.		
R1		C7 Drive Data	Send Victim addr and VAF idx to Mbox. Write the VAF.		Send addr/tag to Mbox	Send address/tag to Mbox
R2	Scache Data Pipe	C8 Fill Bus		Send the Fill addr/tag to Mbox		
E		C9 WR 1				
M0		C10 WR 2	Victim data on the fill bus. Victim data correction.		Fill data on the fill bus.	fill data on the fill bus.
M1		C11 WR 3	Write the VDB (C11B).	Fill data on the fill bus.		
M2		C12 WR 4				
		C13 WR 5				
		C1 4WR 6				
		C15				
		C16		Write SC data array.		

Table 11–18 Scache Control Pipeline Stages (Continued)

	Mbox Retry Pipe													
	Send Req		Ack	Bub-ble		Q	R0	R1	R2	E	M0	M1	M2	
	Scache Pipeline Stages													
	Arb	MAF	Scache Tag Pipe						Scache Data Pipe					
	C0 Arb	C1 MAF	C2 Tag launch	C3 Rd Tag	C4 Tag Compare	C5 Rd Data	C6 Set Select	C7 Drive Data	C8 Fill Bus	C9 WR 1	C10 WR 2	C11 WR 3	C12 WR 4	C13 WR 5	C1 4WR 6	C15	C16
Forwards	Arbitrate for the Scache pipe	Alloc/merge MAF entry. Rd MAF states Bank conflict check.	Compute new MAF states.	WR new MAF states	Set select. Error detection.	Tag ECC correct Update the Stag. SC data rd.		Send address/tag to Mbox. CAM the VAF.			fill data on the fill bus [2]. ECC correction	WR the VDB					
SharedInval	Arbitrate for the Scache pipe																
WRIOAck	Arbitrate for the Scache pipe	Read MAF state.	Compute the new MAF state.	Write the new MAF state.				Send WRIOAck to Mbox.									
Misses	Arbitrate for the Scache pipe	Alloc/merge MAF entry. Read out MAF state. bank conflict check.	Compute the new MAF state. Send ACK/MAF index to Mbox.	Write the new MAF state. Read Scache Tag.	Error detection. Set select.	Tag Error correct. Update the LRU. Rd SC data.		Send tag to Mbox. CAM the VAF.			Fill data on the fill bus.						

Notes:

[1]: Cbox sends the probe address to Mbox if the probe is:

*Forward.

SharedInval*.

[2]: Cbox sends the cache block to Mbox if Scache is to keep a copy of the cache block Shared after the probe.

[3]: If the retry is due to a data ECC error, the Scache tag has already been updated and the retry must not change the tag state again.

[4]: Blk* and InvToDirtyResp take 2 Scache cycles. Cbox sends early warning to Mbox if the fill address bus is not used by other transaction. CBox must ensure to send the early warning such that the Mbox retry will find the fill data (i.e. do not send the early warning too early).

[5]: Writing the VDB with the ECC corrected victim data is not time critical as long as we write the VDB before the merge buffer write to the VDB.

The Scache control reads the physical address from the MAF in C1A and drive them to the Scache tag in C1B.

11.7.1.3 Resource Conflict

The following resources are shared:

- Scache tag pipe.
- MAF PA read port.
- MAF PA CAM port.
- Fill/Victim/Probe address bus (47+ bits).
- Fill data bus (512+ bits).
- Scache tag bank.
- Scache data bank.

Table 11–19 Required Resource

			MAF PA			Scache				
	MAF entry	**VAF entry**	WR	RD	CAM	Tag Bank	Fill Addr Bus	Write-thru Bus	Data Bank	Fill Data Bus
Blk*		1		1		2	2		2	2
InvalToDirtyRespCnt		1		1		2	2			1
ShrToDirty*Cnt		1[1]		1		2	1		1	1
*Req/*Fwd	1	1	1		1	2	1		1	1
SharedInval	1	1	1		1	2	1			
Miss	1		1		1	2	0 ~ 1		1	1
Partial Fill				1			1			1
Write-thru						1		1	1	

[1] **THIS FOOTNOTE NOT SPECIFIED**

Notes:

- We need a VAF entry,
  - For a VictimClean if a ShrToDirtySuccessCnt does not find a Shared copy in the Scache.
  - For a ShrToDirtyComplete or ShrToDirtyRelease for a ShrToDirtyProbCnt.

- Blk* and InvToDirtyRespCnt takes 2 Scache pipe cycle since may victimize a cache block.
  - The victim block: send the victim address to Mbox & write the VDB via the fill bus.
  - The fill block: send the fill address and the fill data to Mbox/Ibox.
- Blk* writes the Scache data ram while Misses and Probes read the Scache data ram.

11.7.1.4 Scache Bank Conflict Check

Table 11–20 shows the Scache bank conflict timing

Table 11–20 Scache Bank Conflict Timing

	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	Notes
Miss0	Update LRU. Read data.	Send fill addr.		Send fill data.											
Miss1	Compare tag.	Update LRU. Read Sdata.	Send fill addr.		Send fill data										
Miss2	Read tag.	Compare tag	Update LRU. Read Sdata.	Send fill addr.		Send fill data.									[1]
Victim3		Read tag/LRU.	Compare tag.	Victim tag ECC.	Send victim addr.										Cache LRU bank conflict with Miss1.
Fill4					Update tag/LRU. Read data.	Send fill addr.		Send fill data.						Write Sdata.	
Victim6					Read tag/LRU.	Compare tag.	Victim tag ECC.	Send victim addr.							Tag bank conflict with Fill4.

Table 11–20 Scache Bank Conflict Timing

	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	Notes
Miss5				Read tag.	Compare tag.	Update LRU. Read data.	Send fill addr.		Send fill data						
Miss6					Read tag.	Compare tag.	Update LRU. Read Sdata.	Send fill addr.		Send fill data					Scache tag bank conflict with Fill4.
Miss13												Read LRU.	Compare tag.	Update LRU. Read Sdata.	Data array bank conflict with Fill4

Notes:

- [1]: We have a separate LRU array; hence, this does not cause a Stag array bank conflict (i.e. no miss-miss bank conflict).
- The current Scache proposal has 16 independent banks.
- The Scache bank conflict check prevents:
  - Two accesses to the same Scache bank (tag & data array).
  - The write through to the bank (index ??) that has a stale victim block (i.e. victim block in the process of being written to the VDB).
  - The write-through to the bank (index) that has an inflight fill block.
  - In case of a bank conflict, the request that entered the Scache pipe later gets retried. However, the system fill preempts the preceding Scache access in case of bank conflict to avoid retrying the system fill.
- Do a system fill at even cycle to prevent the resource conflict with the immediately following system fill.
- We must ensure that there is no bank conflict between reading of the Scache data for LRUVictim and writing of the Scache for a fill block. This means the
- Scache write must be in the even cycle.
- Misses(X) can have bank conflict to the Scache data bank with a preceding system fill.
- In case of bank conflicts, the request that entered the Scache pipe later gets retried except for system fills which forces the conflicting request to get retried even if it entered the Scache pipe earlier.

11.7.2 Fill and LRU Evict Flow

11.7.2.1 Hiccup Flow

11.7.3 Probe Flow

11.7.4 Mbox Request Flow

The Cbox looks up the Scache tag and sends the tag and/or fill data to the Mbox in response to the Mbox Miss request. In case of Scache miss, the Cbox sends the system request. Miss request Scache look=up.

Table 11–21 Miss Request Command Summary

Miss Request Command	Scache State	Dcache Fill Command	VSD to Mbox	Fill Data to Mbox	MAF.sys_cmd<2:0>
Ifetch	0XX	C_DFILL_CMD_MGBPROBE	0XX	No	ReadShared
	100/110	C_DFILL_CMD_MGBPROBE[1]	100/110	No	None
	101	C_DFILL_CMD_MGBPROBE[1]	101	Yes	None
FetchLine	0XX	C_DFILL_CMD_FILLBLK[2]	0XX	No	Read
	100/101/110	C_DFILL_CMD_FILLBLK	100/101[3]/110	Yes	None
FetchLineMod CtoD	0XX	C_DFILL_CMD_FILLBLK[2]	0XX	No	ReadMod
	110	C_DFILL_CMD_FILLBLK	110	Yes	ShrToDirty
	100/101	C_DFILL_CMD_FILLBLK	101[3]	Yes	None
PfetchLineMod	0XX	C_DFILL_CMD_FILLBLK[2]	0XX	No	ReadMod
	100/101/110	C_DFILL_CMD_FILLBLK	100/101[3]/110	Yes	None
PfetchNocache	0XX	C_DFILL_CMD_FILLBLK[2]	0XX	No	ReadShr
	100/101/110	C_DFILL_CMD_NOCACHE	100/101[3]/110	Yes	None
PfetchScache	0XX	C_DFILL_CMD_FILLBLK[2]	0XX	No	ReadShr
	100/101/110	C_DFILL_CMD_NOOP	0XX	No	None
ItoD	0XX	C_DFILL_CMD_FILLBLK[2]	0XX	No	InvToDirty
	110	C_DFILL_CMD_FILLBLK	110	Yes	ShrToDirty[4]
	10X	C_DFILL_CMD_ITODRESP	101[3]	Yes	None
CtoDSTC	0XX	C_DFILL_CMD_STCDONE[2]	0XX	No	None
	110	C_DFILL_CMD_FILLBLK	110	Yes	ShrToDirtySTC
	10X	C_DFILL_CMD_STCDONE	101[3]	Yes	None

[1] Mbox send dl%probe_hit when the Ifetch hits the dirty block in the MGB.

[2] Mbox invalidates the Dcache block.

[3] The cache state to the Mbox will be ExclCln if the block is not coherent to prevent the merge buffer from wring the Scache.

4 In order not to replay InvToDirtyRespCnt due to Scache tag ECC error, we send a ShrToDirtyReq if we have a shared copy even if Mbox has the full cache block modified.

Notes:

- The MAF.sys_cmd gets set after Misses look up the Scache tag.
- The MAF.sys_cmd can be changed before we make the system request (i.e. MAF.need_sys_rqst):
  - A new Miss request gets merged.
  - A probe hits a ShrToDirtyReq.
  - If a probe hits a ShrToDirtySTCReq, we may de-allocatethe MAF entry.

System request pipe arbitration

- The system request pipe is arbitrated between Miss requests from the MAF and Victim from the VAF.
- System requests queued in the MAF are arbitrated based on the age priority.
- The responses in the VAF has the priority over the reqeusts in the MAF.

11.7.5 Victim Flow

The sources of Scache Victim(X) are:

- External probe (*Forwards).
- Internal probe.
- LRU displacement by a system fill.

The victim block has to be coherent before the Victim can be sent to the home node. This requires a victim to CAM the MAF.

The LRU evicted Victim(X) at the system fill time must:

1. Pull the victim out of the Scache.
2. Perform the victim Tag ECC correction.
3. If the victim block is shared, then the victim process completes.
4. If the victim block is Exclusive or Dirty:
   - Write the ECC corrected victim address to the VAF.
   - Send the ECC corrected victim address to the Mbox.
   - Put the ECC corrected victim data into the VDB.
   - CAM the MAF to see if the victim block is coherent.
   - If the merge buffer has modified bytes, the merge buffer writes the modified bytes to the VDB after the ECC corrected Scache victim is in the VDB.

- There exists a time window where the merge buffer does not know the block has been victimized until it receives the victim address even though the victim block has been removed from the Scache. The bank conflict check in the Scache prevent the merge buffer from writing to the displaced Scache block.

Probe induced victim. (CHECK WEBSITE)

Table 11–22 Victim Command Summary

SC command	SC VSD	Victim command 1 (cmd1)	Victim Command 0 (cmd0)	Extract MGB Data	CAM the MAF[1]
LRU Displacement					
CS_MAF_SC_CMD_LRUEVICT	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_NOOP	No	No
	100	CS_VAF_CMD_NOOP	CS_VAF_CMD_VICTIM_ CLEAN	Yes	Yes
	101	CS_VAF_CMD_NOOP	CS_VAF_CMD_VICTIM	Yes	Yes
Internal Probe					
CS_MAF_SC_CMD_VICTIM	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_NOOP	No	No[2]
	100	CS_VAF_CMD_NOOP	CS_VAF_CMD_VICTIM_ CLEAN	Yes	
	101	CS_VAF_CMD_NOOP	CS_VAF_CMD_VICTIM	Yes	
CS_MAF_SC_CMD_VICTOSHR	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_NOOP	No	
	100	CS_VAF_CMD_NOOP	CS_VAF_CMD_VICTIM_ CLEAN_TO_SHR	Yes	
	101	CS_VAF_CMD_NOOP	CS_VAF_CMD_VICTIM_TO_ SHR	Yes	
CS_MAF_SC_CMD_INVAL	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_NOOP	No	
	100 / 101	Must not happen			
External Probe					

Table 11–22 Victim Command Summary (Continued)

SC command	SC VSD	Victim command 1 (cmd1)	Victim Command 0 (cmd0)	Extract MGB Data	CAM the MAF[1]
CS_MAF_SC_CMD_FETCHFWD	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_FOWARD_ MISS	No	No[3]
	100	CS_VAF_CMD_BLK_ INVAL	CS_VAF_CMD_FORWARD_ ACK_SHR	Yes	
	101	CS_VAF_CMD_BLK_ INVAL	CS_VAF_CMD_VICTIM_ACK_ SHR	Yes	
CS_MAF_SC_CMD_READSFWD	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_FOWARD_ MISS	No	
	100	CS_VAF_CMD_BLK_S HARED	CS_VAF_CMD_FORWARD_ ACK_SHR	Yes	
	101	CS_VAF_CMD_BLK_S HARED	CS_VAF_CMD_VICTIM_ACK_ SHR	Yes	
CS_MAF_SC_CMD_READFWD	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_FOWARD_ MISS	No	
	100	CS_VAF_CMD_BLK_S HARED	CS_VAF_CMD_FORWARD_ ACK_SHR	Yes	
	101	CS_VAF_CMD_BLK_S HARED	CS_VAF_CMD_VICTIM_ACK_ SHR	Yes	
CS_MAF_SC_CMD_READMFWD	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_FOWARD_ MISS	No	
	100	CS_VAF_CMD_BLK_E XCL	CS_VAF_CMD_FORWARD_ ACK_EXCL	Yes	
	101	CS_VAF_CMD_BLK_ DIRTY	CS_VAF_CMD_FORWARD_ ACK_EXCL	Yes	
CS_MAF_SC_CMD_READMFWD requester is IO proc.	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_FOWARD_ MISS	No	
	100	CS_VAF_CMD_BLK_ EXCL	CS_VAF_CMD_FORWARD_ ACK_EXCL	Yes	
	101	CS_VAF_CMD_BLK_ EXCL	CS_VAF_CMD_VICTIM_ACK_ EXCL	Yes	
CS_MAF_SC_CMD_ITODFWD	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_FOWARD_ MISS	No	
	100	CS_VAF_CMD_INV_ TO_DIRTY_RESP	CS_VAF_CMD_FORWARD_ ACK_EXCL	No	
	101	CS_VAF_CMD_INV_ TO_DIRTY_RESP	CS_VAF_CMD_FORWARD_ ACK_EXCL	No	
CS_MAF_SC_CMD_ITODFWD requester is IO proc.	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_FOWARD_ MISS	No	
	100	CS_VAF_CMD_INV_ TO_DIRTY_RESP	CS_VAF_CMD_FORWARD_ ACK_EXCL	Yes	
	101	CS_VAF_CMD_INV_ TO_DIRTY_RESP	CS_VAF_CMD_VICTIM_ACK_ EXCL	Yes	
CS_MAF_SC_CMD_SHRINVAL	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_INVAL_ACK	No	
	100 / 101	Must not happen.			
CS_MAF_SC_CMD_SHRINVALB	0XX / 110	CS_VAF_CMD_NOOP	CS_VAF_CMD_SPCL_INVAL_ ACK	No	
	100 / 101	Must not happen.			

Table 11–22 Victim Command Summary (Continued)

SC command	SC VSD	Victim command 1 (cmd1)	Victim Command 0 (cmd0)	Extract MGB Data	CAM the MAF[1]
Non-Block Responses					
CS_MAF_SC_CMD_STODSUCC	0XX	CS_VAF_CMD_NOOP	CS_VAF_CMD_VICTIM_ CLEAN	Yes	Yes
	110	CS_VAF_CMD_NOOP	CS_VAF_CMD_NOOP	No	No
	100 / 101	Must not happen.			
CS_MAF_SC_CMD_STODPROB	0XX	CS_VAF_CMD_NOOP	CS_VAF_CMD_SHR_TO_ DIRTY_RELEAS	No	No
	110	CS_VAF_CMD_NOOP	CS_VAF_CMD_SHR_TO_ DIRTY_COMPL	No	No
	100 / 101	Must not happen.			

1 CAM the MAF to check whether the victim block is coherent.

2 Internal probes are processed only when the block is coherent.

3 If the block is not coherent, we send ForwardMiss in response to a *Forward.

11.7.6 Retry Flow

11.8 Special Support

11.8.1 Input – Output

I/O request is reference to blocks in the I/O portion of the physical address space (i.e. PA<47> = 1). In contrast to the memory address space, both I/O read and I/O write may have side effects, and data may change without having been written. Since I/O space behaves differently from the memory space, (1) I/O blocks are not cached, (2) I/O requests may not be issued speculatively, and (3) I/O requests must follow the same order given by the program order.

11.8.1.1 I/O Request Ordering and Merging

- The Mbox maintains the I/O ordering. Mbox will send a new I/O request only after the previous I/O request for the same thread is sent out to the system. Cbox is responsible for notifying Mbox when Cbox launches an I/O request to the system by sending the thread processing unit(TPU). For WRIO requests, Cbox is also responsible for notifying Mbox when the IOWrAck is received so Mbox can retire Memory Barrier (MB).
- RDIO and WRIO from the same thread get merged in Mbox. WRIO get merged in the merge buffer while RDIO get merged by (YTD). The MAF provides Mbox with the physical address CAM to assist the I/O request merging in Mbox.
- if (m%miss_pa<47:6> == pa[i]<47:6> &io_ok_to_send[i] = 0)

  merge the new I/O request & return the merged MAF index.

else

allocate MAF entry & return the new MAF index.

- The merging window (i.e. io_ok_to_send) is managed by Mbox.
- We may have only ONE I/O request per thread waiting for the system launch.
- I/O requests whose merging window is open does not bid for the system request.
- Mbox does not send multiple I/O requests to the same block from the different threads unless the merging window is closed (i.e. no merging across threads).
- I/O requests to the same block from different thread are not merged.
- There will be no I/O requests from the Ibox.

11.8.1.2 I/O System Request

- Both I/O read (RDIO) and I/O write (WRIO) are queued in the miss address file (MAF) waiting to be system launched.
- Since WRIO has a victim-like flow, we may consider putting WRIO requests into the VAF???
- I/O read and I/O write transfers can be variable length depending on the instruction size (Quad-word, Long-word, Byte). I/O packets contain the byte mask to accommodate variable length transactions.
- MAF may have the maximum of 4 I/O requests (one for each thread) to be system launched at any given time. To save the MAF width, I/O specific flags will not be kept in the MAF. Instead we have a small 4-entry buffer to store IO_mask<7:0> and IO_size<1:0>. This is possible because once we send out I/O request, we do not need to keep those flags and data for RDIO or WRIO. Since we have one pending I/O request per thread, we use the thread ID of the pending I/O request to address the 4-entry buffer.
- For a WRIO, we store the data block in the victim data buffer (VDB). When Mbox does WRIO, Mbox sends the request along with the physical address to the MAF (or VAF --- YTD). At the same time Mbox sends the data to the VDB. When we are ready to launch the WRIO to the system, we read the data out of VDB. We reserve the first 4 VAF/VDB entries for WRIO data.

11.8.1.3 Others

- We victimize a cache block at fill time and each MAF system request may create a victim. In order not to stall a fill, we need to guarantee a VAF spot at fill time. This is implemented by stalling a new system request when the number of empty VAF entries becomes less than r equal to the number of outstanding system requests.
- The 21464 does not provide any special hardware support for WRIO handshaking. Any synchronization of I/O is done by software using memory barriers (MB).
- Ldx_L/Stx_C is not supported in I/O address space.

11.8.1.4 I/O Request Flow

```
&/* Allocation */
```

```
If (m%io_fill_rqst_valid && c%maf_full)
{
     ask Mbox to retry;
}
else if (m%io_fill_rqst_valid && !c%maf_full)
{
     allocate MAF entry;
     write PA & flag bits;
     allocate Pending Queue entry and arbitrate for the system launch;
}
/* I/O request system launch */
if (PA<47>[to_be_sys_launched_idx] == 1)
{
     if (io_rqst_type[thread_id] == 1)
     { /* read I/O */
          read PA<47:6>[to_be_sys_launched_idx];
          IO_mask<7:0>[thread_id], IO_size<1:0>[thread_id];
          send them to the router;
     }
     else
     { /* write I/O */
          read PA<47:6>[to_be_sys_launched_idx];
          read IO_mask<7:0>[thread_id], IO_size<1:0>[thread_id];
          read VDB[thread_id];
          send them to the router;
     }
}
/* MAF de-allocation, no cache coherence check is necessary */
for (i = 0; i < MAX_MAF_ENTRIES; i++)
{ /* we could follow the same routine as RdBlk request if desired */
/* this will require the marker to be sent for RDIO as well as WRIO */

     if (RDIO && have_data[i])
     {
          deallocate MAF and PQ entries;
     }
     if (WRIO && have_marker[i])
     {
          deallocate MAF, PQ, VAF, and VDB entries;
     }
```

}

11.8.1.5 I/O Specific Structures/Operations

- MBOX
  - Merge Buffer.
  - Memory Barrier.
  - RDIO merge.
- Interface
- CBOX
  - MAF allocation and system request

    I/O requests to the same block get merged in the MAF until Mbox closes the merging window. Then the closed MAF entry bids for the system request pipe.

    When the I/O request is picked, we send the physical address along with the I/O mask and the MAF index to the Router. If the request is WRIO, the data block is read from VDB and is sent to the Router.

    At the same time as we send the selected I/O request to the Router, we also send the thread ID of the I/O request to the Mbox so that Mbox may send a new I/O request for the thread.

    We can de-allocate the MAF entry when the requested BlkIO and WrIOACK are received for RDIO and WRIO respectively.
  - I/O buffer.

    The small 4-entry buffer, one entry for each thread, contains I/O request-specific control flags. The control flags are loaded when a new MAF entry is allocated for I/O request and are sent to the router along with the physical address when the I/O request is picked for the system launch.

Name	# Bits	Description
Valid	1	May not need this ???
IO_MASK<7:0>	8	Byte mask
IO_SIZE<1:0>	2	QW, LW, BYTE
IO_RD_WR	1	0: WRIO, 1:RDIO
TOTAL	12	

11.8.1.6 I/O System Request Timing

I/O system request follows the same path and timing as ones for system requests for memory space.

C1	C2	C3A	C3B	C4A	C4B
1. WR MAF 2. WR I/O buffer	1. Set MAF Valid bit. 2. WR PQ.	1. Arbitrate.	1. RD MAF idx and send to the MAF.	1.Decode MAF idx.	1. RD PA. 2. RD I/O flags and data. 3. Drive them to Router.

11.8.1.7 I/O Request Packet Format

Table 11–23 I/O Request Packet Format

Channel	Length (phit)	Command Name	Packet Format	Description
QIO	3	IORdBytes	PIO	I/O read with byte mask.
QIO	3	IORdLWs	PIO	I/O read with longword mask.
QIO	3	IORdQWs	PIO	I/O read with quadword mask.
QIO	3 + 15	IOWrBytes	PIO	I/O write with byte mask
QIO	3 + 15	IOWrLWs	PIO	I/O write with longword mask.
QIO	3 + 15	IOWrQWs	PIO	I/O write with quadword mask

11.8.1.7.1 Read I/O (RDIO)

Command	System Request	System Response
RdBytes RdLWs RDQWs	PA<47:5> Mask<7:0> MAF IDX<5:0> Processor ID	BlkIO or NXMResp

NAMResp indicates the request referenced a non-exsistent block. This command is a possible response to RdBlk, RdBlkMod, RdBlkShared, and RdIO.

11.8.1.7.2 Write I/O (WRIO)

Command	System Request	System Response
WrBytes WrLWs WrQWs	PA<47:5> Mask<7:0> MAF IDX<5:0> Processor ID DATA	WrIOAck or WrIONack

11.8.2 Memory Barriers — the MB Instruction

A memory barrier retires when all prior (program order) memory references are visible to the whole system. This is managed by MBox.

MBox retires memory barriers after prior loads have retired and after prior stores have obtained modify permission of the cache block they require. (The store may still be in the write buffer, and if an invalidate arrives, this store may never get made.)

IO writes return a 'coherency' or 'completion' message once they have been made visible to the whole system, and this coherency message must be passed on to MBox so it can retire following MB's. Regular stores must have been retired to the Merge Buffer, and we must have obtained the ownership marker and data for the block which they write before a following MB can retire. We do not need to wait for all the coherency markers to have been received, and the cache block to be coherent, before we retire the MB. (We need to wait for coherency before we can forward or victimise the block).

All other memory transactions have implicit completion marks which MBox sees. Loading shared data has the data as the marker, what about loads returning exclusive? CTD has the state change.

The 21464 speculatively executes ahead of memory barriers, but any speculative loads must be replayed if another processor or thread writes the location before the MB retires. If another processor writes, we will receive an invalidate probe which will cause the load to trap and reply.

CBox sends all invalidate probes to MBox so it can trap any speculated loads in the shadow of an MB. (Not just invalidates hitting the MAF.

11.8.3 Load-Locked Store-Conditional (LDx_L/STx_C) Instruction Processing

The basic LDx_L/STx_C flow is:

5. Mbox executes a LDx_L instruction at retire time and loads the lock address into a TPU specific lock register. The Cbox no longer requires that the LDx_L be forced to miss the Dcache.
6. Mbox executes a STx_C instruction at retire time, and sends a CTODSTC(X) to the Cbox if Mbox does not have ownership of block X.
7. The Cbox may find block X shared, exclusive, or invalid in the Scache, and takes different actions for each.

CTODSTC(X) finds block X shared in the Scache of Processor 0 (P0):

1. Cbox fills block shared to the Mbox, and sends S2DSTC(X) to home.
2. At the home:

a. P0 is on sharing list: home sends S2DsuccessCnt() to P0 and SharedInvals to sharers.

b. Block X is shared or invalid, and P0 is not a sharer (including sharing mask case where P0's group is not a sharer): home sends S2DFail to P0.

c. Block X is exclusive at some other processor: home sends S2DFail to P0.

d. Sharing mask bit for P0 is set: home sends S2DprobCnt() to P0. Home will send SharedInval's if it receives S2Dcomplete from P0.

3. At P0: (3a corresponds to P0 action in response to dir. message in 2a, etc.)

a. Cbox sends CTOD to Mbox when block is coherent.

b. Cbox sends NOOP to Mbox (SharedInval will fail the lock)

c. Cbox sends NOOP to Mbox (SharedInval will fail the lock)

d. If P0 has received a SharedInval, P0 sends S2Drelease to home and NOOP to Mbox. If P0 has not received SharedInval, Cbox sends S2Dcomplete to home and CTOD to Mbox when block becomes coherent.

CTODSTC(X) finds block X exclusive in the Scache of P0:

1. Cbox fills block exclusive or dirty to Mbox (depending on coherence)

CTODSTC(X) finds block X invalid in the Scache of P0:

How does this situation arise? P0's Scache must have displaced block X (if P0 has instead received an inval or forward, the Mbox would have failed the lock.) We cannot just fail the lock on a displacement (due to livelock). We also run into problems if we just victimize an exclusive block that the Mbox has locked because the home will no longer send us invals and we could incorrectly succeed the lock. When we displace a block (due to either LRU eviction or an ECB instruction) for which the Mbox has a lock, we want the home to let us know if another processor takes ownership of the block, so we can fail the lock. Thus when a processor displaces an exclusive or dirty block, we send C_DFILL_CMD_LRUVICTIM.

The Mbox should check the victim PA from the Cbox against the lock registers. If there is a match, the Mbox responds to the Cbox with victim_addr_locked; the Mbox does NOT invalidate the lock registers. When the Cbox sees victim_addr_locked asserted, it sends a VictimToShared message (instead of a Victim) to the home. This message will cause the home to add P0 to the sharing list for block X, ensuring that P0's lock will get invalidated should another processor succeed its STx_C.

1. Cbox sends ReadModSTC(X) to home.

2. At the home:

a. P0 is on sharing list: home sends BlkExclCnt() to P0 and SharedInvals to sharers.

b. Block X is shared or invalid, and P0 is not a sharer (including sharing mask case where P0's group is not a sharer): home sends S2DFail to P0.

c. Block X is exclusive at some other processor: home sends S2DFail to P0.

d. Sharing mask bit for P0 is set: home sends BlkExclProbCnt() to P0. Home will send SharedInval's if it receives BlkExclComplete from P0.

3. At P0:

a. Cbox sends FILLBLK to Mbox.

b. Cbox sends NOOP to Mbox (SharedInval will fail the lock)

c. Cbox sends NOOP to Mbox (SharedInval will fail the lock)

d. If P0 has received a SharedInval, P0 sends S2Drelease to home and NOOP to Mbox. If P0 has not received a SharedInval, Cbox sends BlkExclComplete to home and FILLBLK to Mbox.

11.8.3.1 Lock Register for Each Thread

- Ldx_L retires when the requested data block is received.
- The lock register is set when the requested data is received, which can be before all the coherences are received.
- Mbox is responsible for clearing the lock register for invalidate probe, write from other threads, and exception.
- Write from the same thread is considered legal and does not clear the lock register.
- A new Ldx_L by the same thread overwrites the lock register causing all previous Ldx_L/Stx_C to fail.

11.8.3.2 Stx_C Issuing

- The lock register is compared with Stx_C address when STx_C retires. If the STx_C address matches the address in the lock register:
  - In the normal mode, if DCache hit then the LDx_L/STx_C succeeds. If DCache miss, then Mbox sends CTDSTx_C request to the MAF (need Scache tag launch & system launch).
  - In the conservative mode, LDx_L/STx_C succeeds if addresses match and we have the ownership of the block. If we do not have the ownership, we request the ownership to the system. LDx_L/STx_C succeeds if the CTDSTx_C request succeeds.
- Ldx_L/Stx_C may work using regular CTD because the home node would send us an Invalidate when it grants the ownership to other processor and the invalidate will clear the lock register which will cause the Stx_C to fail. But Issuing CTD for a Stx_C can cause unnecessary ownership changes and unnecessary data communication, which can have significant performance impact or possible live-lock.

11.8.4 Prefetch/Modify

11.9 IPRs, CSRs, and Error Handling

11.9.1 Required IPRs and CSRs

Cbox CSR's, with the exception of interrupt controls, contain static values that are loaded at system initialization time and do not change while power is on. They therefore do not maintain the speculative/committed protocol that is required of many control registers; software may be required to jump through hoops to write them safely and/or read them accurately.

In general, Cbox registers are accessed as memory-mapped I/O devices, with register identifiers passed along address paths, and contents along data paths (we have yet to decide which quadword of the block).

- Interrupt requests and current level or mask
  - Interrupt mask
  - Interrupt request bits
  - Queue (Read, Delete, Append)
- ECC correction reports (Scache Data, Scache Tag, Router Ports, Memories)
  - Physical Address (not useful in Router port)
  - Syndrome
  - Corrected Block (wrong if double-error)
- Memory configuration
  - Access to presence-detect EEPROM on RIMM
  - Redundant-channel enables
  - Directory state-machine controls
  - Directory/ecc initialization
  - Fairness-scan timer
  - PLL controls
  - Datasheet constants
  - Debug stream controller

    Select debug write mode

    Current debug write pointer

    Debug data read enable
- Router configuration
  - This node number, first NXM
  - Output port to each node
  - Sharing mask for each node
  - Output port for each mask bit
  - ECC correction reports
  - PLL controls
  - Virtual-channel buffer thresholds (depend on link latency)
- Diagnostic control and access
  - Force Scache hit or miss, per set
  - Examine tag
  - Examine selected MAF entry
- Debugging/performance-analysis controls and logs

- Optimization enables:
  - LDx_L can issue RdBlkMod
  - Read/InMemory returns data Exclusive
  - Migratory data prediction
  - Timeout before forwarding ownership
  - Purge controls
  - Uniprocessor (no directory accesses required)
  - Small MP (mask bits uniquely identify processors)

11.9.2 Error Handling

11.9.3 Cbox Deadlock Avoidance Mechanisms

11.10 Profiling Support

11.11 Stuff From Original Cbox Spec Not in Outline

(That I can see anyway....)

Were H1's 10.6 through 10.15 and are now all H2's

Last section was 10.20.9 Cbox Mechanisms

11.11.1 Scache Index (paddr<18:6>) Conflict

Scache index can be a hash function of the physical address. But the hashing introduces the extra delay in the critical path. Our current thinking is that the potential performance improvement does not justify the extra complexity.

In contrast to the 21264 and the 21364, the 21464 allows the Scache to service multiple system requests to the same Scache index concurrently.

- Scache index conflict is resolved by victimizing the LRU cache block at system fill time (i.e. Blk* or InvToDirtyRespCnt).
- The Scache fill schedules both the read and write pipeline and use the read pipe to extract the victim just ahead of the fill data write.
- Current proposal:
  - Cbox read the victim block out of the Scache, performs ECC correction of the victim data, and writes the corrected victim data to the VDB.
  - Cbox does not send the victim data to the Mbox.
  - If the merge buffer has the cache block modified, the merge buffer overwrites modified bytes in the VDB using byte write capability of the VDB. The merge buffer should write the VDB after the ECC corrected victim block has been written to the VDB.

  - Cbox rejects the write-throughs to the banks that have in-flight system fills (i.e.Possible Stale Victim) until the ECC corrected data gets written to the VDB. This can be implemented using the bank conflict check.

- In order not to stall system fills, we need an available victim buffer slot before launching a system fill request (Read*Req or InvToDirtyReq).

  Outstanding system request to the memory space < Available VAF/VDB entries for memory space (4 slots are reserved for WrIo) - 4 (reserved for Probes).

- Mbox is responsible for victimizing a cache block in the Mbox.
- We decided against the proposal to victimize the LRU block at the Scache miss time as well as at the system fill time.
  - Pros:

    Allows the victim process to start early (no need to wait until the fill)

    Reduces the probability of stalling the MAF system fill request pipe.
  - Cons:

    If a block that has been victimized at the miss time is referred again, then we have to send the victim data to the home and bring the data back in again, which may result in increased network traffics.

    A MAF entry may cause two victims, which may require 2X VAF entries.

    Incompatible victim/fill path.

11.11.2 ShrToDirty[STC]Req

Cbox sends a ShrToDirty[STC]Req to the home node when it needs the write permission of a Shared block for a non-speculative store (i.e. merge buffer write). Cbox does NOT send a ShrToDirtyReq for a speculative store.

When the home memory receives a ShrToDirty[STC]Req:

- If the cache block is not owned by other processor, i.e. the directory state is either InMemory or Shared, and the requester is a sharer, the directory sends ShrToDirtySuccessCnt to the requester and SharedInval to other sharers.
- If the directory state is InMemory or Shared but the requester is not a sharer, the driectory sends BlkExcl to the requester and SharedInval to sharers in response to the ShrToDirtyReq. The directory sends ShrToDirtySTCFail in response to a ShrToDirtySTCReq.
- If the directory state is RemoteExcl, the directory sends the ReadModFwd to the current owner. However, the directory DOES not forward ShrToDirtySTCReq and sends a ShrToDirtyFail to the requester.
- If the directory state is SharedMask, the directory node can't tell whether the requester is a sharer or not even though the requester is in the sharing mask.

  e. A sendsReadModReq(X).

  f. B sendsShrToDirtyReq(X).

  g. The directory receives the ReadModReq(X) from A and sends SharedInval (X) to node B.

h. B invalidate the shared block and sends a InvalAck to node A.

i. The directory receives Victim(X)from node A.

j. C sends ReadShrReq to the directory.

k. The directory processes the ReadShrReq from C. If A belongs to the same group as C, A becomes a sharing node even though it does not have the block.

l. When the directory processes the ShrToDirtyReqfrom A, the directory does not know whether A is true sharing node or not.

- For a ShrToDirtyReq and a sharing mask is used, the directory optimistically succeeds the ShrToDirtyReq and sends a ShrToDirtySuccessCnt to the requester. The requester is responsible for resolving the ShrToDirtySuccessCnt. The appropriate action for the requester if the requester does not have the shared copy in its Scache is to do a VictimClean followed by a ReadModReq.
- To avoid a dead-lock, for a ShrToDirtySTCReq when a sharing mask is used, the directory sends a ShrToDirtyProbCnt to the requester. If the ShrToDirtySTCReq succeeds, the requester sends a ShrToDirtyComplete to the directory and then the directory sends SharedInval to sharers. If the ShrToDirtySTCReq fails, the requester sends a ShrToDirtyRelease to the directory.

If the directory receives a *Req from the current owner, it means that the victim is on its way to the directory. The directory must wait for the victim. After the directory receives the victim, it sends a response to the requester.

The EV7 protocol does not forward a ShrToDirty[STC]Req to the current owner. To reduce the latency, the 21464 forwards ShrToDirtyReqs to the current owner but not ShrToDirtySTCReqs.

- If the requester is not a sharer and the directory state is not RemoteExcl, then the DIFT sends BlkExclCnt to the requester.
- If the requester is not a sharer and the directory state is exclusive, the DIFT sends a ReadModFwd to the owner.
- This proposed scheme does not break the assumption that Cbox will not receive a Blk* if the Scache has a copy of the cache block.

ShrToDirtyResp

- ShrToDirtySuccess.
- ShrToDirtyProb.

11.11.3 Scache Tag Launch Pipe

Table 11–24 shows the Scache block state.

Table 11–24 Scache Block State

Scache state							
Valid	**Shared**	**Dirty**	**State**	**Coherent**	**Ownership**	**Mbox Can Write**	**Victim**
0	X	X	Invalid	X	No	No	None
1	0	0	ExclClean	X	Yes	No	VictimCln or VictimClnToShr

Table 11–24 Scache Block State

1	0	1	Dirty	0	Yes	No	Victim or VictimToShr
1	0	1	Dirty	1	Yes	Yes	Victim or VictimToShr
1	1	0	Shared	X	No	No	None
1	1	1	Must not happen				

We have separate pipelines for the Scache tag launch and the write-through operations. In case of the Scache bank conflict, we must stall one pipeline. For the performance reason, we prefer to stall the write-through pipeline.

The Scache pipe is arbitrated with the following priority.

4. Blk*/InvToDirtyRespCnt from system or System fill hiccup recovery.
5. Replays (from the retry queue):
   - Scache bank conflicts.
   - Scache tag ECC error.
   - Scache data ECC error.
6. Internal probes from the Internal probe queue.
7. Probes from the PRQ.
8. The system fill early warning.
9. New Misses from the Pre-MAF.

System fills (Blk* and InvToDirtyRespCnt) may cause victims:

- Send an early warning (fill address) to Mbox if the early warning wins the Scache pipe arbitration. Otherwise, we do not send the early warning incurring extra latency to the Scache miss retry pipe.
- Take 2 cycles of the Scache tag pipe.
- To minimize the cost, we'd like to avoid skidding or bypassing of the system fill:
  - The system fill consists of 2 Scache tag pipes: one for victim extraction and the other for fill.
  - Launch a system fill at even cycles to avoid the bank conflict with the previous system fill.
  - System fill can have the bank conflict with a Scache accesses which is already in the pipe (i.e. 2 cycle earlier). Then the conflicting Scache access gets pre-emptied and must be retried.

Scache retry pipe.

- Any Scache access except for system fill (Blk*) may get retried due to
  - Scache tag ECC error.
  - Scache Data ECC error.
  - Bank conflicts.
- The retry queue entry contains:
  - 6-bit MAF index.

  - 6-bit VAF index.
  - Command type.
- Need 2 ~ 3 write ports since the retry may come from
  - Scache bank conflict.
  - Scache tag ECC.
  - Scache data ECC.
- The retry do not CAM the MAF again but it reads the PA from the MAF.

Internal probe queue

- 64 entry FIFO.
- Timer queue is loaded when:
  - Invalidate: received a Shared block from the system and MAF.mb_retired = 0 & MAF.inval_seen = 1.
  - Victimize:

    Received ownership from the system, the block is coherent, and MAF.victimize = 1 | MAF.vic_to_shr = 1.

    Received EvictBlk or CleanBlk requests from Mbox.
- If Cbox has the pending internal probe which Invalidates the cache block, Mbox must not retire a MB. Cbox sends the per-thread signal (i.e. cs%mb_ret_tpu_c4a_h) to Mbox indicating whether Mbox can retire MB.

ShrToDirty*Cnt from the PRQ.

- A ShrToDirty*Cnt can be retried due to Scache tag ECC error, Scache data ECC error, or Scache bank conflict.
- ShrToDirty*Cnt and subsequent Probe to the same block must be processed in order.
  - Problem: ShrToDirty*Cnt (X) ---> Tag ECC error---> Probe(X)---> ShrToDirty*Cnt (X) retry.
  - Probes to the block that has in-flight Scace transaction (MAF.sc_inflight) stalls in the PRQ until the conflicting Scache transaction passes the retry point.

Misses check the Scache to see if the requested cache block is in the Scache with the required state.

- If Scache has the requested block for requests:
  - Fill the D-fill buffer and/or I-fill buffer.
  - Update the LRU bits in the Scache tag.
  - If Scache has a Shared block for an ownership request, then make a ShrToDirtyReq request to the system.
- If the cache block doesn't exist in the Scache, then the MAF makes a Read*Req to the system.
- Since I/O blocks are not cache-able, I/O requests do not need the Scache tag lookup.

Scache Bank conflicts.

Scache tag ECC error or Scache data ECC error.

- We store the ECC corrected tag(data) in the Scache Tag (Data) ECC register and replay the request.
- The replayed request takes the tag (data) from the Scache Tag (Data) ECC register.
- The ECC registers get cleared if we modifies the Scache tag in the same Scache index (i.e. probe or fill).
- We write back corrected tag into the tag array after the retry accesses the corrected tag using the tag write cycle.
- We do not write corrected data back to the Scache. If the error block gets referred many times, we need software intervention or evict the block.

Arbitration among Miss request is done in the Pre-MAF to:

- Reduces the Dcache miss retry latency.
- Accesses to the Scache follow the program order more closely.

Scache Tag Request Command

Table 11–25 Scache Tag Request Command

Scache pipe Commands	cs%st_req_cmd_c3a_h<3:0>	Notes
Bank conflict, WrIOAck, BlkIO	C_ST_CMD_NOOP	No need for Scache tag.
Ifecth, FetchLine, PfetchLineMod, PrefNocache, PrefScache	C_ST_CMD_MISS	Look up the tag and update the LRU.
FetchLineMod, CtoD[STC], ItoD	C_ST_CMD_SETDIRTY	Set the dirty bit if the specified block is ExclCln.
LRUEvict	C_ST_CMD_LRUEVICT	Victimize the least recently used block from the same Scache index as the specified cache block. The victim set number is used for the fill block in the following cycle.
Victimize, ReadModFwd, InvToDirtyFwd, SharedInval	C_ST_CMD_INVAL	Invalidate the specified cache block.
VictimToShr, FetchFwd, ReadShrFwd, ReadFwd	C_ST_CMD_CTOS	Make the specified cache block Shared.
ShrToDirty*Cnt	C_ST_CMD_STOD	Make the cache block Dirty if shared.
BlkShared	C_ST_CMD_BLKINV	Invalidate the set victimized in the previous cycle.
BlkShared	C_ST_CMD_BLKSHR	Fill block. Use the Scache set victimized in the previous cycle.
BlkExclCln	C_ST_CMD_BLKEXCL	
BlkExclCln, BlkDirty, InvToDirtyRespCnt	C_ST_CMD_BLKDIRTY	

Also see the Scache tag state transition table, Table 11–12.

11.11.4 Probe Processing in Cbox

See Section 6.5.2

11.11.5 Order Dependency

Scache access to the same cache block

Table 11–26 Scache Access Order to the Same Cache Block

	MAF states		
	miss_inflight_in_sc	**probe_inflight_in_sc**	**fill_inflight_in_sc**
Miss from Mbox	Reject the miss request.	Reject the miss request [1].	Reject the miss request [2].
***Fwd from PRQ**	Stall the probe queue [3].	Stall the probe queue [4].	Stall the probe queue [2].
InvalAck/Timer Expiration from PRQ	Stall the probe queue [6].		
ShrToDirty*Cnt from PRQ	Must not happen [5].		
Blk*/I2DRespCnt from system	Must not happen [5]		

Notes:

- [1]: If a ShrToDirty*Cnt (or InvToDirty*Cnt) is in the retry queue, the Miss will see the stale Scache tag and we may send a system request even though we have the ownership of the block. This will break the cache coherence protocol.
- [2]: Stale fill data access:
  - There are 6 ~ 8 cycle separation between the Scache tag update and the Scache data update for a system fill.
  - A Miss(X) from Mbox gets rejected if we have a Blk*(X) in-flight in the Scache to prevent the Miss from accessing the stale fill data.
  - A probe(X) to a stale fill block gets stalled in the PRQ.
  - A Blk* must not get evicted before the fill data gets written into the Scache.

    We have a stale fill table to prevent the Scache tag control from evicting a stale fill block..
- [3]: The probe can proceed even if there is a miss request in-flight in the Scache. But to simplify the design, we will stall the probe queue.
- [4]: Probes to the same cache block must be serviced in order.
- [5]: The miss request gets merged and do not enter the Scache pipe if we have an outstanding system request for the cache block.
  - If Cbox gets a Load miss request from Mbox to a cache block which has an outstanding ShrToDirtyReq to the system, Cbox will not fill the shared block to Mbox until it receives a response for the ShrToDirtyReq even though the Scache has a shared copy.
  - This case happens if the shared block which was filled at the time when Cbox made the ShrToDirtyReq has been evicted from Mbox. In addition the load can't retire until the store which needs the ownership gets retired. So we believe the performance impact will be minimal. Alternatively we could allow the load miss request enter the Scache pipe even if we have an outstanding ShrToDirty-Req to the system. However this may add a significant complexity.

- [6]: This is not required to be functionally correct. But to simplify the de-allocation of MAF entry, we do not allow changing the MAF states if the MAF entry has in-flight Scache transaction.

11.11.6 Possible Race Conditions and Other Concerns

- For the system launch, or system fill, we read out the PA and control flags. We must not change control flags while we try to read them out:
  - Read the physical address in B-phase
  - Change control flags in B-phase..
  - Read control flag bits a phase later in A-phase.
- We must not fill the I-fill buffer twice for a MAF entry since the I-FB entry may get recycled.

11.11.7 CBox mechanisms

- IO write 'coherency mark' to MBox
- All Invalidate or ForwardInvalidate probes to LQ

12
Cache Coherence Protocol Processing

The 21464 adopts the 21364 cache coherence protocol with small enhancements. The protocol is a directory based CC-NUMA and tolerates out-of-order channels except for the I/O channel, thereby supporting an adaptive packet routing.

12.1 Introduction to the Protocol

The coherence protocol is the mechanism by which large numbers of processors maintain a consistent image of the contents of memory, as required by the Alpha SRM.

Small-scale multiprocessors like Turbolaser maintain consistency by monitoring a bus, so that all caches observe all memory transactions; this approach does not work well for larger numbers of processors because the bus becomes physically too large to be fast, and because the number of external transactions that each cache must process grows with the number of processors.

Mid-scale systems like Wildfire depend on a central switch to impose the same order as a bus, and require that messages be kept in order along their communication paths. The directory serves as a filter to minimize the traffic to any particular node, but maintaining order is costly and inefficient, and limits the scalability of these systems.

Larger systems, such as the 21464, try to avoid dependence on any single resource for a number of reasons, including reliability and load-distribution. Further, they use nondeterministic routing, to make the best use of available network resources. This means that two messages can take different paths and get out of order, even if they start and end at the same nodes.

The protocol is designed to ensure that all processors which cause and/or observe changes in memory see those changes occur in the same apparent order, even though the messages between processors and memories may get out of order along their way. The order observed by all processors is the order in which requests are serviced in their home memory, and in particular, in the DIFT, a control module in the memory controller. Caches communicate with the DIFT as they manipulate memory data, and the DIFT delays multiple requests for any individual block until it has coordinated previous requests with any caches affected by those requests.

There are two major states in which caches hold data (each with a number of minor variants): Shared, meaning that any number of processors can read the data, but none can write it, and Exclusive, meaning that there is exactly one processor with a valid copy, and that processor is permitted to read or write it. The so-called Invalid state

means that there is no valid copy of the block in any cache; the name does not refer to the validity of the block in memory. References to Invalid are being replaced by Local, which is more accurately descriptive.

The protocol, as managed by the DIFT, is concerned with the transitions between states, and with performing the transitions in such a way that as much of the communication latency as possible is kept out of the critical paths.

Whenever a block is held Exclusive by some processor (which we refer to as the owner), and another processor needs access, the protocol requires the DIFT to tell the owner to give up Exclusive state, and the owner to report to the DIFT when it has done so. Until the DIFT hears back from the owner, the DIFT does not process other requests for the block.

On the other hand, when a block is held Shared by some processor(s), the DIFT can permit any number of other processors to read the block, until one needs write access. At that time, it notifies all processors which have read the block to invalidate it in their caches, and they report to the new owner, rather than the DIFT, when they have done so. The new owner must not release exclusive access until it receives acknowlegement that all previously-existing copies of the block have been invalidated.

> During the transitions when Exclusive access is passed from one processor to another, there can be periods during which two processors believe that they have exclusive access; in some circumstances it is possible for the "second" processor to complete its writes and send a victim block to the memory which arrives before that sent by the first writer. This rare event is called a dual-victim race, and is sorted out by special rules in the DIFT (see Section 12.4).

The memory system is designed with the expectation that a disproportionate fraction of the memory traffic produced by any processor will be addressed to its own local memory; this is true for most multiprocessor applications, though precisely how much is highly application-dependent. We use this fact, and the on-chip communication between a cache and its local controller, to optimize references to the local memory. The directory cache optimizes the directory accesses for requests both from local and remote processors. The on-chip directory cache stores the directory information of most frequently used cache blocks to minimize memory accesses for directory information. Requests from the local Cbox as well as remote processors update the directory, thereby eliminating needs for the LPR.

12.2 Structures that Maintain the Cache Coherence

The Cbox maintains the cache coherence with the following structures.

- Miss address file (MAF)
- System request pending queue (SRQ)
- Victim buffer
  - Victim address file (VAF)
  - Victim data buffer (VDB)
- Probe queue (PRQ): probe queue
- Directory In-Flight Table (DIFT)

- Scache tag array (Sbox) **(not described)**
- Scache data array (Sbox) **(not described)**

12.2.1 Miss Address File (MAF)

The Scache is a non-blocking cache, which means that it continues to accept new requests while waiting for responses from the system resulting from previous misses. In order to associate those responses with the original requests, and to know how to process each response, the Cbox records each request in the MAF as processing begins, and compares each request address against the addresses of all outstanding requests to detect multiple requests to the same block and avoid conflicts. The MAF contains:

- Miss Requests from Ibox/Mbox, both to local addresses and to remote addresses.
- Probes that are in-flight in the Scache pipe.
- Coherence states that are associated with the outstanding request.

12.2.2 System Request Queue (SRQ)

Requests which miss in the Scache must be serviced by memory, either local or remote, but the paths to memory cannot accept requests as rapidly as they can be generated. The system request pending queue (SRQ) stores the MAF index of requests which are waiting to be sent to memory. As buffers become available, the SRQ arbitrates among pending requests round-robin among threads, then FIFO within thread, to select which request to launch next. The current proposal is to do away with the thread round-robin since Mbox does consider thread fairness for load retry. Hence the system request pending queue is a simple 64-entry FIFO queue.

12.2.3 Victim Buffer

The Scache retains most blocks until the space they occupy is needed for another block. If a block is not held exclusively in this cache at the time it is evicted, then it is simply overwritten, but if it is in exclusive state, then the directory must be notified that this cache is releasing exclusive access, and if the block is dirty, it must be written back to memory. Rather than delaying the fill which overwrites this block, the Scache moves the old contents to the victim buffer, where it waits before being sent. The victim buffer contains:

- Victim, VictimToShr, VictimCln, and VictimClnToShr, which can be sent to the home memory.

  To minimize the number of *Forwards, EV8 is considering adding the Purging option to the protocol where Cbox gives up the ownership of the cache block voluntarily and send Victim[Cln]ToShr to the remote directory. Notice that if the cache block is in the local memory space, there is no benefit of purging. EV8 decided against the purging option since performance simulation indicated the purging provides no significant benefits.

- Response to the home memory:
  - VictimAckShr, VictimAckExcl, FwdAckShr, or FwdAckExcl to the remote memory or to the local memory.
  - FwdMiss to a remote or local memory.

  - ShrToDirtyComplete or ShrToDirtyRelease to a remote or local memory.
- Response to the remote requester
  - Blk* to be sent to the remote requester in response to a forward.
  - InvToDirtyRespCnt to a remote requester.
  - InvalAcks to be sent to a remote requester.

12.2.4 Probe Queue (PRQ)

The Scache must service several kinds of requests from other nodes in the system. For some of these requests, it is important that probes to the same cache block be serviced in the same order in the Scache as they are in the DIFT, so they are all stored in the probe queue, called the PRQ, while awaiting service in the Scache. The protocol does not mandate keeping the same order of the probes in the Scache as in the DIFT. The probe queue is a simple 24-entry FIFO. The PRQ contains:

- *Forward: Forwards from remote and local directories.
- Response without a fill data.
  - ShrToDirtySuccessCnt.
  - ShrToDirtyFail.
  - ShrToDirtyProbCnt.
  - InvalAck.
  - WrIoAck.
  - WrIoNack.
  - NXMResp/ERRResp.
- Since the InvalToDirtyRespCnt must allocate a Scache set, it takes the same path as the fill block (i.e. Blk*) taking two Scache cycles. Since the InvToDirtyResp does not accompany the data, the Scache has an invalid data until the merge buffer completes the write-through. It will be functionally incorrect if Ifecth grabs the data out of the Scache before the merge buffer writes the cache block. There are two proposals:
  - Option 1 (current favorite): Have Ifecth probe the merge buffer. If the merge buffer has the block, then notify the Ibox and retry the Ifetch.
  - Option 2: always bypass the InvToDirtyResp straight to the victim buffer and have the merge buffer write the cache block to the victim buffer. This scheme prevents from filling Icache with the invalid cache line but will require victimizing the cache block and re-requesting the block. Since we often read the cache block after store, this scheme has some performance impact.
- Remote requests, Local requests, and local victims no longer come to the probe queue for the new EV8 protocol.
- To avoid a deadlock problem, the PRQ takes:
  - Any message if there are more than one free entries.
  - Only a Response from the router if there is exactly one free entry.
  - No transaction if there is no free entry.

- 2~4 VAF entries are reserved for probes.

12.2.5 DIFT

The memory controller may have a large number of requests in progress at any moment, some being serviced in the local memory, others awaiting responses from remote nodes. All requests are recorded in the DIFT as soon as they arrive, and removed when the transaction is complete. For a any given cache block, there is one active transaction at any given time. In other words, the DIFT will not process another request to the same cache block until the current transaction is complete.

12.3 Overview of the Cache Coherency Protocols

12.3.1 Comparison Between 21363 and 21464 Cache Coherence Protocols

Table 12–1 summarizes the differences between the 21363 and 21464 cache coherency protocols.

Table 12–1 Comparison Between 21364 and 21464 Cache Coherence Protocols

Assumptions	21364	21464
Have multiple outstanding system requests to the same Scache index.	No	Yes
A dirty block can't be sent in response to a request from an I/O device. This requires a snarf (i.e. VictimAckExcl) when a modified cache block is to be sent to an I/O device.	Yes	Yes
Cbox may send a VictimToShr or a VictimClnToShr to its local memory.	No	Yes[1]
A successful ReadMod, ShrTodirty[STC], or InvToDirty never results in a VictimCln or a VictimClnToShr. This means that the cache block is considered dirty even if the block is not modified.	Yes	No[2]
The current owner sends a BlkExclCnt(0) to the requester and a VictimAckExcl to the home in response to a ReadFwd when the block is dirty. Otherwise, it must send a Shared block to the requester and a FwdAckShr to the home.	Yes	No[3]
The directory doesn't have the requester for a ShrToDirtyReq as a sharer and no one including its own Cbox owns the block exclusive.	ShrToDirtyFail—> requester.	BlkExclCnt —> requester.
ShrToDirtyReqs get forwarded as ReadModFwds.	No	Yes
To simplify the DIFT design, Cbox can't respond with a FwdAckExcl in response to a forwarded request that originated from an I/O device.	Yes	No[4]
Cbox never responds FwdAckExcl to any ReadForward[3].	Yes	No
Send a BlkExclCnt to the requester and a a VictimAckExcl to the directory in response to a ReadReq or ReadFwd if the block is modified.	Yes	No[3]
Local memory references update the directory state[5].	No	Yes

[1] In order to support the CleanCacheBlk instruction, the processor wants to give up the ownership voluntarily but keeps a Shared copy, Cbox must be able to do a VictimToShr or a VictimClnToShr. Scache will send VictimToShr or VictimCleanToShr if the block addressed by CCB is in dirty or exclusive clean state, respectively (it will send nothing if there is a miss or the block is in shared state).

[2] For the 21464, a BlkExcl is filled as BlkDirty if the MAF state indicates a merge buffer write which is non-speculative and the block is coherent. Otherwise, it is filled as ExclClean block and Mbox is not allowed to write the cache block. Mbox will send CtoD to Cbox to gain write permission.

[3] The 21464 is considering a selective migration option where an exclusive block can be passed in response to a ReadFwd. But it is unlikely that the 21464 will adopt the 21364 migration scheme. The current default is to send BlkShared to the requester and a VictimAckShared to the directory.

[4] This restriction is imposed on the 21364 to avoid read-modify-write of the directory for a VictimClean from a new owner (if FwdAckExcl comes after the VictimClean, then the DIFT entry must do Read - Write - Read - Write which makes the DIFT state machine complicated). Currently, the 21464 is planning to have one-to-one correspondence between the DIFT and the fill buffer. Hence, the VictimClean case above results in Read - Write - Write which is identical to a Victim.

[5] For the 21364, local references do not update the directory to minimize the memory access for directory updates. Instead the cache coherence is maintained by forcing all remote requests to probe the local caches. EV8 has an on-chip 256KB directory cache to store the directory states of the most frequently used cache blocks. The directory cache significantly cut the memory access for both local and remote requests. Thanks to the directory cache, the 21464 local references update the directory thereby eliminating local probes.

12.3.2 Onchip Directory Cache

The 21464 has an onchip directory cache :

- Unlike the 21364, local references DO change the directory states thereby eliminating needs for local Cbox probes.
- No local victim and local request race.
- The DIFT is responsible for keeping order between requests, both local and remote requests. So local *Req, *Req, and local victim* do not come to the probe queue.

12.3.3 Coherence Messages are Split into Three Types

- Requests (*Req)

  All requests come to the home node and the home node is responsible for either responding to the requests or forwarding the request to the current owner including its local Cbox.

- Forwards (*Fwd) from home nodes

  Cbox receives *Forward both from remote directories and the local directory.

  - Cbox may not receive another Forward for the same cache block until Cbox sends a response to the previous Forward to a cache block because the directory does not process another request to the same cache block until it receives a response from the previous owner for the *Forward.
  - For *Forward, Cbox forwards the cache block to the requester and sends a response to the directory. However for the following cases, Cbox sends a Fwd-Miss to the directory:

    Cache block is not coherent

    Cbox did not receive the Blk* (i.e. The exclusive block is on its way to us).

    Cbox did not receive all the InvalAcks from sharers.

For theses cases, Cbox victimize the cache block when the cache block becomes coherent.

The Scache doesn't have the block (i.e. the victim block is on its way to the directory).

- Responses

12.4 Protocol Races

- Victim races
  - Remote Victim race

    Cbox does not have to maintain the ordering between a request to a remote memory and a remote victim. Remote victim races are handled by the directory. If the directory receives the *Req before the Victim*, since the *Req is from the current owner, the DIFT knows that the Victim* is on its way and waits for the victim before servicing the *Req.
  - Local victim race

    Since local requests update the directory, the directory knows the current owner, including the local processor, of a cache block. Cbox no longer has to maintain the ordering between local requests and local victims.
  - Dual Victim race

    A Victim* from the new owner arrives before the forward acknowlegement arrives from the previous owner in response to the *Forward. The DIFT is responsible for resolving the race.
- Early Forward race

  A forward arrives while there is an outstanding system request or not all InvalAck's have been received. The early forward race can happen due to either:
  - The exclusive block is in its way to us and the *Forward is to the the yet-to-be received block.
  - We victimized the block. The directory forwarded a request to us. Before we received the *Forward we send the system request asking for the cache block.

  Send a ForwardMiss to the home memory and victimize the cache block when the cache block arrives or becomes coherent.
- Late Forward race

  A *Forward arrives after we victimized the cache block. We simply send a ForwardMiss to the home memory for this case.
- Early InvalAck race

  An Invalack arrives before the *Cnt arrives. Like the 21364, EV8 has the coherence counter which can keep count of early InvalAck's.
- Early SharedInval race

  A SharedInval arrives while we have the outstanding Read[Shr]Req. The early SharedInval race happens due to one of the following:

- The Shared block is in its way to us and the SharedInval is to the returning block.
- The SharedInval is to the previous copy of the block we already displaced from the Scache.
- The SharedInval is to the previous copy of the block which has been invalidated before. When the sharing mask is used, we can receive the SharedInval if another processor which shares the same sharing mask bit is invalidated even though we do not have the block.

When the Shared block arrives, we must not blindly discard the block to avoid potential live-lock problem.

- If the fill block is after the MB retire, we must discard the block and re-send the system request.
- If the fill block is before the MB retire, we fill Mbox with the block and invalidate the block.

- Wrong SharedToDirtySuccess race

  A SharedToDirtySuccess finds no shared copy in the Scache. The wrong SharedToDirtySuccess race happens due to either:

  - The shared copy has been invalidated. But another processor which shares the sharing mask sends a Read[Shr]Req thereby setting the sharing mask bit. Hence when the directory receives a ShrToDirtyReq and the sharing mask is used, the directory doe not know whether the requester is a true sharer or not. The directory optimistically succeeds the SharedToDirtyReq and the the requester is responsible for resolving the problem by doing VictimClean first them sending a ReadMod.
  - The shared copy has been displaced out of the Scache. We bypass the ShrToDirtyResp directly to the victim buffer and extract the merge buffer into the victim buffer thereby making forward progress.

  We can distinguish two cases by recording SharedInval in the MAF. In other words, if we receive a ShrToDirtySuccess and the MAF.inval_seen bit is set, then we know the ShrToDirtyReq was incorrectly granted.

  To avoid live-lock, the directory can't not optimistically succeed a SharedToDirtySTCReq when the sharing mask is used. Instead the directory sends a ShrToDirtyProb to the requester but do not send SharedInval to sharers. If the SharedToDirtyProb succeeds, the requester sends a ShrToDirtyComplete to the directory and the directory sends SharedInval to sharers. If the SharedToDirtyProb fails, then the requester send a SharedToDirtyRelease to the directory.

12.5 Probe Processing

- Probe Pipeline Stages
- Unlike the 21364 protocol, the directory sends a SharedInval to its local Cbox since remote requests (*Req) no longer come to Cbox.
- Responses without a fill block from the PRQ :
  - Changes the states of corresponding MAF entries.
  - Updates the Scache tag.

- Unlike a Blk*, it can be replayed if it encounters a Scache tag ECC error or a Scache bank conflict.

- A ShrToDirtySuccessCnt does not find the shared cache block in the scache.
  - The previous proposal was to do a VictimClean and send a ReadModReq if the request was not StxC. If the request was for a StxC, then we would send a Stx-Fail message to Mbox. However, this creates a potential live-lock.
  - The current proposal

    If the MAF.inval_seen bit is set:

    Do VictimClean and send a ReadModReq if the request was not StxC.

    If the MAF.inval_seen bit is not set:

    This means the shared block has been displaced from the Scache.

    Send the StxCSuccess to Mbox.

    Extract the merge buffer entry to the VDB.

    Send Victim to the home node.

    Cbox sends a ReadShrReq if it has a pending Ifetch to the cache block.

- A ShrToDirtyProbCnt does not find the Shared block in the Scache.
  - If the MAF.inval_seen bit is set:

    This means the shared block was invalidated.

    Send a ShrToDirtyRelease to the home node.

    If we have pending non StxC request, send a system request.
  - If the MAF.inval_seen bit is not set:

    This means the shared block has been displaced from the Scache.

    Send the StxCSuccess to Mbox.

    Extract the merge buffer entry to the VDB.

    Send ShrToDirtyComplete and Victim to the home node.

    Cbox sends a ReadShrReq if it has a pending Ifetch to the cache block.

- The only case where Cbox send a StxCFail to Mbox is when Cbox gets a CtoDSTC from Mbox and we do not have the block in the Scache.
- Cbox does not send STCFail message to Mbox when Cbox receives a ShrToDirty-Fail from the directory since a SharedInval must have failed the StxC.
- Forwards from remote directories need the Scache pipe:
  - To update the Scache tag.
  - To send a response to the requester and the directory.
- To avoid a dead-lock, we reserve four VAF and four MAF slots for probes.

12.6 Coherence State

Cbox implements a coherence protocol which ensures that all processors have a consistent image of memory.

Table 12–2 MAF Coherence State Bits

Name	Meaning	Victim to Home	Set By
MAF.coherent	The exclusive cache block is coherent.		
MAF.coh_cnt<5:0>	The number of coherence messages received.		InvalAck, *Cnt.
MAF.timer_on[1]	Has a timer queue entry.		
MAF.sc_inflight	Has an inflight transaction in the Scache pipe.		Forward, ShrTo-Dirty*Cnt, SharedInval.
MAF.victimize	After the ownership is received and the block becomes coherent, the cache block must be victimized (i.e. invalidate the cache block and send a Victim or a VictimCln to the directory). If a BlkShared is received, Cbox may keep the Shared copy.	Victim or Victim-Cln	Remote Request (MemFwdMiss) or Forward (ForwadMiss)
MAF.vct2shr	After the ownership is received and the block becomes coherent, the cache block must be changed to Shared. Cbox should send VictimToShr or VictimClnToShr to the remote home directory.	VictimToShr or VictimClnToShr	Forward (ForwadMiss)
MAF.inval_seen[2,3]	SharedInval has been received before the fill block.	None	SharedInval

[1] The timer is set due to either: 1) An exclusive fill block is received, the block is coherent, and either MAF.victimize or MAF.vic2shr bit is set; or, 2) A shared fill block is received and the MAF.inval_seen bit is set.

[2] This bit is set if we receive a SharedInval and have an outstanding system request. If a BlkShared is returned and the MAF.inval_seen is set, then we do not know whether the block is before the SharedInval or after the SharedInval. To be conservative, the block has to be invalidated.

If the MAF.mb_retired is set, discard the fill block.

If the MAF.mb_retired bit is no set, fill the Mbox with the block and when the timer expires, invalidate the block.

If BlkExclCnt or BlkDirty returns, then we know the block is after the SharedInval and we fill the Scache with the exclusive block.

[3] If we receive SharedToDirtySuccess or SharedToDirtyProb and the MAF.inval_seen is set, then the the SharedToDirty has been incorrectly granted.

Notes:

- Can more than one bit set for a MAF entry?
  - MAF.victimize and MAF.vct2shr

    The MAF.victimize must override the MAF.vct2shr.
  - Any other case???

12.7 MAF Address CAM

- MAF miss

  Load the probe into the MAF to store the probe while the probe is in-flight in the Scache pipe.

  – For a possible probe retry due to bank conflict, Scache tag ECC error, or Scache Data ECC error.

  – To maintain probe sequence to the same cache block.

- MAF hit

  If the probe hits a MAF entry which has a transaction in-flight in the Scache (i.e. MAF.sc_inflight is set), then the PRQ stalls until the conflicting transaction completes and clears the MAF.sc_inflight.

  Otherwise, the probe changes MAF states and probes the Scache if necessary.

Table 12–3 Forwards hit MAF (Full Address Match)

	MAF states						
Forwards (*Fwd)	**MAF.sc_inflight**	**MAF.sys_inflight**	**MAF.coherent**	**MAF.sys_cmd<2:0>**	**Response**	**Action**	**Need Scache action**
***Fwd**	1	X	X	XXX	Stall the Probe pipe (PRQ)		
	1	1	X	XXX	[1]		
	0	0	1	XXX		[2]	Yes
FetchFwd ReadFwd ReadShrFwd	0	0	0	XXX	FwdMiss	Set MAF.vct2shr [3]	No
	0	1	X	Read	FwdMiss	Set MAF.vct2shr	No
	0	1	X	ReadShr	FwdMiss		No
	0	1	X	ReadMod/I2D/S2D/S2DSTC	FwdMiss	Set MAF.vct2shr	No
ReadModFwd InvToDirtyFwd	0	0	0	XXX	FwdMiss	Set MAF.victimize [3]	No
	0	1	X	Read	FwdMiss	Set MAF.victimize [4]	No
	0	1	X	ReadShr	FwdMiss		No
	0	1	X	ReadMod/I2D/S2D/S2DSTC	FwdMiss	Set MAF.victimize[4]	No
SharedInval Shared-InvalLeaf SharedInvalMaster	0	0	0	XXX	Must not happen		
	0	1	X	XXX	InvalAck [5][6]	Set MAF.inval_seen [7]	Yes

Notes:

- We send a FwdMiss for a *Fwd to a non-coherent cache block.

- [1]: This case happens because we clear the MAF.sc_inflight late in order to give the Mbox enough time to consume the fill block before the block gets victimized or invalidated. The current design guarantees 15 cycles between the fill address and a probe to the fill block. Since we delay clearing the MAF.sc_inflight bit, we send a system request before clearing the sc_inflight bit. We believe it takes more than 8 cycles to receive a fill block from the system after sending a system request. Hence, we must not receive a fill block from the system while the MAF.sc_inflight bit is set.
- [2]: This case happens because either:
  - Cbox received a exclusive block and victimized the block. Then Cbox wanted the cache block back but hasn't sent a system request yet OR
  - Cbox received the ownership of the cache block but the cache block is not coherent.
- [3]: Cbox received the ownership of the block but the cache block is not coherent.
- [4]: Cbox does not know whether the *Fwd is for an earlier version of the block we victimized or for new block which is on its way to us. To be conservative Cbox victimizes the block.
- [5]: Cbox sends a InvalAckMaster and a InvalAckLeaf in response to a SharedInvalMaster and a SharedInvalLeaf respectively.
- [6]: We must send the InvalAck right away. If the SharedInval was before the current request and we do not send the InvalAck, then the owner of the cache block will wait for the InvalAck from us indefinitely and we will not receive the fill block.
- [7]: If a BlkShared is returned in response to the outstanding Read[Shr]Req, then we do not know whether the block is before the SharedInval or after the SharedInval; hence, block has to be invalidated. If BlkExclCnt or BlkDirty returns, then we know the block is after the SharedInval and it is safe to fill the Scache with the block.

Table 12–4 Response Hit MAF (MAF Index)

Responses	MAF States:	MAF_inval_seen	MAF.mb_retired	MAF.victimizeMAF.vic_to_shr	merge_buffer_wr	Action
BlkInval		Must not happen				
BlkIO		X	X	X	X	Send the IO block to Mbox.
BlkShared		0	X	X	X	Fill the Scache as Shared.

Table 12–4 Response Hit MAF (MAF Index) (Continued)

Responses	MAF States:	MAF_inval_seen	MAF.mb_retired	MAF.victimizeMAF.vic_to_shr	merge_buffer_wr	Action
		1	0	X	X	Fill the Scache/Mbox and set the timer. When the timer expires, invalidate the block.
		1	1	X	X	Discard the fill block and send a system request.
BlkExclCnt		X	X	0	0	Fill the Scache as ExclCln. Update the coherence counter.
		X	X	1	0	Fill the Scache as ExclCln. Update the coherence counter. Set the timer if coherent.
		X	X	0	1	Fill the Scache as Dirty if the block is coherent. If the block is not coherent, fill the Scache as exclusive. Update the coherence counter.
		X	X	1	1	Fill the Scache as Dirty. Update the coherence counter. Set the timer if coherent.
BlkDirty		X	X	0	X	Fill the Scache as Dirty.
		X	X	1	X	Fill the Scache as Dirty. Set the timer.
InvToDirtyRespCnt[1]		X	X	0	X	Fill the Scache as dirty if coherent and as exclusive if not coherent.
		X	X	1	X	Fill the Scache as dirty if coherent and as exclusive if not coherent. Set the timer and when the timer expires victimize the block.
ShrToDirtySuccessCnt ShrToDirtyProbCnt		0	X	0	X	Enter the Scache pipe to update the Scache tag. Set the coherence timer. Update the coherence counter.
		1	X	X	X	Incorrectly granted ShrToDirtyReq. Do VictimClean.
		X	X	X	0	Must not happen.
ShrToDirtyFail		X	X	X	X	Cbox does not need to send ShrToDirtyFail to Mbox but Cbox must send a Read*Req if a non-StxC request has been merged.
InvalAck[2]		X	X	X	X	Update the coherence counter.
WrIoAck/WrIONack		X	X	X	X	Notify the Mbox.
NXMResp/ERRResp		X	X	X	X	

[1] If we have a pending Ifecth, put it in the retry queue.

[2] The InvalAck makes the block coherent. If the MAF.victimize or MAF.vic_to_shr is set, enter the Scache pipe and victimize the block. If the MAF entry is for a merge buffer write, then change the Scache tag to dirty and send it to Mbox.

Notes:

- Cbox does not receive a ShrToDirtyFail in response to a ShrToDirtyReq. Instead the ShrToDirtyReq gets forwarded and Cbox must receive:
  - ShrToDirtySuccessCnt if the Scache has a shared copy.
  - ShrToDirtySuccessCnt, BlkExclCnt, or BlkDirty if the Scache does not have a shared copy.

12.8 Scache Hit

Note that the 1, 3, 6, and 13 "footnotes" to the tables in the web spec are not called out in the tables. Is this okay? They are:

- Cbox sends ShrToDirtyReq if Mbox needs the ownership of the cache block.
- We may send BlkDirty to the requester and transition to Invalid. Unlike the 21364, the current proposal is to have extra state (migratory bit) for cache blocks.
- The directory, not the Scache, sends this.
- If a StxC request from Mbox finds no shared copy in the Scache, fail the StxC and send a STCFail message to Mbox.

Tables 12–5, 12–6, and 12–7 show.....

Table 12–5 Miss Requests from Mbox

Scache State Commands			Invalid	ExclClean Merge Buffer Does Not Have Block	ExclClean Merge Buffer has Block Modified	Dirty	Shared
IFetch FetchLine PfetchLine	SC State	-->	Invalid	ExclClean		Dirty	Shared
	I/Mbox	<--		FillBlkExclClean		FillBlkDirty	FillBlkShared
	Home	<--	Read[Shr]Req				
PfetchLineMod	SC State	-->	Invalid	ExclClean		Dirty	Shared
	I/Mbox	<--		FillBlkExclClean		FillBlkDirty	FillBlkShared
	Home	<--	ReadModReq				
FetchLineMod	SC State	-->	Invalid	Dirty			Shared
	I/Mbox	<--		FillBlkDirty			FillBlkShared
	Home	<--	ReadModReq				ShrToDirtyReq
CtoD ItoD	SC State	-->	Invalid	Dirty			Shared
	I/Mbox	<--		FillBlkDirty			FillBlkShared
	Home	<--	InvToDirtyReq				ShrToDirtyReq
CtoDSTC	SC State	-->	Invalid	Dirty			Shared
	I/Mbox	<--	StxCFail	FillBlkDirty			FillBlkShared
	Home	<--					ShrToDirtySTCReq
EvictBlk	SC State	-->	Invalid[3]	Invalid			

Table 12–5 Miss Requests from Mbox (Continued)

Scache State Commands			Invalid	ExclClean Merge Buffer Does Not Have Block	ExclClean Merge Buffer has Block Modified	Dirty	Shared
	Home	<—		VictimCln	Victim		
CleanBlk	SC State	—>	Invalid[3]	Shared			
	Home	<—		VictimClnToShr	VictimToShr		
Victimize MAF.victimize*	SC State	—>	Invalid[3]	Invalid			Invalid[1]
	Home	<—		VictimCln	Victim		
VictimToShr MAF.vct2shr*	SC State	—>	Invalid[3]	Shared[2]			Shared
	Home	<—		VictimClnToShr	VictimToShr		
Invalidate MAF.inval_seen*	SC State	—>	Invalid[3]	Must not happen.			Invalid
	Home	<—		Must not happen			

1 This case happens because the MAF.victimize bit was set when a ReadModFwd or a InvToDirtyFwd hit the MAF entry which had outstanding ReadReq. If the BlkShared was returned, the cache block is after the ReadModFwd or the InvToDirtyFwd. Hence Cbox can keep the shared copy. However, it is also possible that this case happens because a Chg2Shared from Mbox hit the Exclusive block. Then we must invalidate the cache block. To be conservative, we invalidate the Shared copy.

2 If the merge buffer has modified data, the merge buffer writes both to the VDB and to the Scache:
- Cbox sends cache block to the merge buffer.
- The cache block gets merged with the merge buffer data.
- The merge buffer writes to the Scache and to the VDB.

3 The block has already been victimized by a fill.

Table 12–6 lists the forwards from the remote directory.

Table 12–6 Forwards From (Remote) Directory

Scache State Commands			Invalid	ExclClean Merge Buffer Does Not Have Block	ExclClean Merge Buffer has Block Modified	Dirty	Shared
FetchFwd	SC State	—>	Invalid	Shared[1]			Shared
	Home	<—	FwdMiss	FwdAckShr	VictimAckShr		FwdMiss
	Requester	<—		BlkInval			
ReadShrFwd ReadFwd	SC State	—>	Invalid	Shared[1]			Shared
	Home	<—	FwdMiss	FwdAckShr	VictimAckShr		FwdMiss
	Requester	<—		BlkShared			
ReadModFwd Requester is a processor	SC State	—>	Invalid				
	Home	<—	FwdMiss	FwdAckExcl			FwdMiss
	Requester	<—		BlkExclCnt(0)	BlkDirty		

Table 12–6 Forwards From (Remote) Directory (Continued)

Scache State Commands			Invalid	ExclClean Merge Buffer Does Not Have Block	ExclClean Merge Buffer has Block Modified	Dirty	Shared
ReadModFwd Requester is a I/O device	SC State	—>	Invalid				
	Home	<—	FwdMiss	VictimAckExcl			FwdMiss
	Requester	<—		BlkExclCnt(0)			
InvalToDirtyFwd Requester is processor	SC State	—>	Invalid				
	Home	<—	FwdMiss	FwdAckExcl			FwdMiss
	Requester	<—		InvalToDirtyRespCnt[2]			
InvalToDirtyFwd Requester is I/O device	SC State	—>	Invalid				
	Home	<—	FwdMiss	FwdAckExcl	VictimAckExcl		FwdMiss
	Requester	<—		InvalToDirtyRespCnt[2]			
SharedInval	SC State	—>	Invalid	Must not happen			Invalid
	Requester	<—	InvalAck	Must not happen			InvalAck

[1] If the merge buffer has modified data, the merge buffer writes both to the VDB and to the Scache:
Cbox sends cache block to the merge buffer.
The cache block gets merged with the merge buffer data.
The merge buffer writes to the Scache and to the VDB.

[2] The Scache, not the directory, sends this.

Table 12–7 lists the responses (fills) from the system.

Table 12–7 Responses (Fills) from System

Scache State Commands			Invalid	ExclClean Merge Buffer Does Not Have Block	ExclClean Merge buffer has Block Modified	Dirty	Shared
BlkInval	Must not happen (21464 processors do not send Fetch request.)						
BlkIO	Mbox <— DataIO						
BlkShared	SC State	—>	Shared	Must not happen			
	I/Mbox	<—	FillBlkShared	Must not happen			
BlkExclCnt	SC State	—>	ExclClean	Must not happen			
	I/Mbox	<—	FillBlkExclCln	Must not happen			
BlkDirty	SC State	—>	Dirty	Must not happen			
	I/Mbox	<—	DataDirty	Must not happen			
InvalToDirtyRespCnt	SC State	—>	Invalid	Must not happen			
	I/Mbox	<—	Victimize	Must not happen			
ShrToDirtySuccessCnt	SC State	—>	Invalid	Must not happen			ExclClean
	I/Mbox	<—	STCFail[1]	Must not happen			DataExclCln

Table 12–7 Responses (Fills) from System (Continued)

Scache State Commands			Invalid	ExclClean Merge Buffer Does Not Have Block	ExclClean Merge buffer has Block Modified	Dirty	Shared
	Home	<—	VictimCln, ReadModReq[2]	Must not happen			
ShrToDirtyProbCnt	SC State	—>	Invalid	Must not happen			ExclClean
	I/Mbox	<—	STCFail	Must not happen			DataExclCln
	Home	<—	ShrToDirtyRe-lease[3]	Must not happen			ShrToDirtyCom-plete
ShrToDirtyFail [14]	SC State	—>	Invalid	Must not happen			
	I/Mbox	<—		Must not happen			
	Home	<—	Read*Req[4]	Must not happen			

1 If the response was for a StxC request. If a non-StxC and a StxC request gets merged at the MAF, Cbox send a ShrToDirtyReq. So when ShrToDirty*Cnt or ShrToDirtyFail returns, Cbox has to look at the MAF.miss_stxc bit, not the request that was sent, to determine whether the response is for a StxC.

2 Cbox does VictimCln when all the InvalAcks are received and then sends ReadModReq if non-StxC request.

3 Cbox needs to send ReadModReq if a non-StxC request.

4 A ShrToDirtyFail does not need Scache action. Cbox sends a Read*Req if a non-StxC request has been merged.

Notes:

- According to the current proposal, InvalToDirtyReqs get forwarded even if they are originated from processor as well as I/O devices. Hence SharedInval must not happen if we have the Exclusive block.
- Since there can be only one outstanding request for a cache block:
  - Must not receive ShareToDirty*Cnt if the Scache has the block ExclClean or Dirty.
  - Must not receive Blk* if the Scache has a copy of the block, either Shared or Exclusive.
- FwdMiss may happen due to:
  - The cache block has been victimized.
  - The cache block is not coherent.
  - The cache block hasn't been received.

12.9 VAF Address CAM

- If there is a VAF hit, then the victim entry gets a high priority to expedite the processing (i.e The DIFT needs the Victim before servicing the request).
- No local victim and local request race for the new cache coherence protocol with no LPRs.

Table 12–8 VAF Hit

Probes	Victims to home directory		Response to remote node	
	Victim*	*Ack*	InvToDirtyRespCnt, Blk*	InvalAck
Miss	Set the high priority bit for the VAF entry			
Victimize*	Set the high priority bit for the VAF entry			
Forwards (*Fwd)	Set the high priority bit for the VAF entry.	Must not happen		
SharedInval	Must not happen	Must not happen		

- Victim*
  - Victim, VictimToShr, VictimCln, VictimClnToShr.
  - Cbox is in the process of giving up ownership.
  - Victim-Request race gets resolved at the DIFT.
- *Ack* (VictimAckShr, VictimAckExcl, FwdAckShr, FwdAckExcl):
  - VictimAckShr, VictimAckExcl, FwdAckShr, FwdAckExcl
  - Cbox is in the process of giving up the ownership in response to A *Fwd.
  - The DIFT has an entry waiting for the *Ack*. Subsequent requests to the block will not be serviced until the *Ack* is received by the DIFT, thereby maintaining the order.
- A VAF entry can generate two messages to the system.
  - A ForwardMiss and a Victim* to the home node (if *Fwd hit a Victim*).
  - An *Ack* to the home node and Blk* to the requester.
  - A Victim* and a ShrToDirtyComplete.

12.10 Directory Responses

Table 12–9 show the directory state request responses.

Table 12–9 Directory State Request Responses

Directory State Request	Local (InMemory)	RemoteExcl (O)	Shared1 (S1)	Shared2 (S1,S2)	SharedM (M)
FetchReq	Dir: –> Local Requester <– BlkInval	Dir: –> Shr1(O) Owner <– FetchFwd	Dir: –> Shr1(S1) Requester <– BlkInval	Dir: –> Shr2(S1,S2) Requester <– BlkInval	Dir: –> ShrM(M) Requester <– BlkInval
ReadShrReq	Dir: –> Shr1(R) Requester <– Blk-Shared	Dir: –> Shr2(O,R) Owner <– ReadShr-Fwd	Dir: –> Shr2(S1,R) Requester <– Blk-Shared	Dir: –> ShrM(S1,S2,R) Requester <– Blk-Shared	Dir: –> ShrM(M,R) Requester <– Blk-Shared
ReadReq	Dir: –> Shr1(R) Requester <– BlkEx-clCnt(0)	Dir: –> Shr2(O,R)[1] Owner <– ReadFwd	Dir: –> Shr2(S1,R) Requester <– Blk-Shared	Dir: –> ShrM(S1,S2,R) Requester <– Blk-Shared	Dir: –> ShrM(M,R) Requester <– Blk-Shared
ReadModReq	Dir: –> Remote-Excl(R) Requester <– BlkEx-clCnt(0)	Dir: –> Remote-Excl(R) Owner <– ReadMod-Fwd	Dir: –> Remote-Excl(R) Requester <– BlkEx-clCnt(1) S1 <– SharedInval	Dir: –> Remote-Excl(R) Requester <– BlkEx-clCnt(2) S1,S2 <– SharedInval	Dir: –> Remote-Excl(R) Requester <– BlkEx-clCnt(M) M <– SharedInvalB-cast

Table 12–9 Directory State Request Responses (Continued)

Directory State Request	Local (InMemory)	RemoteExcl (O)	Shared1 (S1)	Shared2 (S1,S2)	SharedM (M)
InvToDirtyReq	Dir: –> Remote-Excl(R) Requester <– InvTo-DirtyRespCnt(0)	Dir: –> Remote-Excl(R) Owner <– InvToDirty-Fwd	Dir: –> Remote-Excl(R) Requester <– InvTo-DirtyRespCnt(1) S1 <– SharedDirty	Dir: –> Remote-Excl(R) Requester <– InvTo-DirtyRespCnt(2) S1,S2 <– SharedDirty	Dir: –> Remote-Excl(R) Requester <– InvTo-DirtyRespCnt(M) M <– ShrDirtyBcast
ShrToDirtyReq Requester is a sharer (R = S1)	Dir: –> Remote-Excl(R) Requester <– BlkEx-clCnt(0)[2]	Dir: –> Remote-Excl(R) Owner <– ReadMod-Fwd	Dir: –> Remote-Excl(R) Requester <– S2DSuccCnt(0)	Dir: –> Remote-Excl(R) Requester <– S2DSuccCnt(1) S2 <– SharedInval	Dir: –> Remote-Excl(R) Requester <– S2DSuccCnt(M)[3] M <– ShrInvalBcast
ShrToDirtyReq Requester is not a sharer	Dir: –> Remote-Excl(R) Requester <– BlkEx-clCnt(0)[2]	Dir: –> Remote-Excl(R) Owner <– ReadMod-Fwd	Dir: –> Remote-Excl(R) Requester <– BlkEx-clCnt(1)[4] S1 <– SharedInval	Dir: –> Remote-Excl(R) Requester <– BlkEx-clCnt(2) S1,S2 <– SharedInval	Dir: –> Remote-Excl(R) Requester <– BlkEx-clCnt(M) M <– ShrInvalBcast
ShrToDirtySTCReq Requester is a sharer (R = S1)	Dir: –> Local Requester <– Shr2DirtyFail	Dir: –> Remote-Excl(O) Requester <– ShrTo-DirtyFail	Dir: –> Remote-Excl(R) Requester <– Shr2DirtySuccCnt(1)	Dir: –> Remote-Excl(R) Requester <– Shr2DirtySuccCnt(2) S2 <– SharedInval	Dir: –> ShrM(M) Requester <– ShrTo-DirtyProbCnt(M)
ShrToDirtySTCReq Requester is not a sharer	Dir: –> Local Requester <– Shr2DirtyFail	Dir: –> Remote-Excl(O) Requester <– ShrTo-DirtyFail	Dir: –> Shared1(S1) Requester <– Shr2DirtyFail	Dir: –> Shared2(S1,S2) Requester <– Shr2DirtyFail	Dir: –> ShrM(M) Requester <– Shr2DirtyFail

1 The directory state transitions to Shr2(O, R) if Victim[/Fwd]AckShared is received and transition to RemoteExcl(O) if Victim[/Fwd]AckExcl is received.

2 The 21364 fails the ShrToDirtyReq since the request is not a sharer.

3 The directory may incorrectly succeed the ShrToDirtyReq when the sharing mask is used. The request is responsible for the recovery (i.e. do VictimClean and re-send the request).

4 **No corresponding footnote text**

Notes:

- EV8 processors are not allowed to send a FetchReq. Only I/O processors may send a FetchReq.
- This table is based on the new EV8 cache coherence protocol where local requests update the directory. Hence local request and remote requesters are treated exactly the same way.
- If the requester is the exclusive owner, this indicates that the victim block from the requester is on its way to the directory. The DIFT send a response after it receives the victim.
- For *Req if the directory state is exclusive, the DIFT forward the request and speculatively write the directory. If the DIFT receives a ForwardAck*, then the directory does not have to write the directory again. If the DIFT receives a VictimAck*, then the directory has to write the whole cache block. However if the new owner does VictimClean* and the VictimClean* arrives before the ForwardAck*, the DIFT entry must do read-modify-write of the directory. To avoid this scenario, the 21364 Cbox is not allowed to respond with a ForwardAckExcl in response to a ReadFwd. The 21364 Cbox is also not allowed to send a VictimClean* for a cache block obtained through a ReadModReq. EV8 has the same number of the fill buffer entries as for the DIFT thereby eliminating speculative directory writes.
- The 21464 does not support the "Shared3" state.

- If the directory state is "Incoherent", then the directory sends "ERRResp" to the requester.
- The 21464 forwards ShrToDirtyReq if the directory is certain that the requester does not have a shared copy.

12.11 System Command Opcodes

Table 12–10 lists the system command opcodes.

Table 12–10 System Command Opcodes

Type	Command	Name	Opcode
IO	RdIOQWS	CR_OP_IO_RD_QWS	0x40
	RdIOLWS	CR_OP_IO_RD_LWS	0x41
	RdIOBytes	CR_OP_IO_RD_BYTES	0x43
	RdIOIPR	CR_OP_IO_RD_IPR	0x44
	WrIOQWS	CR_OP_IO_WR_QWS	0x50
	WrIOLWS	CR_OP_IO_WR_LWS	0x51
	WrIOBytes	CR_OP_IO_WR_BYTES	0x53
	WrIOLPR	CR_OP_IO_WR_LPR	0x54
Request	ReadReq	CR_OP_REQ_RD	0x60
	ReadShrReq	CR_OP_REQ_RD_SHR	0x61
	FetchReq	CR_OP_REQ_FETCH	0x62
	ReadModReq	CR_OP_REQ_RD_MOD	0x64
	InvToDirtyReq	CR_OP_REQ_INVAL_TO_DRTY	0x65
	ShrToDirtyReq	CR_OP_REQ_SHR_TO_DRTY	0x66
	ShrToDirtySTCReq	CR_OP_REQ_SHR_TO_DRTY_STC	0x67
Forwards	ReadFwd	CR_OP_FWD_RD	0x80
	ReadShrFwd	CR_OP_FWD_RD_SHR	0x81
	FetchFwd	CR_OP_FWD_FETCH	0x82
	ReadModFwd	CR_OP_FWD_RD_MOD	0x84
	InvToDirtyFwd	CR_OP_FWD_INVAL_TO_DRTY	0x85
	SharedInvalSingle	CR_OP_FWD_SHR_INVAL_SINGLE	0x86
	SharedInvalMask	CR_OP_FWD_SHR_INVAL_MASK	0x87
Response with Block	BlkDirty	CR_OP_RSP_BLK_DRTY	0xC0
	BlkShared	CR_OP_RSP_BLK_SHR	0xC1
	BlkInval	CR_OP_RSP_BLK_INVAL	0xC2
	BlkExclCnt	CR_OP_RSP_BLK_EXCL_CNT	0xC4
	BlkIO	CR_OP_RSP_BLK_IO	0xC5
Victim response	Victim	CR_OP_RSP_VIC	0xD8
	VictimToShared	CR_OP_RSP_VIC_TO_SHR	0xD9

Table 12–10 System Command Opcodes

Type	Command	Name	Opcode
	VictimAckExcl	CR_OP_RSP_VIC_ACK_EXCL	0xDa
	VictimAckShared	CR_OP_RSP_VIC_ACK_SHR	0xDb
Responses without Block	NXMResp	CR_OP_RSP_NXM	0xE0
	ERRResp	CR_OP_RSP_ERR	0xE1
	InvalAck	CR_OP_RSP_INVAL_ACK	0xE2
	ShrToDirtySuccessCnt	CR_OP_RSP_SHR_TO_DRTY_SUCC_CNT	0xE4
	ShrToDirtyProbCnt	CR_OP_RSP_SHR_TO_DRTY_PROB_CNT	0xE5
	ShrToDirtyFail	CR_OP_RSP_SHR_TO_DRTY_FAIL	0xE6
	InvalToDirtyRespCnt	CR_OP_RSP_INVAL_TO_DRTY_CNT	0xE7
	WrIOAck	CR_OP_RSP_WR_IO_ACK	0xE8
	WrIONack	CR_OP_RSP_WR_IO_NACK	0xE9
	InvalAckLeaf	CR_OP_RSP_INVAL_ACK_LEAF	0xEA
	InvalAckMaster	CR_OP_RSP_INVAL_ACK_MASTER	0xEB
Release response	VictimClean	CR_OP_RSP_VIC_CLN	0xF0
	VictimClnToShr	CR_OP_RSP_VIC_CLN_TO_SHR	0xF1
	ForwardAckExcl	CR_OP_RSP_FWD_ACK_EXCL	0xF2
	ForwardAckShared	CR_OP_RSP_FWD_ACK_SHR	0xF3
	ForwardMiss	CR_OP_RSP_FWD_MISS	0xF4
	SharedToDirtyComplete	CR_OP_RSP_SHR_TO_DRTY_COM	0xF5
	SharedToDirtyRelease	CR_OP_RSP_SHR_TO_DRTY_REL	0xF6
Special	NZ-NoOp	CR_OP_SPEC_NZNOP	0xA0
	SharedInvalBcast	CR_OP_RSP_SHR_INVAL_BRD	0xB1
	SharedInvalBcastLeaf	CR_OP_RSP_SHR_INVAL_BRD_LEAF	0xB2
	SharedInvalBcastMaster	CR_OP_RSP_SHR_INVAL_BRD_MASTER	0xB3

12.12 Protocol Message Descriptions

12.12.1 IO CHANNEL Message Details

12.12.1.1 RdBytes, RdLWs, RdQWs, RdIPR

This processor/I/O device wishes to do a load to IO space. The request includes an address, a MAF#, PID, and a Mask indicating which parts of the block are being read.

QWADD(5:3) contains the exact address bits of the first load in the block (i.e. the load with the lowest address). For RdQWs the mask indicates the merged quadword loads. For RdLWs 32 bytes of information is expected to be returned (double-pumped into 64-bytes) and the mask indicates the merged longword loads within the given hexaword. For RdBytes no merging is allowed and the mask indicates the valid bytes/ words - one or two bytes of properly aligned information is expected to be returned.

RdIPR is identical to RdQWs, except that the different opcode indicates that the reference was within the range of the processor I/O space rather than the ASIC I/O space (i.e. the address references an 21364 IPR).

The likely response to a Rd* command is BlkIO. Also possible is NXMResp and ERRResp.

Note that 21364 does not support executing instructions directly from I/O space, so these commands can only be generated by load instructions that reference I/O space.

These commands are used on both I/O and router channels. Note that an I/O device can source one of these commands.

12.12.1.2 WrBytes, WrLWs, WrQWs, WrIPR

The processor—I/O device did a store to I/O space. The request has an address, a PID, a mask indicating which parts to write, a write I/O identifier, and the block of data.

QWADD(5:3) contains the exact address bits of the first store in the block (i.e. the store with the lowest quad word address). For WrQWs and WrIPR the mask indicates the merged quadword stores. For WrLWs 32 useful bytes of information is sent and the mask indicates the merged longword stores in the hexaword. For WrBytes no merging is allowed and the mask indicates the valid bytes/word.

The opcode, mask and QWADD are always consistent. Mask is never zero.

The tables below show the useful data for IO writes that are initiated by the 21464 processor.

Table 12–11 shows the location of the useful data for fully-merged WrQW's and WrIPR's. In the table, *N*.L is the lower longword of quadword *N*, *N*.H is the upper longword of quadword *N*. *X* is unused data. Quadwords are merged only if they are issued in ascending address order. Noncontiguous quadwords can be merged.

In the case of a WrIPR, the two longwords indicated by a single bit in the mask and by QWADD(5,3) contain all the useful information. The first longword contains bits 0-31 and the next contains bits 32-63.

Table 12–11 Location of Useful Data for Fully-Merged WrQW's and WrIPR's

QWADD(5:3)	Mask	Ordered Block Data for WrQWs and WrIPR (in Long-words)
0	0xFF	0.L,0.H,1.L,1.H,2.L,2.H,3.L,3.H,4.L,4.H,5.L,5.H,6.L,6.H,7.L,7.H
1	0xFE	X.X,X.X,1.L,1.H,2.L,2.H,3.L,3.H,4.L,4.H,5.L,5.H,6.L,6.H,7.L,7.H
2	0xFC	X.X,X.X,X.X,X.X,2.L,2.H,3.L,3.H,4.L,4.H,5.L,5.H,6.L,6.H,7.L,7.H
3	0xF8	X.X,X.X,X.X,X.X,X.X,X.X,3.L,3.H,4.L,4.H,5.L,5.H,6.L,6.H,7.L,7.H
4	0xF0	X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,4.L,4.H,5.L,5.H,6.L,6.H,7.L,7.H
5	0xE0	X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,5.L,5.H,6.L,6.H,7.L,7.H
6	0xC0	X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,6.L,6.H,7.L,7.H
7	0x80	X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,7.L,7.H

Table 12–12 shows the location of useful data for fully-merged WrLWs. The numbers shown are the sequential longwords. Longwords are merged only if they are issued in ascending address order. Noncontiguous longwords can be merged.

Table 12–12 Location of Useful Data for Fully-Merged WrLW's

QWADD(5:3)	Mask	Ordered Block Data for WrLWs (in Long-Words)
000	0xFF	0,1,2,3,4,5,6,7,X,X,X,X,X,X,X,X
000	0xFE	X,1,2,3,4,5,6,7,X,X,X,X,X,X,X,X
001	0xFC	X,X,2,3,4,5,6,7,X,X,X,X,X,X,X,X
001	0xF8	X,X,X,3,4,5,6,7,X,X,X,X,X,X,X,X
010	0xF0	X,X,X,X,4,5,6,7,X,X,X,X,X,X,X,X
010	0xE0	X,X,X,X,X,5,6,7,X,X,X,X,X,X,X,X
011	0xC0	X,X,X,X,X,X,6,7,X,X,X,X,X,X,X,X
011	0x80	X,X,X,X,X,X,X,7,X,X,X,X,X,X,X,X
100	0xFF	X,X,X,X,X,X,X,X,0,1,2,3,4,5,6,7
100	0xFE	X,X,X,X,X,X,X,X,X,1,2,3,4,5,6,7
101	0xFC	X,X,X,X,X,X,X,X,X,X,2,3,4,5,6,7
101	0xF8	X,X,X,X,X,X,X,X,X,X,X,3,4,5,6,7
110	0xF0	X,X,X,X,X,X,X,X,X,X,X,X,4,5,6,7
110	0xE0	X,X,X,X,X,X,X,X,X,X,X,X,X,5,6,7
111	0xC0	X,X,X,X,X,X,X,X,X,X,X,X,X,X,6,7
111	0x80	X,X,X,X,X,X,X,X,X,X,X,X,X,X,X,7

Here is the complete table of the useful data in the quadword specified by QWADD(5,3) of a WrByte. (The other 7 quadwords in the packet are always garbage so they are not listed in the table.) The table byte values are listed from low-order bytes (left) to high-order bytes (right).

Table 12–13 Location of Useful Data for Quadword Specified by QWADD(5,3) of a WrByte

Mask	Quadword of Data for WrBytes (in Bytes)
0x03	0,1,X,X, X,X,X,X
0x02	X,1,X,X, X,X,X,X
0x0C	X,X,2,3, X,X,X,X
0x08	X,X,X,3, X,X,X,X
0x30	X,X,X,X, 4,5,X,X
0x20	X,X,X,X, X,5,X,X
0xC0	X,X,X,X, X,X,6,7
0x80	X,X,X,X, X,X,x,7

WrIPR is identical to WrQWs, except that the different opcode indicates that the reference was within the range of the processor I/O space rather than the ASIC I/O space (i.e. the address references a 21364 IPR). The "is for IO" bit is set for a WrIPR, though the message is really destined for the CSR master.

The only responses to a Wr*s command are WrIOAck or WrIONAck. (WrIONAck can only be returned in response to an IO write to the RBOX_INTA IPR.)

These commands are used on both I/O and router channels. Note that an I/O device may source one of these commands.

12.12.2 REQUEST CHANNEL Message Details

12.12.2.1 ReadReq

A load miss. The block may either be returned in shared or exclusive state. The request includes an address (offset and routing information), MAF #, PID, and wrap.

The likely response to a ReadReq is BlkShared or BlkExclusiveCnt. Also possible is NXMResp and ERRResp.

This command is used only on the interprocessor channels.

12.12.2.2 ReadSharedReq

Same as ReadReq except we must end up in shared state. Usually generated by an instruction fill. The request includes an address (offset and routing information), MAF#, PID, and wrap.

The likely response to a ReadSharedReq is BlkShared. Also possible is NXMResp and ERRResp.

This command is used only on the interprocessor channels.

12.12.2.3 ReadModReq

A processor store miss, ordered DMA read, or a DMA write. The block may be returned in either exclusive or dirty state. (It will normally be written into the cache in the dirty state by the processor core. If generated by a DMA read request, data returned in the exclusive state may ever be converted to the dirty state, though.) The request includes an address (offset and routing information), MAF#, PID, and wrap.

The likely response to a ReadModReq is BlkExclusiveCnt and InvalAck. NXMResp and ERRResp are also possible. BlkExclusiveCnt returned to an 21364 processor in response to a ReadModReq forces the block to be written into the cache dirty.

This command is used on both I/O and interprocessor channels.

An I/O device may source this packet. In this case, the memory must always have an up-to-date copy of the block. For instance, this means that it is not acceptable to return a dirty block copy from a processors cache directly without also updating memory with a VictimAckExcl. This restriction is so that the I/O device only has to victimize a dirty block if it actually has written to the block-minimizing bandwidth on the I/O port.

12.12.2.4 FetchReq

A no-cache load request or a (possibly unordered) DMA read. The block must be returned in invalid state. The request includes an address (offset and routing information), MAF#, PID, and wrap.

The likely response to a FetchReq is BlkInval. NXMResp and ERRResp also possible.

This command is used on both I/O and interprocessor channels. An I/O device may source this packet. When used for DMA reads, requests to multiple outstanding blocks may be (from a coherence perspective) reordered. See Section 13.6.

12.12.2.5 SharedtoDirtyReq

This processor has/had a shared copy of this block and wishes to write it. There are two possi le responses to this request: success or failure. The request includes an address (offset and routing information), MAF#, and PID.

This command should be failed if it reaches the directory and does not find the block in shared state with the corresponding sharing mask bit set or if it does not find the processor on the sharing list (when the requesting processor is remote).

The likely response to a SharedtoDirtyReq is SharedtoDirtySuccessCnt, SharedtoDirtyFail, or InvalAck. Also possible is ERRResp.

This command is used only on the interprocessor channels.

The directory may incorrectly succeed this request in which case the source processor must recover from the mistake. See Section 12.13.6.

Store-conditional instructions will generate SharedtoDirtyReq commands when the source processor is in STC-optimistic mode. When the source processor is in STC-conservative mode it will instead generate SharedtoDirtySTCReq. See Section 12.13.8.

12.12.2.6 SharedtoDirtySTCReq

This processor is in STC-conservative mode, has/had a shared copy of this block, and wishes to succeed a store-conditional. There are three possible responses to this request: success, probable success, or failure. This conservative request allows the requesting processor to avoid unnecessary invalidates. The request includes an address (offset and routing information), MAF# and PID.

The possible responses to a SharedtoDirtySTCReq are: SharedtoDirtyProbCnt, SharedtoDirtySuccessCnt, SharedtoDirtyFail, or InvalAck. Also possible is ERRResp.

See Section 12.13.8. SharedtoDirtyFail should be the directory's response when the requesting processor is not in the sharing mask or sharing list. SharedtoDirtyProbableCnt should be the directories response when the processor is in the sharing mask. SharedtoDirtySuccessCnt should be the response when the processor is in the sharing list.

This command is used only on the interprocessor channels.

12.12.2.7 InvaltoDirtyReq

A full-block write request (most likely) from the processor or DMA write. The data need not be returned. The request includes an address (offset and routing information), MAF#, and PID.

The response to a InvaltoDirtyReq is InvaltoDirtyRespCnt or InvalAck. NXMResp is also possible (only in the presence of a true software error when from the processor or a mispeculation by a DMA engine). ERRResp is also possible.

This command is used on both I/O and interprocessor channels. An I/O device may source this packet.

Note that the system guarantees that the old value of the memory location resides in memory when responding with InvaltoDirtyRespCnt to a InvaltoDirtyReq generated by a DMA device. This allows the DMA device to "speculatively" launch an InvaltoDirty-Req. See Section 13.6.

12.12.3 FORWARD CHANNEL Message Details

12.12.3.1 ReadForward, ReadSharedForward, ReadModForward, FetchForward, InvaltoDirtyForward

The corresponding requests reached the directory and found the block to be in exclusive state at different processor. A DIFT (Directory in-flight table) entry has been created, which will typically be cleared when the forward request is acked. The command includes an address (including both an offset and a PID of the directory), a MAF#, PID, wrap (except for InvaltoDirtyForward), and routing information to reach the forwarded destination. The MAF# and PID indicate the original source of the request.

There are two categories of responses (to the DIFT) to a *Forward command. The most likely category of responses are: VictimAckExcl, VictimAckShared, ForwardAckExcl, or ForwardAckShared. In these cases a Blk* response is also sent to the requestor.

The less likely response (to the DIFT) to a *Forward command is ForwardMiss. A ForwardMiss is accompanied by a Victim, VictimClean, VictimtoShared, or VictimCleantoShared.

If the block had been thought to be exclusive at the same processor as generated the request, the request must block rather than generate a forward. Otherwise, the block might continually be reloaded and evicted - livelock. (It doesn't make any sense to forward to yourself anyway.)

If a block is exclusively held by a DMA device, forwards must always be generated, even if the exlusive owner is the requestor. A DMA device has the uppermost PID bit set. In response to the forward, a DMA device will send a ForwardMiss and eventually evict the block.

These commands are used on both the I/O and interprocessor channels.

12.12.3.2 SharedInvalSingle

The directory sends these when a processor wishes to gain exclusive access to a block that is in shared state. SharedInvalSingle is generated when a block is in the Shared1 or Shared2 states, or if each mask bit refers to a single processor and the block is in SharedM state. The SharedInvalSingle command contains an address (including both an offset and a PID of the directory), MAF#, PID, and routing information to reach the forwarded destination.

Note that the cache invalidate associated with a SharedInvalSingle should never be performed at the requesting PID, even though it may be included in the sharing list. On 21364, the DIFT never launches a SharedInvalSingle to the requesting processor (and the coherence count never includes the requestor).

This command is used only on the interprocessor channels.

12.12.3.3 SharedInvalBroadcast

The directory sends these when a processor wishes to gain exclusive access to a block that is in SharedM state and each mask bit refers to more than one processor. The router is required to fanout the inval within the cluster of processors that share a mask bit, fan back in the completion, and send a single InvalAck back to the requesting processor. The SharedInvalBroadcast command contains an address (including both an offset and a PID of the directory), MAF#, PID, and routing information to reach the forwarded cluster.

Note that it is required that during the fanout at the cluster that includes the requesting PID, the inval should never be executed on the requesting PID, though it must be executed on all other processors in the cluster that share the same sharing mask bit.

See Section 12.12.3.3 for more detail on the fanin/fanout operation.

This command is used only on the interprocessor channels.

12.12.4 RESPONSE CHANNEL Message Details

12.12.4.1 BlkShared

Data returned in response to a ReadReq or ReadSharedReq command. The data should be deposited into the cache in the shared state. The command header contains a MAF #, wrap, and routing information to reach the destination. The data in the packet must always be wrapped in the octa-word order specified by the corresponding request. The wrap bits must always match this wrap order.

This command can be returned in response to a ReadReq or ReadSharedReq request.

This command is used only on the interprocessor channels.

12.12.4.2 BlkExclusiveCnt

Data returned in response to a ReadReq or ReadMod command. The data should be put into the cache in exclusive-clean state in response to a ReadReq, the dirty state in response to a Readmod generated by a store.The command header contains a MAF #, wrap, a coherence count, and routing information to reach the destination. The data in the packet must always be wrapped in the octa-word order specified by the corresponding request. The wrap bits must always match this wrap order.

A BlkExclusiveCnt with non-zero count can be generated when a ReadMod finds a block in the shared state. The count is the number of InvalAck's to expect in response at the requestor. (Except when solving the for the "Local CBOX Too Far Ahead" problem described below.)

BlkExclusive can be returned in response to a ReadReq or ReadModReq request. A non-zero count can only occur in response to a ReadMod (except when solving the for the "Local CBOX Too Far Ahead" problem described below).

This command is used on both I/O and interprocessor channels.

12.12.4.3 BlkInval

Data returned in response to a FetchReq command. The data should not be cached. The command header contains a MAF #, wrap, and routing information to reach the destination. The data in the packet must always be wrapped in the octa-word order specified by the corresponding request. The wrap bits must always match this wrap order.

This command may be returned in response to a FetchReq request.

This command is used on both I/O and interprocessor channels.

12.12.4.4 BlkIO

Data is returned in response to a read I/O command. The command header contains a MAF #, wrap, and routing information to reach the destination. The data is laned and data bytes in the packet are positioned according to their address.

The wrap bits field must be zero for BlkIO commands.

The coherency count field must be zero for BlkIO commands.

Table 12–14 shows the location of the useful data in the 16 longwords contained in a BlkIO in response to a fully-merged RdQW or RdIPR. *N*.L is the lower longword of quadword *N*, *N*.H is the upper longword of quadword *N*. *X* is unused data.

In the case of a RdIPR, the two longwords indicated by a single bit in the mask and by QWADD(5,3) contain all the useful information. The first longword contains bits 0-31 and the next contains bits 32-63.

Table 12–14 Location of Useful Data in a BlkIO in Response to a Fully-Merged RdQW or RdIPR

QWADD(5:3)	Mask	Ordered Block Data for RdQWs or RdIPR (in Long-Words)
0	0xFF	0.L,0.H,1.L,1.H,2.L,2.H,3.L,3.H,4.L,4.H,5.L,5.H,6.L,6.H,7.L,7.H
1	0xFE	X.X,X.X,1.L,1.H,2.L,2.H,3.L,3.H,4.L,4.H,5.L,5.H,6.L,6.H,7.L,7.H
2	0xFC	X.X,X.X,X.X,X.X,2.L,2.H,3.L,3.H,4.L,4.H,5.L,5.H,6.L,6.H,7.L,7.H
3	0xF8	X.X,X.X,X.X,X.X,X.X,X.X,3.L,3.H,4.L,4.H,5.L,5.H,6.L,6.H,7.L,7.H
4	0xF0	X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,4.L,4.H,5.L,5.H,6.L,6.H,7.L,7.H
5	0xE0	X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,5.L,5.H,6.L,6.H,7.L,7.H
6	0xC0	X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,6.L,6.H,7.L,7.H
7	0x80	X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,X.X,7.L,7.H

Table 12–15 shows the location of the useful data in response to fully-merged RdLWs. The numbers shown are the (up to 8) merged longwords. *X* is unused data.

Table 12–15 Location of Useful Data in Response to Fully-Merged RdLW's

QWADD(5:3)	MAsk	Ordered Block Data for RdLWs (in Long-Words)
000	0xFF	0,1,2,3,4,5,6,7,X,X,X,X,X,X,X,X
000	0xFE	X,1,2,3,4,5,6,7,X,X,X,X,X,X,X,X
001	0xFC	X,X,2,3,4,5,6,7,X,X,X,X,X,X,X,X
001	0xF8	X,X,X,3,4,5,6,7,X,X,X,X,X,X,X,X
010	0xF0	X,X,X,X,4,5,6,7,X,X,X,X,X,X,X,X
010	0xE0	X,X,X,X,X,5,6,7,X,X,X,X,X,X,X,X
011	0xC0	X,X,X,X,X,X,6,7,X,X,X,X,X,X,X,X

Table 12–15 Location of Useful Data in Response to Fully-Merged RdLW's

QWADD(5:3)	MAsk	Ordered Block Data for RdLWs (in Long-Words)
011	0x80	X,X,X,X,X,X,X,7,X,X,X,X,X,X,X,X
100	0xFF	X,X,X,X,X,X,X,X,0,1,2,3,4,5,6,7
100	0xFE	X,X,X,X,X,X,X,X,X,1,2,3,4,5,6,7
101	0xFC	X,X,X,X,X,X,X,X,X,X,2,3,4,5,6,7
101	0xF8	X,X,X,X,X,X,X,X,X,X,X,3,4,5,6,7
110	0xF0	X,X,X,X,X,X,X,X,X,X,X,X,4,5,6,7
110	0xE0	X,X,X,X,X,X,X,X,X,X,X,X,X,5,6,7
111	0xC0	X,X,X,X,X,X,X,X,X,X,X,X,X,X,6,7
111	0x80	X,X,X,X,X,X,X,X,X,X,X,X,X,X,X,7

Table 12–16 lists all of the useful data in the quadword specified by QWADD(5,3) of a BlkIO packet that is a response to RdBytes. (The other 7 quadwords in the packet are unused so they are not listed in the table.)

The table byte values are listed from low-order bytes (left) to high-order bytes (right). *X* is unused data.

Table 12–16 Location of Useful Data in Quadword Specified by QWADD(5,3) of a BlkIO Packet

Mask	Quadword of Data for WrBytes (in Bytes)
0x03	0,1,X,X, X,X,X,X
0x02	X,1,X,X, X,X,X,X
0x0C	X,X,2,3, X,X,X,X
0x08	X,X,X,3, X,X,X,X
0x30	X,X,X,X, 4,5,X,X
0x20	X,X,X,X, X,5,X,X
0xC0	X,X,X,X, X,X,6,7
0x80	X,X,X,X, X,X,x,7

This command may be returned in response to a RdBytes, RdLWs, RdQWs, or RdIPR request. This command is used on both I/O and interprocessor channels. An I/O device may source this packet.

12.12.4.5 Victim

Data written back to memory because it was dirtied. The block must have been in the exclusive state in the directory (when non-local). The command header contains an address.

Victim writes a block into memory in the state invalid as well as CAM'ing the DIFT and updating DIFT state.

The Victim command is used on both I/O and interprocessor channels. An I/O device may source this packet.

12.12.4.6 VictimtoShared

Write back data to memory and change the state of the block to shared. The block must have been in the exclusive state in the directory. The command header contains an address and a sharing PID.

VictimtoShared is similar to Victim except the final state of the block is shared in the directory.

VictimtoShared is similar to VictimAckShared except that a ForwardMiss may (or may not) also be in flight to the DIFT. VictimAckShared is sure to find a DIFT entry waiting when it returns to the directory, whereas VictimtoShared is not.

This command is used only on the interprocessor channels.

12.12.4.7 VictimAckExcl

Data written back to memory as a result of a ReadForward or an InvaltoDirtyForward that orignated from a DMA reference. The command header contains an address.

This command is similar to the victim command except in how it affects the DIFT and the final state of the block in the directory. VictimAckExcl implies that a forwarded command found a dirty block at the exclusive owner. The final state of the block should be exclusive owned by the requesting PID. It is assumed that the requesting PID is stored in the DIFT at the time the DIFT entry was created, and can be extracted in response to the VictimAckExcl address cam.

In a special race case, the write of the block to memory and/or the directory update must not be performed. See Section 12.13.3.

VictimAckExcl may be generated by an InvaltoDirtyForward or ReadModForward that originated from a DMA reference - so that memory always has an up-to-date copy of the block. See Section 13.6.

This command can only be generated in response to a ReadForward when the migratory data optimization is implemented. See section "Migratory Data Optimization" TBS.

This command is used only on the interprocessor channels.

12.12.4.8 VictimAckShared

Data written back to memory as a result of a ReadForward or ReadSharedForward. The command header contains an address and the sharing PID of the (prior) exclusive owner of the block.

This command is similar to the victim command except in how it affects the DIFT And the final state of the block in the directory. VictimAckShared implies the final state of the block should be shared, with both the prior exclusive owner and the requestor (if the requestor is non-local) on the sharing list. It is assumed that the PID of the requestor was stored at the time the DIFT entry was created and can be extracted from the DIFT to update the sharing list.

This command is used only on the interprocessor channels.

12.12.4.9 InvaltoDirtyRespCnt

Response to a InvaltoDirtyReq command. The command contains the requestor MAF #, a coherence count, and routing information to reach the destination.

This command is a possible response to a InvaltoDirtyReq request.

Note that InvaltoDirtyRespCnt responses to a DMA device must ensure that memory has an up-to-date copy of the block. See Section 13.6.

These commands are used on both the I/O and interprocessor channels.

12.12.4.10 SharedtoDirtySuccessCnt

Success response to a SharedtoDirtyReq or a SharedtoDirtySTCReq. The command contains the requestor MAF #, coherence count, and routing information to reach the destination.

This command is a possible response to a SharedtoDirtyReq or a SharedtoDirtySTCReq. SharedtoDirtySuccessCnt is generated in response to a SharedtoDirtyReq whenever the block is in shared state and the source processor is on the sharing list or the sharing mask bit corresponding the source processor is set. SharedtoDirtySuccessCnt is generated in response to a SharedtoDirtySTCReq only when the block is in Shared1 or Shared2 state (or in SharedM state when there is only one processor per mask bit) and the source processor is on the sharing list.

The directory may incorrectly succeed a SharedtoDirtyReq, but may not incorrectly succeed a SharedtoDirtySTCReq. See Sections 12.13.6 and and 12.13.8.

This command is used only on the interprocessor channels.

12.12.4.11 SharedtoDirtyProbCnt

Probable success response to a SharedtoDirtySTCReq (the source processor must be the final arbiter of success). The command contains the requestor MAF #, a coherence count, and routing information to reach the destination.

This is a response to a SharedtoDirtySTCReq when the block is in SharedM state and the mask bit corresponding to the source processor is set. See Sections 12.13.6 and 12.13.8. Note that SharedInval's are not sent out together with SharedtoDirtyProbCnt; Rather, the SharedInval's are sent out only with SharedtoDirtyComplete receipt.

This command is used only on the interprocessor channels.

12.12.4.12 SharedtoDirtyFail

Failure response to a SharedtoDirtyReq or SharedtoDirtySTCReq. The command contains the requestor MAF # and routing information to reach the destination.

This command is a possible response to a SharedtoDirtyReq or SharedtoDirtySTCReq requests. A SharedtoDirty must fail when the block is not in shared state or if the source processor is not on the sharing list or the sharing mask bit corresponding to the source processor is not set.

This command is used only on the interprocessor channels.

12.12.4.13 NXMResp

Indicates the request referenced an area of memory that does not exist. The response contains the requestor MAF # and routing information to reach the destination.

This command is a possible response to a ReadReq, ReadSharedReq, ReadModReq, FetchReq, InvaltoDirtyReq, or Rd* (IO read) request. Of this list, on 21364 the InvaltoDirtyReq (from a processor, not a DMA engine) is one one that cannot be generated

speculatively, and therefore indicates a software error. Software may use NXM responses to RdBytes requests to detect the presence of I/O devices. For all the other requests a NXMResp is not known to be an error.

This command is used on both the I/O and interprocessor channels. An I/O device may source this packet.

12.12.4.14 ERRResp

Indicates the request referenced an area of memory that encountered a hardware error. The response contains the requestor MAF # and routing information to reach the destination.

This command is a possible response to a ReadReq, ReadSharedReq, ReadModReq, FetchReq,InvaltoDirtyReq, SharedtoDirty*Req, or RdBytes request.

This command is used on both the I/O and interprocessor channels. An I/O device may source this packet.

12.12.4.15 InvalAck

A command sent to the requesting processor in response to a SharedInvalSingle or SharedInvalBroadcast command at a (possible) sharer. The command contains a MAF # and routing information to reach the destination.

When a processor wishes to write a (non-local) shared block (a ReadMod, Sharedto-Dirty*, InvaltoDirty, or SharedtoDirtyComplete), two things normally happen: (1) a *Cnt response is returned to the requestor, and (2) a SharedInval* (forward) is sent to all sharers. The coherence count is the number of InvalAcks to expect, and also the number of SharedInval* messages sent.

Note that the router broadcasts invalidates and sends one InvalAck in response to a SharedInvalBroadcast message.

This command is a possible response to a ReadModReq, SharedtoDirtyReq, Sharedto-DirtySTReq, or InvaltoDirtyReq request. Note that SharedtoDirtySTCReq invals are sometimes delayed until receipt of a SharedtoDirtyComplete to avoid unnecessary invalidates. See Sections 12.13.6 and 12.13.8.

This command is used on both the I/O and interprocessor channels. An I/O device may source this packet.

12.12.4.16 WrIOAck

A command sent back to the source of an I/O write in response to each I/O write command. The command contains a requesting WRIO #. This response indicates that the write is MB complete. The source processor is expected to track outstanding I/O write requests.

This command is the response to a Wrbytes, WrLWs, WrQWs, or WrIPR request.

This command is used on both the I/O and interprocessor channels. An I/O device may source this packet.

12.12.4.17 WrIONAck

This command is identical to the WrIOAck command except for the following:

1. It can only be returned in response to a write of the RBOX_INTA register,

2. In this special case the WrIONAck indicates that the interrupt request was not accepted.

See Section 13.5 for more information on interrupts.

This command is the response to a WrIPR request.

This command is used on both the I/O and interprocessor channels.

12.12.4.18 VictimClean

A command sent to the directory to release exclusive access to a clean block (much like a victim with no data). The command contains the address to release.

This command updates the directory state to invalid. It also cams the DIFT and may update DIFT state.

This command is used on both the I/O and interprocessor channels. An I/O device may source this packet.

12.12.4.19 VictimCleantoShared

A command sent to the directory to release exclusive access to a clean block (much like a victim with no data) but leave the block in shared state. The command contains the address to release and the sharing PID of the (prior) exclusive owner of the block.

VictimCleantoShared is similar to ForwardAckShared except that a ForwardMiss may (or may not) also be in flight to the DIFT. ForwardAckShared is sure to find a DIFT entry waiting when it returns to the directory, whereas VictimCleantoShared is not.

VictimCleantoShared is similar to VictimClean except the final block state should be shared rather than invalid.

VictimCleantoShared is similar to VictimtoShared except the data need not be written back.

This command is used only on the interprocessor channels.

12.12.4.20 ForwardAckExcl

Directory/DIFT update as a result of a ReadModForward or InvaltoDirtyForward. The command contains an address.

This command is similar to the VictimAckExcl command except that the block need not be written back. The final state of the block at the directory should be exclusive owned by the requesting PID. It is assumed that the requesting PID was stored in the DIFT at the time the DIFT entry was created and can be extracted in response to the ForwardAckExcl address cam. This command is used only on the interprocessor channels.

12.12.4.21 ForwardAckShared

Directory/DIFT update as a result of a ReadForward or ReadSharedForward. The command contains an address and the sharing PID of the (prior) exclusive owner of the block.

This command is similar to the VictimAckShared command except that the block is not written back. ForwardAckShared implies the final state of the block should be shared, with both the prior exclusive owner and the requestor (if the requestor is non-local) on the sharing list. It is assumed that the requesting PID was stored at the time the DIFT entry was created and can be extracted from the DIFT to update the sharing list.

This command is used only on the interprocessor channels.

12.12.4.22 ForwardMiss

This command indicates that a forwarded request did not find the block at the exclusive owner of the block. The command contains the address.

This command is generated in the unlikely event that a forwarded request does not find the block at the exclusive owner of the block (this can only happen when the victim was/is in flight from the exclusive owner of the block to the directory). The DIFT can determine the occurence of this unlikely event when it sees a Victim, VictimClean, or ForwardMiss while a forward request is pending to the block. When this unlikely case does happen, the DIFT regenerates the request (after waiting for both the Victim* and ForwardMiss responses to return) as if the block were originally found invalid in the directory. See Sections 12.13.1 and 12.13.2.

This command is used on both I/O and the interprocessor channels. An I/O device may source this packet.

12.12.4.23 SharedtoDirtyComplete

This command is a response from the source processor to the DIFT indicating that the source processor agreed that the SharedtoDirty should succeed. The command contains the address.

This command results when a SharedtoDirtyProbCnt was successful. The result of receipt of a SharedtoDirtyComplete is that SharedInvalBroadcast's are launchedby the DIFT.

See Sections 12.13.6 and 12.13.8 for the usage of this command.

This command is used only on the interprocessor channels.

12.12.4.24 SharedtoDirtyRelease

This command is a response from the source processor to the DIFT indicating that the source processor disagreed that the SharedtoDirty should succeed. The command contains the address.

This command results when a SharedtoDirtyProbCnt was unsuccessful.

See Sections 12.13.6 and 12.13.8 for the usage of this command.

This command is used only on the interprocessor channels.

12.12.5 SPECIAL CHANNEL Message Details

12.12.5.1 NZNOP

This command is used to fill idle slots on the interconnect.

The ALERT wires are used for system broadcast interrupt. These signals fan out to all 5 ports (north, south, east, west, I/O). The alert wires also contain the SYNCH functionality. The SYNCH signals need only be used on the compass points (north, south, east, and west).

Three of the four ALERT wires are currently allocated:

Table 12–17 ALERT Wire Allocation

ALERT Wire	Type of Connection
ALERT[0]	Hardware ALERT
ALERT[1]	Software ALERT
ALERT[2]	Hardware SYNCH
ALERT[3]	Unused

An ALERT is a (high-priority) broadcast interrupt. When an 21364 receives an ALERT and does not already have its ALERT bit set, then it sets its ALERT bit and sends out an ALERT pulse on all five output ports. An 21364 can receive an ALERT in the following ways:

- It can receive an ALERT assertion on one of the five input ports
- It can suffer a hardware error that is expected to produce an ALERT (see Rbox Port Config IPR). (This is only true for a hardware ALERT.)
- It can receive a IPR write indicating that it should launch an ALERT.

The current ALERT status is indicated in the Rbox_INT register. Software clears the (local) ALERT status (and allows the receipt of another ALERT) by a write to this register.

A software ALERT is initiated by a write to the Rbox_IREQ register. This sets the local SW ALERT bit and causes the ALERT to propagate throughout the network.

A SYNCH forces the SYNCH counter(s) to remain in synchronization with the other counters in the other 21364's in the system. (See the Rbox Config register for a description of the function of the SYNCH counter(s).)

The following things may cause the SYNCH counter(s) to enter the SYNCH interval. Note that the period counter is re-initialized at each entry into the SYNCH interval in order to keep the counters in sync. In both of these cases the local 21364 sends out a SYNCH pulse on all four compass pointsif it is not already in the SYNCH interval:

- The local period has been covered (i.e. the period counter has stepped through the required number of cycles since the last SYNCH)
- A synch is received on one of the four input ports

This command is used both on the I/O and interprocessor channels. An I/O device may source this packet.

12.12.5.2 SpecialInvalBroadcast

This command is used to complete a SharedInvalBroadcast. The message is broadcast among the processors that share the same sharing mask bit. The command contains the full address (DPID and offset) and the requesting processor's PID.

SpecialInvalBroadcast has many special properties that are different from other messages. See Section 12.12.3.3 for more details.

This command is used only on the interprocessor channels.

12.13 Protocol Race Descriptions

12.13.1 Early Forward Race

A forward arrives while there is an oustanding ReadReq or ReadModReq (for example, an exclusive block has not been returned) or if not all InvalAck's have arrived. This may be caused by either:

(A) the block is returning from the directory (or another processor) and the forward is for a subsequent request, but the forward beat the block

or

(B) The forward was for a prior copy of the block (the victim is on its way to the directory)

1. Forward sets "force shared" ("force vic" if ReadModForward) bit in the MAF, sends back a ForwardMiss to DIFT.
2. When the block returns and coherence count is zero, the MAF entry is not deleted until it forces a Victim, VictimClean, VictimtoShared, or VictimCleantoShared, whichever is appropriate
3. DIFT waits for both the Victim (or VictimClean, VictimtoShared, orVictimCleantoShared) and the ForwardMiss before proceeding
4. Then, the DIFT entry returns data to the requestor and updates directory

This solution conservatively victimizes the block even though in some cases (for example, case (B) above) it would not have been necessary.

The final directory state should be set as if the state were invalid before the last request when Victim or VictimClean are produced. The final directory state should be set as if the state were shared with the prior exclusive owner on the sharing list when VictimtoShared or VictimCleantoShared are produced. If a Victim or VictimtoShared was created, the request must get the data from the Victim or VictimtoShared command, and the memory copy must be updated to this value also. Otherwise, the memory copy of the data was/is correct.

12.13.2 Late Forward Race

A forward arrives to a block that has been recently victimized. This is simular to case (B) of the above "Early Forward" race, except it is simpler in that there is no outstanding request to the block.

1. The forward sees the block is not in the cache and a ForwardMiss is returned to the directory
2. DIFT waits for both the Victim (or VictimClean) and the ForwardMiss before proceeding
3. Then, the DIFT entry returns data to the requestor and updates directory

With a Victim, the final directory state is set as if the state was invalid before the new request. With a VictimClean, however, the final directory state is shared (due to speculative directory writes). If a Victim was created, the request must get the data from the Victim command and the memory copy must also be updated to this value. Otherwise, the memory copy of the data was correct.

12.13.3 Dual Victim Race

A Victim (or VictimClean) arrives from the requesting node before the forward acknowledgement (VictimAckExcl or ForwardAckExcl) arrives from the node the forward went to.

1. Victim sets "block written" and "set invalid" bits (VictimClean sets only "set invalid") as it cams the DIFT
2. VictimAckExcl does not write to memory when the "block written" bit is set
3. VictimAckExcl and ForwardAckExcl set the directory state to invalid whenever the "set invalid" bit is set

The final directory state must be invalid. If it was a Victim command, the final memory copy of the data must be the data from the Victim command. If it was a VictimClean command and the ack was VictimAckExcl, then the final memory copy must be the data from the VictimAckExcl.

12.13.4 Early InvalAck Race

An InvalAck arrives at the requestor before the *Count response arrives.

- CMAF must keep count of early InvalAck's (i.e. the count must be able to go negative) before the count from the *Count command is added in

12.13.5 Early InvalShared Race

An InvalShared arrives while a ReadReq or ReadSharedReq is still outstanding (for example, the block hasn't yet been returned). This may be caused by either:

(1) the block is returning from the directory and the InvalShared is for a subsequent request, but the InvalShared beat the block

or

(2) The InvalShared was for a prior copy of the block (either in this processor or in another processor that shares the same sharing mask bit at the directory)

a. InvalShared sets "force inval" bit in CMAF (and fails set dirties) and sends InvalAck

b. If the block returns shared, put the data in the cache (shared) but do not delete the MAF entry until it has invalidated the block

This solution conservatively invalidates the block even though in some cases (for example, case (2) above) it would not be necessary.

The final state of the block at this processor is invalid.

12.13.6 Wrong SharedtoDirtySuccess Race

A SharedtoDirtySuccess arrives from the directory even though the cache copy of the block has been invalidated (and probably the "force inval" bit is set?). The directory incorrectly responded success in this case. This can happen when the block was written (thus causing an InvalShared) while the set dirty was outstanding, and then another processor that shares a sharing mask bit at the directory received a shared copy of the block, and only then did the directory see the set dirty, which it responds success to since the corresponding sharing mask bit is set for the source of the set dirty.

- Fail the set dirty in this case (and make sure the cache is invalid)
- But do not release the MAF until a VictimClean is sent to the directory

Once the VictimClean reaches the directory the final state of the block is invalid. In response to the set dirty failure, the processor will generate a new ReadModReq MAF (unless it was due to a store conditional).

12.13.7 A Note on SharedtoDirties and their Resolution

The ultimate arbiter of StoD success is always the requesting MAF entry (and the cache at that processor). If the cached copy of the block has been deleted between the time that a request is launched and resolved, the StoD fails. If the cached copy has not been invalidated, the StoD succeeds.

For local StoD's (i.e. StoD's from the local processor), success or failure of the StoD is determined at the time that the request enters the DIFT and CAM's across the local MAF. If no prior reference has invalidated the block the StoD succeeds.

For StoD's from a remote processor, success or failure is determined at the time of the response from the directory is received at the requesting processor. In some cases the directory can prove that a StoD has succeeded or failed. The request will certainly fail when the block is not in shared state, the processor is not on the sharing list, or the corresponding sharing mask bit is not set. The request will certainly succeed if the block is in Shared1 or Shared2 state and the requesting processor is in the sharing list. But when the block is in SharedM state and the corresponding sharing mask bit is set, the directory cannot be certain whether the request should succeed or fail since it may be that another processor that shares the same mask bit has the up-to-date copy of the block, and the requesting processor's copy has been invalidated. See Section 12.13.6 for more details.

12.13.8 Special Store-Conditional Support

Even though the directory cannot determine the success or failure of a StoD for certain in some cases, it normally optimistically assumes that the StoD will succeed and acts accordingly. If the directory turns out to be incorrect in this assumption, the source processor corrects the error. See Section 12.13.6.

General use of this optimistic behavior could lead to a store-conditional livelock problem. The problem is that in optimistically assuming the StoD will succeed, the directory ends up performing unneeded invalidates in the cases when it was wrong. It may be possible to get into the situation where no store-conditionals could ever succeed if there are too many unneeded invalidates.

We resolve this problem by supporting an optimistic and conservative mode for store-conditional SharedtoDirty's. When in optimistic mode, a store-conditional generates a normal (i.e. optimistic) SharedtoDirty. When in conservative mode, a store-conditional generates a SharedtoDirtySTC (i.e. a conservative StoD). When a processor detects that a StoD was incorrectly succeeded by the directory, it enters conservative mode. This conservative mode is controlled by a counter. The counter is set to seven whenever a STOD that was generated from a store-conditional incorrectly succeeds. The counter is decremented on every successful SharedtoDirtySTC. Conservative mode is when the counter is not zero. A set CBOX_CTL[FORCE_STXC_CONS] always makes SharedtoDirtySTC's conservative.

The directory never optimistically succeeds a StoDSTC, it only responds success when it can prove that the StoDSTC will succeed. In cases where the directory cannot determine success/fail exactly, a SharedtoDirtyProbCnt is returned to the requesting processor, the invals are not performed, and a DIFT entry is retained to wait for the success/failure determination from the source. After the source processor receives the SharedtoDirtyProbCnt, it responds to the DIFT with a SharedtoDirtyComplete or SharedtoDirtyRelease. Upon receipt of the SharedtoDirtyComplete, the directory/DIFT sends out the invals and updates the directory state. Upon receipt of a SharedtoDirtyRelease, the DIFT entry is deallocated.

12.13.9 Local CBOX Too Far Ahead

Since the directory state is generally not updated when a block is loaded into the local cache, all remote requests arriving at a destination 21364 must probe the local L2 cache tags as well as the directory state to determine where the response should come from (L2 or memory) and what the final state of the block is. As the remote request arrives it is queued in the DIFT and the CBOX probe queue. Eventually, the CBOX responds to the DIFT after checking the L2 tags. The DIFT does not perform any coherence actions other than reading the current directory state until it receives the CBOX response. Generally, the CBOX runs ahead of the DIFT since the L2 tags are faster than memory and the DIFT must wait for the CBOX response.

Consider the following scenario. In the beginning a block is owned by a remote processor. A local ReadMod is forwarded to the remote processor by the (local) DIFT. The remote processor responds with a BlkExclusiveCnt and (A) a ForwardAckExcl. The (A) ForwardAckExcl gets stalled in the network but the BlkExclusiveCnt arrives at the destination. The local CBOX evicts the block, so there is a (B) Victim being transferred to the DIFT. Meanwhile another remote request arrives, which the CBOX responds to the DIFT with (C) ForwardMiss. At this point there are three things in-flight to the DIFT: (A) The ForwardAckExcl from the original forward, (B) The Victim from the local CBOX, and (C) the ForwardMiss from the local CBOX. It is easy for the DIFT to get confused and think that the Victim applies to the original forward, then leaving the DIFT waiting for the Victim pair to the (C) ForwardMiss.

The solution to this race is to prevent the local cbox from sending the (C) Victim until after the DIFT has received the (A) ForwardAckExcl. The remote CBOX sets the coherence count for the BlkExclusiveCnt to 1. This causes the local CBOX to wait. Once the DIFT receives the ForardAckExcl it responds to the local CBOX (internally) with an "InvalAck" that allows the Cbox to proceed with this block.

Specifically, the remote CBOX sets the coherence count to one to solve this instance whenever (1) it sends a response to a requestor resulting from a forward (2) with a matching DPID and RPID (and the request did not originate from the I/O ASIC at the node) (3) where the response gives the requestor exclusive access to the block. The DIFT sends an "InvalAck" to the CBOX after it receives a ForwardAckExclusive or VictimAckExclusive when the request source is local.

13
Router Interface — the Rbox

Introductory information about the Rbox is located in Section 2.7.2. Information about the Rbox IPRs is located in Section 16.5.

13.1 Protocol Messages

The following direction symbols are used in Tables 13–1 through 13–5:

Symbol	Meaning
T	Sent to IO7 ASIC
F	Received from IO7 ASIC – can be sent to a 21464 processor only
G	Received from IO7 ASIC – can be sent to an IO7 ASIC only
H	Received from IO7 ASIC – can be sent to a 21464 processor or an IO7 ASIC

13.1.1 Messages on the IO_CHANNEL

Table 13–1 lists messages on the IO_CHANNEL.

Table 13–1 Messages on the IO_CHANNEL

Command	Direction	Contents
RdBytes Packet Format (TBD Ticks)		
RdBytes	T,G	Route(16), Dealloc(9), Opcode(8), Block Address(32), req MAF(6), req PID(11), QW add(3), Mask(8)
RdLWs	T,G	Route(16), Dealloc(9), Opcode(8), Block Address(32), req MAF(6), req PID(11), QW add(1), Mask(8)
RdQWs	T,G	Route(16), Dealloc(9), Opcode(8), Block Address(32), req MAF(6), req PID(11), QW add(0), Mask(8)
RdIPR	F	Route(16), Dealloc(9), Opcode(8), Block Address(32), req MAF(6), req PID(11), QW add(0), Mask(8)
WrBytes Packet Format (19 Ticks)		
WrBytes	T,G	Route(16), Dealloc(9), Opcode(8), Block Address(32), reqWRIO(6), reqPID(11), Mask(8), QWadd(3), Data
WrLWs	T,G	Route(16), Dealloc(9), Opcode(8), Block Address(32), reqWRIO(6), reqPID(11), Mask(8), QWadd(1), Data
WrQWs	T,G	Route(16), Dealloc(9), Opcode(8), Block Address(32), reqWRIO(6), reqPID(11), Mask(8), QWadd(0), Data
WrIPR	F	Route(16), Dealloc(9), Opcode(8), Block Address(32), reqWRIO(6), reqPID(11), Mask(8), QWadd(0), Data

13.1.2 Messages on the REQUEST_CHANNEL

Table 13–2 lists messages on the REQUEST_CHANNEL.

Table 13–2 Messages on the REQUEST_CHANNEL

Command	Direction	Contents
Request Packet Format (3 Ticks)		
ReadReq	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), req MAF(6), req PID(11), Wrap(2)
ReadSharedReq	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), req MAF(6), req PID(11), Wrap(2)
ReadModReq	F	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), req MAF(6), req PID(11), Wrap(2)
FetchReq	F	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), req MAF(6), req PID(11), Wrap(2)
SharedtoDirtyReq	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), req MAF(6), req PID(11)
SharedtoDirtySTCReq	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), req MAF(6), req PID(11)
InvaltoDirtyReq	F	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), req MAF(6), req PID(11)

13.1.3 Messages on the FORWARD_CHANNEL

Table 13–3 lists messages on the FORWARD_CHANNEL.

Table 13–3 Messages on the FORWARD_CHANNEL

Command	Direction	Contents
Forward Packet Format (3 Ticks)		
ReadForward	T	Route(16), Dealloc(9), Opcode(8), DPID(10), Stripe(1), Block Address(31), req MAF(6), req PID(11), Wrap(2)
ReadSharedForward	T	Route(16), Dealloc(9), Opcode(8), DPID(10), Stripe(1), Block Address(31), req MAF(6), req PID(11), Wrap(2)
ReadModForward	T	Route(16), Dealloc(9), Opcode(8), DPID(10), Stripe(1), Block Address(31), req MAF(6), req PID(11), Wrap(2)
FetchForward	T	Route(16), Dealloc(9), Opcode(8), DPID(10), Stripe(1), Block Address(31), req MAF(6), req PID(11), Wrap(2)
InvaltoDirtyForward	T	Route(16), Dealloc(9), Opcode(8), DPID(10), Stripe(1), Block Address(31), req MAF(6), req PID(11)
SharedInvalSingle	—	Route(16), Dealloc(9), Opcode(8), DPID(10), Stripe(1), Block Address(31), req MAF(6), req PID(11)
SharedInvalBroadcast	—	Route(16), Dealloc(9), Opcode(8), DPID(10), Stripe(1), Block Address(31), req MAF(6), req PID(11)

13.1.4 Messages on the RESPONSE_CHANNEL

Table 13–4 lists messages on the RESPONSE_CHANNEL.

Table 13–4 Messages on the RESPONSE_CHANNEL

Command	Direction	Contents
Block Response Packet (18 Ticks)		
BlkShared	—	Route(16), Dealloc(6), Opcode(8), req MAF(6), Wrap(2), Data
BlkExclusiveCnt	T	Route(16), Dealloc(6), Opcode(8), req MAF(6), Wrap(2), CohCnt(5), Data
BlkInval	T	Route(16), Dealloc(6), Opcode(8), req MAF(6), Wrap(2), Data
BlkIO	T,H	Route(16), Dealloc(6), Opcode(8), req MAF(6), Wrap(2), Data
Victim Block Response Packet (19 Ticks)		
Victim	F	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), Source PID(11), Data
VictimtoShared	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), Source PID(10), Data
VictimAckExcl	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), Data
VictimAckShared	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), Source PID(10), Data

Table 13–4 Messages on the RESPONSE_CHANNEL (Continued)

Command	Direction	Contents
No Block Response Packet (2 Ticks)		
InvaltoDirtyRespCnt	T	Route(16), Dealloc(6), Opcode(8), req MAF#(6), CohCnt(5)
SharedtoDirtySuccessCnt	—	Route(16), Dealloc(6), Opcode(8), req MAF#(6), CohCnt(5)
SharedtoDirtyProbCnt	—	Route(16), Dealloc(6), Opcode(8), req MAF#(6), CohCnt(5)
SharedtoDirtyFail	—	Route(16), Dealloc(6), Opcode(8), req MAF#(6)
NXMResp	T,H	Route(16), Dealloc(6), Opcode(8), req MAF#(6)
ERRResp	T,H	Route(16), Dealloc(6), Opcode(8), req MAF#(6)
InvalAck	T	Route(16), Dealloc(6), Opcode(8), req MAF#(6)
WrIOAck	T,H	Route(16), Dealloc(6), Opcode(8), req WRIO#(6)
WrIONAck	T	Route(16), Dealloc(6), Opcode(8), req WRIO#(6)
Release Response Packet (3 Ticks)		
VictimClean	F	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), Source PID(11)
VictimCleantoShared	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), Source PID(10)
ForwardAckExcl	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31)
ForwardAckShared	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31), Source PID(10)
ForwardMiss	F	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31)
SharedtoDirtyComplete	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31)
SharedtoDirtyRelease	—	Route(16), Dealloc(9), Opcode(8), Stripe(1), Block Address(31)

13.1.5 Messages on a SPECIAL_CHANNEL

Table 13–5 lists messages on the SPECIAL_CHANNEL.

Table 13–5 Messages on a SPECIAL_CHANNEL

Command	Direction	Contents
NZNOP	T,F	Opcode(8), Dealloc(6), Alert(5)
SpecialInvalBroadcast	—	Route(16), Dealloc(9), Opcode(8), Dir PID(10), Stripe(1), Block Address(31), req PID(11)

13.2 Message Format Details

Physical channels have 32 bits of information plus a 7-bit SECDED ECC code. Packets are always sent contiguously in time.

13.2.1 Route Information

This information tells where to route a message. Along the taken route it also (dynamically) determines the buffers that are used.

The 16 route information bits in the first tick of each message are shown in Table 13–6. This table shows that:

- Most of these bits do not change their value as a message hops.
- "Virtual channel" is recalculated for every corner turn.
- "Did adapt" inverts every time a message converts from adaptive/deadlock-free buffers (and vice-versa).
- "Virtual channel" is unused (but must be preserved) when "did adapt" is asserted.
- "Did adapt" need not be zero when "can adapt" is zero:
  - A message in the IO_CHANNEL may take an initial hop into the adaptive/initial buffers.
  - A message not in the IO_CHANNEL that has "can adapt" clear can still route in the adaptive buffers – a clear "can adapt" implies only that the message must route in a fixed path.
- The 21464 uses only four of the NS value and EW value bits because the largest configuration is 16 processors in each dimension.

Table 13–6 Route Information Bits

Bits	Bit Position(s)	Meaning
NS dir (1)		NS direction to travel (north or south).
NS value (5)		No more NS travel needed when NS WHOAMI is this value.
EW dir (1)		EW direction to travel (east or west).
EW value (5)		No more EW travel needed when EW WHOAMI is this value.
Can adapt (1)		This message can travel in adaptive directions.
Is for I/O (1)		This message is for the I/O channel (or CSR master) at the node.
Virtual channel (1)		Select which of the two virtual channels in this direction.
Did adapt (1)		Place this message into an adaptive buffer.

The opcode bits are included in the first tick of each message so that the length of the message and the exact virtual channel can be determined quickly.

13.2.2 Flow Control and Dealloc Information

The flow control works as follows (thought of as "credit-based"):

- Senders and receivers are paired (in one direction one is the sender and the other the receiver, and in the other direction their roles are inverted).
- The sender has N buffers in each class to send into (the actual N value varies for each class).
- Each sent packet allocates one of the buffers.
- The sender knows how many buffers are available. If there may be an overflow, sender stops sending until space is available.
- The "dealloc" encodings listed in Table 13–7 are used to free up buffer space of each class.
- There is one deallocation response from the receiver for each message sent.

The buffer space is deallocated using 3-bit signals sent as part of the header information with each message sent along the corresponding return channel. Each tick in the control portion of the packets contains at least one 3-bit signal. The packets, including cache blocks, contain one or more 3-bit signals in each control tick.

The concatenation of all the dealloc information included in all messages produces a string of 3-bit deallocation signals. The encoding is a "huffman" encoding that may require multiple 3-bit signals per deallocation. (The dealloc encoding to release a multiple-signal deallocation may span across multiple messages.) As listed in Table 13–7, deallocation of the adaptive messages requires a single 3-bit signal, while the virtual channels require two 3-bit signals. Also, the RDIO and WRIO packets are merged into one.

Table 13–7 Dealloc 3-Bit Variable-Length Encoding (IPs)

Code	Meaning
0	Nop
1	Special inval broadcast complete
61	Special inval broadcast (special)
2	Request adaptive
62	Request virtual channel 0
72	Request virtual channel 1
3	Forward adaptive
63	Forward virtual channel 0
73	Forward virtual channel 1
4	Response non-block adaptive
64	Response non-block virtual channel 0
74	Response non-block virtual channel 1
5	Response block adaptive
65	Response block virtual channel 0
75	Response block virtual channel 1

Table 13–7 Dealloc 3-Bit Variable-Length Encoding (IPs) (Continued)

Code	Meaning
60	Read I/O initial/adaptive
66	Read I/O virtual channel 0
67	Read I/O virtual channel 1
70	Write I/O initial/adaptive
76	Write I/O virtual channel 0
77	Write I/O virtual channel 1

The inval broadcast packets are given special buffering, separate from the rest of the traffic. There is also inval completion information that flows on the dealloc channel. See Section 13.3 for a description of the special operation of the inval broadcast packets.

Table 13–8 shows the message formats that can flow on each set of buffers.

Table 13–8 Buffer Message Formats

Buffer Pool	Size of Buffer (Ticks)	Number of Entries (adap+vc0+vc1)	Formats Included
Request	3	8+1+1	Request Packet
Forward	3	8+1+1	Forward Packet
Response non-block	3	8+1+1	No Block Response Packet Release Response Packet Interrupt Response Packet
Response block	19	3+1+1	Block Response Packet Victim Block Response Packet
Read I/O	3	1+2+2	RdBytes Packet
Write I/O	19	1+2+2	WrBytes Packet
Inval broadcast	3	8	Inval Broadcast Packet

The "size of buffer" is the maximum number of ticks in each format included in that buffer pool.

The I/O port has a simpler single-tick dealloc encoding as listed in Table 13–9. It is simpler because the I/O port does not separate the adaptive and virtual channel buffers.

Table 13–9 Dealloc 3-Bit Encoding (I/O port)

Code	Meaning
0	Nop
1	Unused
2	Request
3	Forward
4	Response non-block
5	Response block
6	Read I/O
7	Write I/O

Table 13–10 shows the size and number of each buffer on the I/O port.

Table 13–10 I/O Port Buffer Size and Number

Buffer Pool	Size of Buffer	Number In	Number Out
Request	3	8	—
Forward	3	—	1–8
Response non-block	3	8	1–8
Response block	19	4	1–8
Read I/O	3	2	1–8
Write I/O	19	2	1–8

The column, *Number In*, indicates the number of buffers that the 21464 has on its I/O port. The number out (on the IO7 ASIC) is variable as specified in the RBOX_IO_BUF register.

Table 13–11 shows the size and number of each buffer for each of the Zports.

Table 13–11 Zport Buffer Message Format

Buffer Pool	Size of Buffer (ticks)	Number of Entries	Formats Included
Forward	3	8	Forward Packet
Response non-block	3	8+1[1]	No Block Response Packet Release Response Packet Interrupt Response Packet
Response block	19	4	Block Response Packet

[1] The 1 extra is used by the broadcast inval widget

Table 13–12 shows the size and number of each buffer on the Cport.

Table 13–12 Cport Buffer Message Format

Buffer Pool	Size of Buffer (ticks)	Number of Entries	Formats included
Request	3	8	Request Packet
Response non-block	3	4+4[1]	No Block Response Packet Release Response Packet Interrupt Response Packet
Response block	19	6	Block Response Packet
Read IO	3	4	Rdbytes Packet
Write IO	19	4	Wrbytes Packet

[1] The 4 extra are used by the broadcast inval widget

13.2.3 Packet Formats

Table 13–13 lists the packet format identifiers and thier contents.

Table 13–13 Packet Formats

Identifier	Contents
OP	Opcode (8 bits).
ADD	Block address of block at DPID (offset at DPID) (32 bits).
STRIPE	Stripe bit.
RMAF	Requestor MAF (6 bits).
RWRIO	Requestor WRIO number (6 bits).
RPID[1]	Requestor PID (11 bits) – uppermost bit is I/O.
DPID	PID of directory for block (10 bits).
SPID[1]	Source PID (11 or 10 bits depending if it can be sourced from I/O).
DESTPID[1]	PID for a packet generated by the IO7 ASIC (11 bits). The DESTPID indicates the destination for the packet. (An IO7 ASIC can address another IO7 ASIC.) DESTPID[10] must be set for RdIPR and WrIPR operations received by the 21464 on the incoming I/O port.
IOADD	I/O space address bits (32 bits).
QWADD	Bits below cache block for I/O address (3 bits).
IOMASK	Bits indicating bytes/longwords/quadwords read/written (8 bits).
COUNT	Coherency count.
DATA	Data bits.
ECC	ECC bits.
TBD	For header ticks from the IO7 ASIC – value depends on the opcode.
WRAP	Indication of which 128 bits from the block will/should arrive first. The wrap is address bits [5:4] and specifies the order of the octaword transfers. The following table specifies the order of the longwords, where X.Y means longword Y within octaword X.
ALERT	4 bits allocated, 3 used: one HW ALERT, one SW ALERT, one HW SYNCH (not needed for the I/O port).
MBZ	Must be zero.

Wrap Value	Wrapped Longword Order in the Packet
0	0.0, 0.1, 0.2, 0.3, 1.0, 1.1, 1.2, 1.3, 2.0, 2.1, 2.2, 2.3, 3.0, 3.1, 3.2, 3.3
1	1.0, 1.1, 1.2, 1.3, 2.0, 2.1, 2.2, 2.3, 3.0, 3.1, 3.2, 3.3, 0.0, 0.1, 0.2, 0.3
2	2.0, 2.1, 2.2, 2.3, 3.0, 3.1, 3.2, 3.3, 0.0, 0.1, 0.2, 0.3, 1.0, 1.1, 1.2, 1.3
3	3.0, 3.1, 3.2, 3.3, 0.0, 0.1, 0.2, 0.3, 1.0, 1.1, 1.2, 1.3, 2.0, 2.1, 2.2, 2.3

[1] RPID, SPID, and DESTPID are typically one more bit than DPID because a DPID cannot be an IO7 ASIC.

Packet formats are listed in Tables 13–14 through 13–19. In all formats, bits ADD[34:7] equal OFF[34:7] (??OLD SECTION 3.1??). Bits ADD[36:35] and IOADD[37:35] are unused (zeros). Bit ADD[34] must be zero when not in 32GB/processor mode.

In Tables 13–14 through 13–19, RPID[10], SPID[10], DESTPID[10] when asserted, indicate the IO7 ASIC connected to the corresponding 21464. RPID[9:8], SPID[9:8], IPID[9:8], DESTPID[9:8], and DPID[9:8] are unused (zeros).

Bits RMAF[5:4] are unused, but all of RMAF[5:0] are passed on; therefore, each IO7 ASIC can have up to 64 outstanding references.

Bits RWRIO[5:0] are unused by the 21464 because outstanding write I/Os are tracked only by means of the account; however, all of RWRIO[5:0] are passed on. Therefore, each IO7 ASIC can track up to 64 outstanding I/O writes.

In the following formats, the most-significant bits are on the left (the ECC bits are a separate field). The line following each header tick line indicates the bit positions of the corresponding field in the header tick line.

13.2.3.1 IO_CHANNEL Formats

Table 13–14 lists the IO_CHANNEL packet formats.

Table 13–14 I/O_CHANNEL Formats (3 Ticks)

Tick	Contents							
RdBytes Packet Format (3 Ticks)								
Tick 0	ROUTE[15:0]				DEALLOC[2:0]	OP[7:0]	IOADD[10:6]	ECC[6:0]
	[31:16]				[15:13]	[12:5]	[4:0]	
Tick 1	IOADD[37:22]				DEALLOC[5:3]	IOADD[21:11]	QWADD[5:4]	ECC[6:0]
	[31:16]				[15:13]	[12:2]	[1:0]	
Tick 2	RPID[10:0]	QWADD[3]	RMAF[5]	SPARE[2:0]	DEALLOC[8:6]	IOMASK[7:0]	RMAF[4:0]	ECC[6:0]
	[31:21]	[20]	[19]	[18:16]	[15:13]	[12:5]	[4:0]	
WrBytes Packet Format (19 Ticks)								
Tick 0	ROUTE[15:0]				DEALLOC[2:0]	OP[7:0]	IOADD[10:6]	ECC[6:0]
	[31:16]				[15:13]	[12:5]	[4:0]	
Tick 1	IOADD[37:22]				DEALLOC[5:3]	IOADD[21:11]	QWADD[5:4]	ECC[6:0]
	[31:16]				[15:13]	[12:2]	[1:0]	
Tick 2	RPID[10:0]	QWADD[3]	RWRIO[5]	SPARE[2:0]	DEALLOC[8:6]	IOMASK[7:0]	RWRIO[4:0]	ECC[6:0]
	[31:21]	[20]	[19]	[18:16]	[15:13]	[12:5]	[4:0]	
First Wrapped Octaword:								
Tick 3	DATA[31:0]	ECC[6:0] (low-order longword of the octaword)						
Tick 4	DATA[63:32]	ECC[6:0]						
Tick 5	DATA[95:64]	ECC[6:0]						
Tick 6	DATA[127:96]	ECC[6:0] (high-order longword of the octaword)						

13.2.3.2 REQUEST_CHANNEL Format

Table 13–15 lists the REQUEST_CHANNEL packet format.

Table 13–15 REQUEST_CHANNEL Format

Tick	Contents							
REQUEST_CHANNEL Packet Format (3 Ticks)								
Tick 0	ROUTE[15:0] [31:16]				DEALLOC[2:0] [15:13]	OP[7:0] [12:5]	ADD[10:6] [4:0]	ECC[6:0]
Tick 1	STRIPE[0] [31]	ADD[36:22] [30:16]			DEALLOC[5:3] [15:13]	ADD[21:11] [12:2]	WRAP[5:4] [1:0]	ECC[6:0]
Tick 2	RPID[10:0] [31:21]	SPARE[0] [20]	RMAF[5] [19]	SPARE[2:0] [18:16]	DEALLOC[8:6] [15:13]	SPARE[7:0] [12:5]	RMAF[4:0] [4:0]	ECC[6:0]

13.2.3.3 FORWARD_CHANNEL Format

Table 13–16 lists the FORWARD_CHANNEL packet format.

Table 13–16 FORWARD_CHANNEL Format

Tick	Contents								
FORWARD_CHANNEL Packet Format (3 Ticks)									
Tick 0	ROUTE[15:0] [31:16]					DEALLOC[2:0] [15:13]	OP[7:0] [12:5]	ADD[10:6] [4:0]	ECC[6:0]
Tick 1	STRIPE[0] [31]	ADD[36:22] [30:16]				DEALLOC[5:3] [15:13]	ADD[21:11] [12:2]	WRAP[5:4] [1:0]	ECC[6:0]
Tick 2	RPID[10:0] [31:21]	SPARE[0] [20]	RMAF[5] [19]	SPARE[0] [18]	DPID[9:8] [17:16]	DEALLOC[8:6] [15:13]	DPID[7:0] [12:5]	RMAF[4:0] [4:0]	ECC[6:0]

13.2.3.4 RESPONSE_CHANNEL Formats

Table 13–17 lists the RESPONSE_CHANNEL packet formats.

Table 13–17 RESPONSE_CHANNEL Formats

Tick	Contents							
Block Response Packet (18 Ticks)								
Tick 0	ROUTE[15:0] [31:16]			DEALLOC[2:0] [15:13]	OP[7:0] [12:5]	RMAF[4:0] [4:0]		ECC[6:0]
Tick 1	SPARE[11:0] [31:20]	RMAF[5] [19]	SPARE[2:0] [18:16]	DEALLOC[5:3] [15:13]	SPARE[5:0] [12:7]	COUNT[4:0] [6:2]	WRAP[5:4] [1:0]	ECC[6:0]
First Wrapped Octaword:								
Tick 2	DATA[31:0]	ECC[6:0] (low-order longword of the octaword)						
Tick 3	DATA[63:32]	ECC[6:0]						
Tick 4	DATA[95:64]	ECC[6:0]						
Tick 5	DATA[127:96]	ECC[6:0] (high-order longword of the octaword)						
Second Wrapped Octaword:								
Tick 6	DATA[31:0]	ECC[6:0] (low-order longword of the octaword)						
Tick 7	DATA[63:32]	ECC[6:0]						
Tick 8	DATA[95:64]	ECC[6:0]						
Tick 9	DATA[127:96]	ECC[6:0] (high-order longword of the octaword)						
Victim Block Response Packet (19 Ticks)								
Tick 0	ROUTE[15:0] [31:16]		DEALLOC[2:0] [15:13]	OP[7:0] [12:5]	ADD[10:6] [4:0]	ECC[6:0]		
Tick 1	STRIPE[0] [31]	ADD[36:22] [30:16]	DEALLOC[5:3] [15:13]	ADD[21:11] [12:2]	MBZ[1:0] [1:0]	ECC[6:0]		
Tick 2	SPID[10:0] [31:21]	SPARE[4:0] [20:16]	DEALLOC[8:6] [15:13]		SPARE[12:0] [12:0]	ECC[6:0]		
First Unwrapped Octaword:								
Tick 3	DATA[31:0]	ECC[6:0] (low-order longword of the octaword)						
Tick 4	DATA[63:32]	ECC[6:0]						
Tick 5	DATA[95:64]	ECC[6:0]						
Tick 6	DATA[127:96]	ECC[6:0] (high-order longword of the octaword)						
No Block Response Packet (2 Ticks)								
Tick 0	ROUTE[15:0] [31:16]			DEALLOC[2:0] [15:13]	OP[7:0] [12:5]	RMAF[4:0] [4:0]		ECC[6:0]
Tick 1	SPARE[11:0] [31:20]	RMAF[5] [19]	SPARE[2:0] [18:16]	DEALLOC[5:3] [15:13]	SPARE[5:0] [12:7]	COUNT[4:0] [6:2]	SPARE[1:0] [1:0]	ECC[6:0]

Table 13–17 RESPONSE_CHANNEL Formats (Continued)

Tick	Contents						
Release Response Packet (3 Ticks)							
Tick 0	ROUTE[15:0]		DEALLOC[2:0]	OP[7:0]	ADD[10:6]	ECC[6:0]	
	[31:16]		[15:13]	[12:5]	[4:0]		
Tick 1	STRIPE[0]	ADD[36:22]	DEALLOC[5:3]	ADD[21:11]	SPARE[1:0]	ECC[6:0]	
	[31]	[30:16]	[15:13]	[12:2]	[1:0]		
Tick 2	SPID[10:0]	SPARE[4:0]	DEALLOC[8:6]		SPARE[12:0]	ECC[6:0]	
	[31:21]	[20:16]	[15:13]		[12:0]		

13.2.3.5 SPECIAL_CHANNEL Formats

Table 13–18 lists the SPECIAL_CHANNEL packet formats.

Table 13–18 SPECIAL_CHANNEL Formats

Tick	Contents							
Nop Packet (1 Tick)								
Tick 0	SPARE[12:0]	DEALLOC[5:0]	OP[7:0]	SPARE[1:0]	SYNCH[0]	SW_ALERT[0]	HW_ALERT[0]	ECC[6:0]
	[31:19]	[18:13]	[12:5]	[4:3]	[2]	[1]	[0]	
Inval Broadcast Packet Format (3 Ticks)								
Tick 0	ROUTE[15:0]			DEALLOC[2:0]	OP[7:0]	ADD[10:6]	ECC[6:0]	
	[31:16]			[15:13]	[12:5]	[4:0]		
Tick 1	STRIPE[0]	ADD[37:22]		DEALLOC[5:3]	ADD[21:11]	SPARE[1:0]	ECC[6:0]	
	[31]	[30:16]		[15:13]	[12:2]	[1:0]		
Tick 2	RPID[10:0]	SPARE[2:0]	DPID[9:8]	DEALLOC[8:6]	DPID[7:0]	SPARE[4:0]	ECC[6:0]	
	[31:21]	[20:18]	[17:16]	[15:13]	[12:5]	[4:0]		

13.2.3.6 INPUT I/O PORT HEADER TICK Formats

Table 13–19 lists the INPUT I/O PORT HEADER TICK packet formats.

Table 13–19 INPUT I/O PORT HEADER TICK Formats

Tick	Contents					
Nop Packet (1 Tick)						
Tick 0	SPARE[15:0]		DEALLOC[2:0]	OP[7:0]	ALERT[4:0]	ECC[6:0]
	[31:16]		[15:13]	[12:5]	[4:0]	
All Other Packets (2 – 19 Ticks)						
Tick 0	SPARE[4:0]	DESTPID[10:0]	DEALLOC[2:0]	OP[7:0]	TBD[10:6]	ECC[6:0]
	[31:27]	[26:16]	[15:13]	[12:5]	[4:0]	

13.2.3.7 ROUTE FIELD Format

Table 13–20 lists the ROUTE FIELD format.

Table 13–20 ROUTE FIELD Format

ROUTE Bit	Meaning
ROUTE[0]	Did adapt
[16]	
ROUTE[1]	Virtual channel
[17]	
ROUTE[2]	Is for I/O
[18]	
ROUTE[3]	Can adapt
[19]	
ROUTE[8:4]	EW value
[24:20]	
ROUTE[9]	EW direction
[25]	
ROUTE[14:10]	NS value
[30:26]	
ROUTE[15]	NS direction
[31]	

13.3 SharedInvalBroadcast Details

SharedInvalBroadcasts are sent from the directory to one of the nodes in the cluster of processors sharing a mask bit. That processor receives the SharedInvalBroadcast and buffers it in an internal structure called the inval widget. This is the root of the inval fanin/fanout tree.

SpecialInvalBroadcast messages are fanned out within the cluster from node to node. At each node a new inval widget entry is allocated. The inval widget entry waits for all the children processors in its subtree to complete before it completes. It also performs the invalidate on the local processor (if the local processor is not the requesting processor) and must wait for the local inval to complete before the inval widget entry completes. This means that once the children of the root node complete and the root node itself performs its inval, all of the invalidations are complete. Once a node is complete, the inval widget entry is deallocated and a SpecialInvalBroadcast complete message is sent to the parent node. Once the root node completes the SharedInvalBroadcast, an InvalAck is returned to the requesting processor.

To avoid deadlock, SpecialInvalBroadcast messages are expected to be fanned out (and back in) in a dimension order. This, combined with the fact that the inval widgets are specific to particular inputs, allows multiple broadcast messages to be fanned out from different starting points while avoiding deadlock.

SpecialInvalBroadcast messages have their own special buffering. This buffering is deallocated in the normal way. The SpecialInvalBroadcast complete signals are also transferred along the deallocation channel.

13.4 I/O Port and I/O ASIC Assumptions

The I/O ASIC communicates with the rest of 21464 via the I/O port. The packet formats are the same as the 21464-to-21464 packet formats. The I/O ASIC can only issue a subset of the commands, and only needs to be able to receive a subset of the commannd, as described in Section 13.5.

The interface to the I/O ASIC follows the same dealloc strategy as the 21464-to-21464 ports. The only difference is that the packets coming from the I/O ASIC encode the destination PID in place of the route information. (The 21464 then does a routing table lookup and replaces the destination PID with the routing information.)

The header tick format is specified in Section 13.2.3.6

New messages on the out-going I/O port can only be initiated on the rising edge of the out- going forward clock. This can make the decode on the I/O ASIC simpler. The only packet type that can start on the falling edge is the NZNOP instruction (or the true NOP during reset).

An I/O ASIC has a PID with the uppermost bit set. The lower PID bits equal the PID of the processor with the I/O port the ASIC is connected to. The PID of the I/O ASIC is what allows the messages routed on the 21464 interconnect to reach the I/O device -- first route to the processor with the same lower bits, then route out the I/O port.

Here are the high-level operations supported in the 21464 I/O port interface:

- DMA read and write access by the I/O ASIC to the memory in the 21464 system
- Read and write access by the 21464 microprocessors to registers in the I/O ASIC and on the I/O buses connected to the I/O ASIC
- Read and write access by the I/O ASIC to the system IPR's in the 21464 microprocessors

DMA read and write access to memory space via the I/O ASIC is described in Section 13.6.

The read and write access by the 21464 microprocessors to the registers in the I/O ASIC and on the I/O buses connected to the I/O ASIC allow the microprocessors to control the I/O devices connected on the port. The 21464 interconnect has enough virtual channels and the coherence protocol is such that the I/O ASIC may stall these accesses pending completion of DMA references and the system will not deadlock.

I/O ASIC references can read/write the system IPR's of any 21464 in the system. This allows for 21464 system IPR's to be configured by the I/O ASIC or another device on an I/O bus connected to the I/O ASIC. This is also the mechanism by which interrupts are delivered from an I/O device to a 21464 (see Section 13.5 for more information on

interrupts). Note that these references must never block either of the two prior types of access (DMA or I/O register access by the microprocessor), otherwise deadlock may occur.

Note that an I/O ASIC may send an IO_CHANNEL Rd* or Wr* message to another I/O ASIC in order to implement peer-to-peer I/O. See Section 13.7 for deadlock-avoidance requirements.

Note that the I/O port protocol does not explicitely support coherent I/O TLB's, but that I/O TLB coherence can be maintained by hardware exclusive caching of TLB entries.

The I/O port also has a synchronous mode for lock-step operation. In this mode, data from the I/O port input is not directly taken into the router core. Rather, it is written into a four-entry FIFO using a (two-bit) write pointer. The router core later reads the data from this FIFO using its two-bit read pointer. The FIFO write path writes a piece of data into the FIFO and increments the write pointer every cycle that valid data is sampled. The FIFO read path reads a FIFO entry and increments the read pointer at the rate of the incoming data, synchronous to the internal 21464 clock.

This synchronous mode allows the 21464 router core to sample data from the incoming I/O port (via the FIFO) at a predictable time even though the 21464 pads may sample the incoming I/O port data at an unpredictable time. Short-term jitters can be tolerated. Over the long term, the system must be synchronous.

The read and write pointers are initialized as follows. At boot time the 21464 forward clocks on the I/O output ports are not transitioning. Also, the forward clocks in the I/O input ports are not transitioning. The write pointer increments only when an incoming forward clock is received, so the write pointer is initialized at this time. At some point boot software starts the outgoing forward clocks from 21464. It does this by a write to the RBOX_IO_CFG IPR. During the same register write it also initializes the read pointer to the appropriate value. The I/O ASIC must detect the start of the forward clock from 21464 and start its forward clocks a fixed (predictable) amount of time later. When the 21464 starts receiving the forward clocks from the I/O ASIC (a fixed time later), the write pointer starts incrementing. Since this time is fixed, it should have been possible to pre-calculate the necessary read pointer value. (Either that, or try all 4 possible combinations.) The same read pointer value can be used from one boot to the next even though the latency for the forward clocks to transition may vary slightly from one boot to the next.

13.5 Interrupt Delivery

There are two mechanisms to deliver an interrupt to an 21464 microprocessor:

- Queueing an identifier
- Setting a mask bit

21464 includes a 4 entry queue to hold 24-bit identifiers that can uniquely identify the source of an interrupt. These 24-bit identifiers are called interrupt id's (IID's). Interrupt software can read the head of the queue to determine how to process an interrupt. This queue is accessible via references to the RBOX_INTQ and RBOX_INTA IPR's.

21464 also includes a 32 bit mask for coarser interrupt receipt. This mask can be referenced via the RBOX_INT IPR. The individual interrupts can be masked via the RBOX_IMASK system IPR. Some of the bits in this mask will be reserved for specific

purposes - e.g. interrupt queue, performance counter, error, I/O error, and other bits will be available for general software use - e.g. interprocessor interrupt. New interrupts can be launched via the RBOX_IREQ IPR.

I/O devices will typically queue an IID to produce an interrupt. In order to use this method, the I/O ASIC will issue a write to the RBOX_INTA IPR in the appropriate 21464. The I/O device must be prepared to receive a WrIONAck response indicating that the given 21464's interrupt queue has overflowed. When the I/O ASIC receives the overflow response, it must resend the interrupt again to the same or another 21464 until it is accepted by one of them.

Interprocessor interrupts will typically be performed via writes (to the RBOX_IREQ IPR) that set a mask bit in the RBOX_INT IPR of another 21464. Interprocessor interrupts will typically not use the interrupt queue method since there is no hardware mechanism to determine when the interrupt queue overflows.

The 21264 core allows for 6 interrupt wires into the core. 21464 will partition the interrupt sources onto these six wires as follows:

Table 13–21 Interrupt Level Sources

Interrupt Level	Source
IRQ(0)	System correctable / performance count
IRQ(1)	Interrupt queue
IRQ(2)	Interval timer
IRQ(3)	Other (interprocessor/SW ALERT)
IRQ(4)	Halt interrupt/other
IRQ(5)	Uncorrectable/machine-check/HW ALERT

13.6 DMA Device Assumptions

This section describes two alternative techniques for the I/O ASIC to perform DMA accesses to the 21464 system memory -- exclusive caching and timeouts.

A DMA Device is contained within the I/O ASIC off the I/O port. Its purpose is to service I/O bus reads and writes.

A DMA device can access data in one of three different ways:

1. An uncached FetchBlk request to read the block
2. A ReadMod request to obtain exclusive access to the block (often to write a portion of the block)
3. An InvaltoDirty request to gain exclusive access to the block (presumably to write the entire block).

For a DMA read stream there are two ways to prefetch data in multipleblocks, depending on the ordering required by the DMA device. The most efficient way is to use a stream of FetchReq (i.e. non-cacheable fetch) commands. As an example, the I/O controller might Fetch blocks A and B. The references to blocks A and B may be serviced in any order by the memory system, and the responses may return in any order. Note the two sources of difficulty: (1) the references are serviced out of order, and (2) the refer-

ences may return out of order. Source (1) may violate the memory reference ordering constraints required by the DMA read stream (the returned loads are not sequentially consistent, for example). Source (2) makes the implementation of the DMA controller more difficult because the data may have to be reordered.

The second way to prefetch data in multiple blocks for a DMA read stream is to use ReadModReq commands. The advantage of this method is that the I/O device can implement a sequentially consistent read stream since the exclusive access forces order. One disadvantage is that VictimClean must be generated to release exclusive access to the block. The other disadvantage is that exclusive access is required. Multiple DMA devices that attempt to access the same block at the same time will be serialized, as a consequence, as will a processor and a DMA device.

There are also two ways to prefetch data in multiple blocks for a DMA write streams. The first way is via a stream of ReadModReq commands. The second is via a stream of InvaltoDirtyReq commands. The InvaltoDirtyReq's require that the writes be full-block writes.

Note that the protocol specifies that InvaltoDirty's may be issued speculatively from a DMA device since the memory always contains the prior copy of the block -- a Victim-Clean will back out the request if it is found to be a mis-speculation. Also, the DMA device will never dirty blocks in response to a ReadModReq. This means that Victim commands will never be needed for a DMA read (via ReadMod command) stream.

13.6.1 I/O DMA Access and Exclusive Caching

When using this technique, the DMA device is expected to force the eviction of a cache block soon after receiving a forward for the cache block. The I/O ASIC may (exclusively) cache copies of blocks for long time periods. If a processor or another I/O ASIC (or even if this I/O ASIC) requests a copy of the block, the directory will see that this I/O ASIC is the exclusive owner of the block and will forward the request to the I/O ASIC. When this happens, the directory expects to eventually receive both a Forward-Miss and a Victim (or Victim Clean) in response.

When the I/O ASIC is using this mode to access DMA, it should respond ForwardMiss to every received forward request. The following is additionally required:

- Any currently cached blocks/TLB entries that could possibly match the address in the forward must be marked for eventual eviction (after a time-out)
- Any currently pending MAF entries that could possibly match the address must be marked so that the block eventually gets evicted after it returns.

Note that the receipt of a forward does not imply that the I/O ASIC currently holds a copy of the block. (A victim may be on its way from the I/O ASIC to the directory before the I/O ASIC receives the forward.)

Note that this scheme allows the I/O ASIC to (exclusively) cache copies of scatter-gather maps, or I/O TLB entries.

13.6.2 I/O DMA Access via Timeouts

When using this technique, the DMA device is expected to evict blocks soon after they obtain exclusive access to the block. This allows the I/O ASIC to ignore the forwards.

When the I/O ASIC is using this mode to access DMA, it should simply respond ForwardMiss to every received forward request, and otherwise ignore the forward.

DMA devices must take care to avoid deadlock in this mode. Take the following scenario:

1. The DMA device requests exclusive access to blocks A and B simultaneously,
2. The response for block B returns but cannot be written until the response for block A returns. In this scenario deadlock could result if the DMA device does not eventually release exclusive access to the block B. It is possible that the response to the request A cannot be completed since requests are blocked waiting for the eviction of B. Thus, after a timeout, block B must be released in order to make forward progress, even though the reference to the block has not been completed.

Note also that I/O TLB's may not be cached (long-term) when this timeout mechanism is used.

13.7 I/O Space Ordering and Assumptions

21464 supports the same I/O space ordering rules as the 21264: LD-LD ordering is maintained to the same I/O ASIC or processor, ST-ST ordering is maintained to the same I/O ASIC or processor, LD-ST or ST-LD ordering is maintained to the same address, and LD-ST or ST-LD ordering is not maintained when the addresses are different.

All these ordering constraints are on a single processor basis to the same I/O ASIC or processor. Multiple loads (to the same or different addresses) may be in flight without being responded to, though their in-flight order is maintained to the destination by the core/CBOX and the router. Similarly, multiple stores (to the same or different addresses) can be in flight.

When there is a load I/O outstanding to address A, 21464 will not launch a store to address A until the BlkIO response to the load I/O is received. 21464 may have an earlier write I/O request to address B in flight at the same time as there are load I/O requests in flight to address B; the CBOX/router guarantee that the earlier write I/O request reaches the destination before the later load I/O requests.

21464 also supports peer-to-peer I/O. In order to avoid deadlock among peer I/O ASIC clients, writes must be able to bypass prior reads. This is required because read responses cannot be returned until prior writes have completed in order to maintain some PCI ordering constraints. By allowing the writes to bypass the reads, we guarantee that the writes will eventually drain, thereby guaranteeing that the reads will eventually drain.

In order to implement all these requirements, the 21464 router must maintain the following point-to-point rules on the IO_CHANNEL:

Table 13–22 Router IO_CHANNEL Point-to-Point Rules

First	Second	Description
Rd*	Rd*	Order must be maintained
Rd*	Wr*	The later Wr* must be allowed to bypass the earlier Rd* to avoid deadlock
Wr*	Rd*	Order must be maintained
Wr*	Wr*	Order must be maintained

In other words, except for the case of a read followed by a write, a total order must be maintained.

Note that 21464 does not support instruction references to I/O space. 21464 cannot execute code received directly from the I/O ASIC. Code residing in I/O space must first be copied/DMA'ed into cacheable memory before it can be directly executed.

Note that all I/O writes are acknowledged. MB's wait for all I/O write acknowledgements to be received before proceeding. MB's also wait for the response to all I/O reads before proceeding.

Note also there there are no ordering constraints between different I/O space accesses that reference different I/O ASIC's or processors; the ordering rules apply only with the same source and destination for the references. MB's must be used to order references to different I/O ASIC's or processors.

14
Rambus Interface — the Zbox

Introductury information about the Zbox is located in Section 2.7.3. Information about the Zbox IPRs is located in Section 16.6.

14.1 The 5th Rambus Channel

For higher reliability in large memory systems, a fifth Rambus channel (one extra for every four channels) can optionally be enabled. This extra channel allows the system to tolerate the failure of any single DRAM part or any single DRAM row.

The technique used is one used in RAID schemes for disks. (Is it RAID 5?) When enabled, the stored bits in the 5th channel are the bit-wise exclusive-or of the corresponding 4 bits in the original 4 channels. If the stored bits in channel i are a bit-stream Pi, and if the expected information for channel i is a bit-stream Vi, then the operation to store the expected bit-stream to memory is:

```
P0 = V0
P1 = V1
P2 = V2
P3 = V3
P4 = V0 ^ V1 ^ V2 ^ V3
```

And, the operation to read the expected bit-stream from memory under normal operation is:

```
V0 = P0
V1 = P1
V2 = P2
V3 = P3
check that (P4 == P0 ^ P1 ^ P2 ^ P3)
```

In the case when the parity check calculation on the stored bits indicates that there are no mismatches, the read operation is complete; We assume that the reconstructed data V0-V3 is correct. In the case that there is a single-bit mismatch on the parity check calculation, the read operation is also complete; We assume that the ECC codes contained in V0-V3 will correct the (likely) resultant single-bit error.

However, in the case when the parity check calculation mismatches on more than one bit, this indicates that the resultant data V0-V3 may have a multi-bit error that is uncorrectable, and that it may have bad data that appears either good or correctable. In this case we can attempt to correct the error by mapping out a channel.

The algorithm to map out a channel is simply to try all possible combinations and pick the one that results in the cleanest resultant ECC codes.

After we have decided to map out a physical channel, we can reconstruct the expected data using the redundant information contained in the 5th channel. If we assume that P0 has failed, the read operation to reconstruct the original data would be:

```
V0 = P1 ^ P2 ^ P3 ^ P4
V1 = P1
V2 = P2
V3 = P3
```

Similar calculations can be applied when mapping out one of the other four channels.

Note that after we map a channel out, we still store the expected bit-stream to memory in the same way that we did before we mapped the channel out. This makes it simple to remap the channel back in when there is a soft-error; After re-writing the bits to memory the channel can be restored to full functionality.

Note also that when the 5th channel is enabled, correctable memory errors will not propagate; All errors that can be corrected by remapping a channel will be corrected without sending corrupt data on the network or putting corrupt data in the local caches. (When the 5th channel is not enabled, corrupted memory data may be propagated in the network (via "garbage codes") and may also be written into the local caches.

15
Miscellaneous Interfaces

15.1 The GIO Port

As in the 21363, the GIO port is a way for the 21464 to interface with miscellaneous external I/O functions. During power-up initialization sequences, the GIO port provides access to system configuration information including PLL divisors, WHOAMI and other router configuration components, the Rambus SIO and I2O chains. The GIO port also provides a connection to server management memory where XSROM or console code can be loaded or mailboxes set-up to allow communication with the server management subsystem.

The functional model is for the 21464 to control the operation of the port performing all reads and writes. Status bits and interrupts are used for reverse communications. This is in contrast to the JTAG port where the system is the master and initiates all read and write transactions to the 21464.

15.1.1 Signals

The GIO port is a low bandwidth, simple interface to external logic. In currently planned systems the GIO port is expected to interface with an external FPGA running in the 33Mhz range. The GIO port signals are:

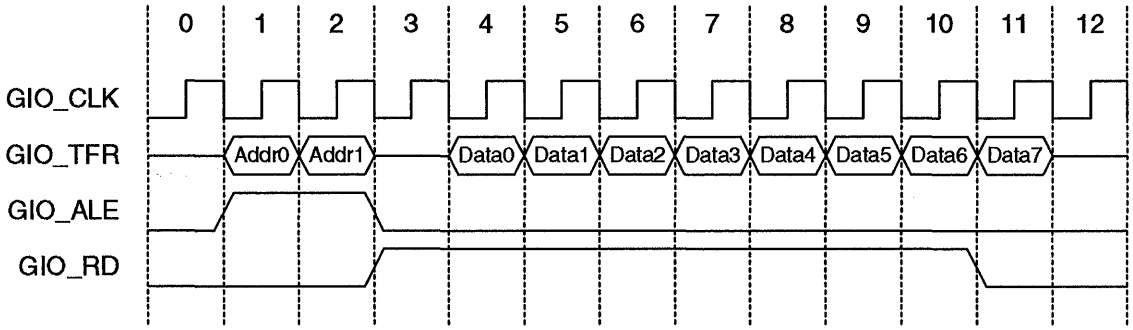
Table 15–1 GIO Port Signals

Name	Path	Description
GIO_TFR<3:0>	B	Address/Data bus. Transfers 32 data bits per transaction in 8 cycle bursts of 4 bits per cycle. Addresses are 8-bit values transferred in the first two 4-bit cycles.
GIO_ALE	O	Address latch enable signal. Asserts for two cycles at the beginning of each transaction defining when transaction address bits 0-3, then 4-7 are driven on GIO_TFR<3:0>.
GIO_RD	O	Read enable signal. If asserted in cycles 2-9 of a transaction, the operation is a read, if deasserted, the operation is a write transaction.
GIO_CLK	O	Free running clock.
GIO_INT	I	External interrupt signal. Asserted when system logic is requesting communication with the 21464. Interrupt handlers then perform reads and writes across the GIO port to examine and respond to the interrupt request.
GIO_HINT	I	High priority interrupt. Currently just a separate device interrupt but we are evaluating a true HALT type of function.

15.1.2 Transactions

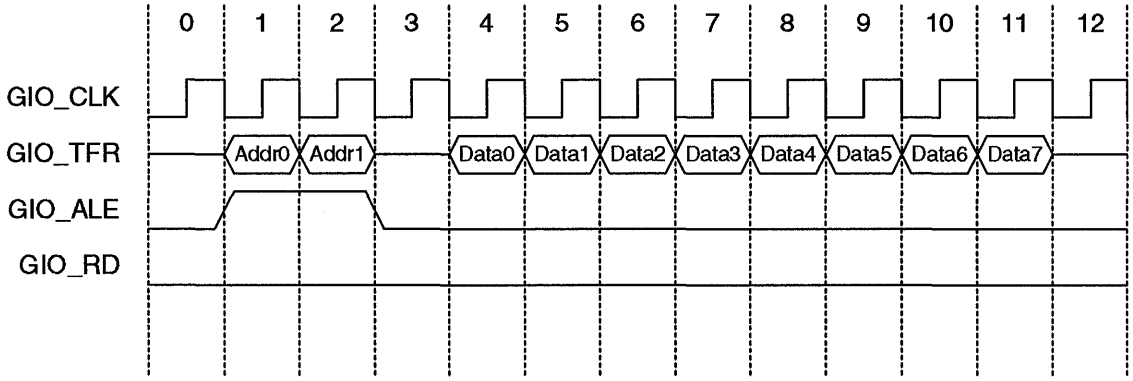
Both read and write transactions are fixed 12 GIO clock cycle operations. Eight bits of address are transferred in the first two cycles and 32 bits of data are transferred in cycles 4-11. Cycles 3 and 12 are used to turn-around the bus for read operations and are included in write transactions to simplify the state machines.

Figure 15–1 GIO Port Read Transaction Timing



Read Transaction

Figure 15–2 GIO Port Write Transaction Timing



Write Transaction

15.1.3 Registers

There are three registers in the 21464 that software uses to interact with the GIO port. These CSRs reside in the Rbox and are mapped to IO space.

15.1.3.1 GIO_CNFG

The GIO_CNFG register defines the characteristics of the GIO_CLK signal. The GIO_CLK pin is disabled by reset (which resets?) and enabled by writing a non-zero value to the Divisor. The Divisor allows GIO_CLK to be between half and 1/256th of the core clock. This allows the GIO port to run as slow as 10MHz on a 2.5 GHz 21464.

Figure 15–3 GIO_CNFG Register

Table 15–2 GIO_CNFG Register Field Descriptions

Field Name	Extent	Type	Description
DIVISOR	7:1	RW,0	Defines the number of Core clock cycles in each GIO_CLK phase (half cycle). A divisor of zero also disables the clock.

15.1.3.2 GIO_ADDR

The GIO_ADDR register defines the address of the next GIO transaction. If the START_READ bit is set when the GIO_ADDR register is written, a read transaction on the GIO bus is initiated. If the START_READ bit is written with a zero, no transaction is initiated.

Software must not write this register while a GIO transaction is in progress. A GIO transaction is in progress when the DONE bit of the GIO_DATA register is clear.

Figure 15–4 GIO_ADDR Register

Table 15–3 GIO_ADDR Register Fields Description

Field Name	Extent	Type	Description
ADDR	7:1	RW,0	Defines the eight-bit address of the GIO transaction.
START_READ	0	RW,0	When written with a 1, a read transaction to ADDR is performed on the GIO bus.

15.1.3.3 GIO_DATA

The GIO_DATA register specifies the data to be written on the GIO bus and holds the data read from the GIO bus on read transactions. When the GIO_DATA register is written, a write transaction is initiated on the GIO bus. Software must poll the DONE bit of the GIO_DATA register to detect the completion of the write transaction and must not perform any subsequent writes to the GIO_DATA register before the DONE bit is set. The DONE bit also indicates completion of a read transaction.

*** I find the sense of the DONE bit to be backwards. It should be a BUSY bit. Software will typically issue a read and poll until complete. The loop control would be a BLT instead of a BGE but with a BUSY bit, the value returned from the completed read would be correct whereas a DONE bit would need to be masked-off before the value was used. A BUSY bit also naturally resets to zero.

*** The BUSY vs. DONE debate is a minor nit. My inclination is to mimic the 21364, rather than implement the more natural interface but comments are welcome.

Figure 15–5 GIO_DATA

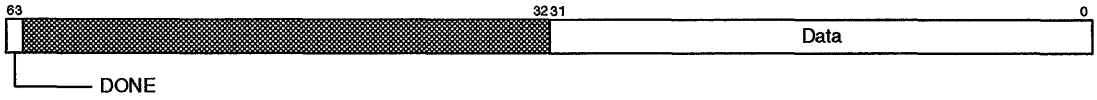


Table 15–4 GIO_DATA Register Fields Description

Field Name	Extent	Type	Description
DONE	63	RO, 1	Status bit indicating the GIO bus controller is idle and has completed any read or write transactions. Software must query this bit to ensure the GIO logic is not busy before updating any GIO registers.
DATA	31:0	RW,0	Data written to the GIO_DATA register is immediately written to the GIO bus. For read transactions, the DATA field is the valid result of the read operation once the DONE bit is set.

15.1.4 Use

Aside from the restrictions imposed by an 8-bit address and 32-bit data word, the 21464 does not define the external structure of the GIO port. Marvel systems intend to use a FPGA to interface to the GIO port so the external structure of the GIO port could be modified by a simple update to the FPGA program.

To better understand the capabilities and operation of the GIO port, consider the proposed interface to server management on a Marvel platform. Marvel has defined several registers in GIO address space:

Table 15–5 GIO Address Space Registers Defined by Marvel

Register	Name
00h	CPUx_CTRL
01h	CPUx_DMA_Data
02h	CPUx_DMA_Addr
03h	CPUx_RIMM_Serial_Port
04h	CPUx_Com0_Xmit
05h	CPUx_Com0_Rcv
06h	CPUx_Com1_Xmit
07h	CPUx_Com1_Rcv

These registers allow three basic functions to be performed:

- Direct interaction with the RAMbus serial controls
- Dual Simple UART communication links
- DMA access to Server Management memory

As an example of how the GIO bus is used, consider the sequence of operations necessary to read a value from server management memory:

Operation	Description
IOstore GIO_ADDR CPUx_DMA_Addr	\\Send DMA address to FPGA
IOstore GIO_DATA addr	
IOload GIO_DATA (until done bit set)	\\Wait for GIO port to idle
IOstore GIO_ADDR CPUx_CTRL	\\Tell FPGA to start the DMA
IOstore GIO_DATA 1	
IOload GIO_DATA (until done bit set)	\\Wait for GIO port to idle
IOstore GIO_ADDR CPUx_CTRL | Start	\\Get DMA status from FPGA
IOload GIO_DATA (until done bit set)	\\Wait for GIO port to idle
(repeat read of CPUx_CTRL until bit<0> clear)	\\Repeat read until DMA done
IOstore GIO_ADDR CPUx_DMA_Data | Start	\\Get DMA data from FPGA
IOload GIO_DATA (until done bit set)	\\Wait for GIO port to idle
(mask off done bit)	

Each three-line group above equates to a single (360ns) GIO bus transaction. For the 16-bit DMA transactions currently planned, the third group would likely execute twice because the actual DMA from server management memory into the FPGA is expected to take ~600ns.

In total the 16-bit transfer would require (5 * 360ns) 1.8us to complete yielding an effective bandwidth of just over 1MB/sec.

If the FPGA implementation can accommodate it, a recommended optimization is to define the DMA start bit and opcode in the CPUx_DMA_Addr register instead of the CPUx_CRTL register. This would merge the first two GIO bus transactions into a single transaction and reduce the latency of each read sequence to 1.44us. Increasing the DMA transfer size to 32-bits per operation is another available option. The bandwidth of large transfers would be increased to over 2MB/sec but the latency of a single operation would likely be extended back to 1.8us.

To initiate communications, server management will post a GIO interrupt after storing a message packet in a predefined location in server management memory. 21464 PALcode, in response to the interrupt, will read the packet and can respond by writing a response packet to a predefined location in server management memory.

Multi-threading creates several problems unique to the 21464:

- Which TPU(s) will service the GIO interrupt?
- The GIO port is a shared resource, how to we prevent multiple TPUs from accessing it simultaneously?

- Does server management need to allocate separate message blocks or addresses for each TPU?

The current plan is to require software to restrict access to the GIO port to a single TPU at a time. A mask register will define which TPU(s) are to receive the GIO interrupt. If multiple TPUs are selected, software must synchronize among the TPUs and ensure there is no contention for the GIO port. We currently do not have a mechanism that allows server management to target specific interrupts to specific TPUs. The mask register must be pre-set.

15.1.4.1 Differences In Implementation Between the 21364 and 21464

The following differences exist between the 21464's GIO port implementation and the 21364 implementation:

- The enable bit in GIO_CNFG does not exist in the 21464. If software ensures the divisor is zero whenever the GIO_CLK should be disabled, both chips will behave identically.
- The 21364 defined GIO transactions to be 64-bit writes and 63-bit reads. The 21464 restricts all GIO transactions to 32 bits. As long as systems define GIO port registers to be no larger than 32-bits, the size difference should be transparent. The largest GIO operation defined in Marvel is currently 23 bits.
- Because the 21464 sequences 4-bits per cycle across the GIO bus, the FPGA will need to shift/pack differently than the 21364, but this will be transparent to software.
- It has been requested that we make the GIO_HINT pin perform a real halt function rather than act as another device interrupt. This is under consideration.

The motivation for implementing a different interface from the 21364 was primarily the availability of pins. The Marvel CMM module connector and FPGA are already pin constrained. The 21464 will have additional voltages to control and needs to connect to the JTAG port for access to debug controls. Optimizing the GIO port to utilize a four bit datapath and single read/write control wire saves enough pins to connect the CMM to the JTAG port on the 21464. We are looking to consolidating other functions (like GIO_CLK and SROM_CLK) as a way to further optimize the interface to the Marvel CMM.

16 Internal Processor Registers

16.1 Internal Processor Register Summary

See the Preface for the location of other documents that provide additional information about the 21464 internal processor registers.

Information can be read from and written into IPRs in various ways, as described in Section 2.12.1. Table 16–1 distinguishes registers that are explicitly written by an MTPR instruction, implicitly written as the result of executing an instruction, and implicitly written as a result of some event not directly associated with the execution of a specific instruction.

Not all IPRs can be read. To aid debug, it is important that those IPRs with a strong need to be readable be identified early.

The ability for one TPU to read or write another TPUs IPRs is still a source of debate. Currently no mechanism exists, but it is generally believed that debugging needs and error fix-up code might require this capability.

Table 16–1 Internal Processor Register Summary

Name	Mnemonic	Per-TPU	Index	Writer Class[1]	Read	Init[2]	Grp
Performance Monitoring IPRs[3]							
Event Counter IPR Bundle							
I General Events for TPU 0	IAGG_EVENT0	N	1 1100 000	E	Y	Dbg	—
I General Events for TPU 1	IAGG_EVENT1	N	1 1100 001	E	Y	Dbg	—
I General Events for TPU 2	IAGG_EVENT2	N	1 1100 010	E	Y	Dbg	—
I General Events for TPU 3	IAGG_EVENT3	N	1 1100 011	E	Y	Dbg	—
M Event Counter IPR Bundle							
M General Events for TPU 0	MAGG_EVENT0	N	0 1100 000	E	Y	Dbg	—
M General Events for TPU 1	MAGG_EVENT1	N	0 1100 001	E	Y	Dbg	—
M General Events for TPU 2	MAGG_EVENT2	N	0 1100 010	E	Y	Dbg	—
M General Events for TPU 3	MAGG_EVENT3	N	0 1100 011	E	Y	Dbg	—
Profile I Data IPR Bundle							
Profile Instruction Control	PR_INST_CTL	N	1 1101 000	M	Y	All	I1

Table 16–1 Internal Processor Register Summary (Continued)

Name	Mnemonic	Per-TPU	Index	Writer Class[1]	Read	Init[2]	Grp
Profile Trigger on PC	PR_TRIG_PC	N	1 1101 001	M	Y	All	I1
Profile Instruction Character.	PRn_PC	N	1 1101 01n	E	Y	Dbg	—
Profile Instruction Ibox Info	PR_I_INFO	N	1 1101 100	E	Y	Dbg	—
Profile Instruction Qbox Info	PR_Q_INFO	N	1 1101 101	E	Y	Dbg	—
Profile M Data IPR Bundle							
Profile MAGG_EVENT CTRL	PR_MEM_EVENT_CTL	N	0 1101 000	M	Y	All	M1
Profile Memory Information	PRn_MEM_INFO	N	0 1101 01n	E	Y	Dbg	—
Profile Store Latency Info	PR_ST_LATENCY	N	0 1101 100	E	Y	Dbg	—
PROFILE Timeline IPR Bundle							
Profile Instr Timeline part 0	PRn_TIMELINE0	N	1 1110 00n	E	Y	Dbg	—
Profile Instr Timeline part 1	PRn_TIMELINE1	N	1 1110 01n	E	Y	Dbg	—
Profile Instr Timeline part 2	PRn_TIMELINE2	N	1 1110 10n	E	Y	Dbg	—
Profile Instr Timeline part 3	PRn_TIMELINE3	N	1 1110 11n	E	Y	Dbg	—
Profile D Miss Bundle							
Profile Dcache Miss Info	PRn_DMISS_INFO	N	0 1110 00n	E	Y	Dbg	—
Ibox (Instruction Fetch Unit) IPRs							
Cycle Counter	CC	Y	1 0111 0XX	M	Y	*	I1
CPU Configuration	CPU_CNFG	N	1 1001 000	M	Y	All	I1
DTB Single-Miss Return Address	DTBMS_RET_ADDR	Y	1 0100 111	I	Y	Dbg	I1
Exception Address	EXC_ADDR	Y	1 0100 001	I	Y	Dbg	I1
Exception Summary	EXC_SUM	Y	1 0100 000	I	Y	Dbg	I1
Ibox Control	I_CTL	Y	1 0000 0XX	M	Y	Dbg	I1
Ibox Mode	I_MODE	Y	1 0001 0XX	M	Y	Dbg	I1
Ibox Process Context	I_PCTX	Y	1 0010 XXX	M	Y	Dbg	I1
Icache Status	IC_STAT	Y	1 1001 001	E	Y	Dbg	I1
Icache Flush	IC_FLUSH	Y	1 1001 100	M	N	—	—
Icache Flush (ASM=0)	IC_FLUSH_ASM	Y	1 1001 101	M	N	—	—
ITB Invalidate Multiple	ITB_IM	Y	1 1000 000	M	N	—	I2
ITB Invalidate Single	ITB_IS	Y	1 1000 010	M	N	No	I2
Instruction PTE Array Write	ITB_PTE	Y	1 1000 100	M	N	No	I3
Instruction TAG Array Write	ITB_TAG	Y	1 1000 110	M	N	No	I2
Inst. Virtual Address Format	IVA_FORM	Y	1 0100 011	—	Y	—	—
PALcode Base	PAL_BASE	Y	1 0101 000	M	Y	?	I1
CPU Base	CPU_BASE	N	1 0101 010	M	Y	All	I1

Table 16–1 Internal Processor Register Summary (Continued)

Name	Mnemonic	Per-TPU	Index	Writer Class[1]	Read	Init[2]	Grp
PALcode Temporary 1	PAL_TEMP1	Y	1 0101 001	M	Y	Dbg	I1
PALcode Temporary 2	PAL_TEMP2	Y	1 0101 010	M	Y	Dbg	I1
Thread Config	TPU_CNFG	Y	1 1011 000	M	Y	All	I1
Mbox (Internal Memory Controller Unit) IPRs							
Dcache Control	DC_CTL	N	0 1001 000	M	Y	All	M1
Dcache Status	DC_STAT	Y	0 1001 001	I	Y	Dbg	M1
DTB Invalidate Multiple	DTB_IM	Y	0 1000 000	M	N	—	M1
DTB Invalidate Single	DTB_IS	Y	0 1000 010	M	N	No	M2
DTB PTE Array Write	DTB_PTE	Y	0 1000 10X	M	N	No	M3
DTB Tag Array Write	DTB_TAG	Y	0 1000 11X	M	N	No	M2
Mbox Control	M_CTL	Y	0 0000 0XX	M	Y	Dbg	M1
Mbox Process Mode	M_MODE	Y	0 0001 XXX	M	Y	Dbg	M1
Mbox Process Context	M_PCTX	Y	0 0010 0XX	M	Y	Dbg	M1
Mbox Mem. Management Status	M_STAT	Y	0 0100 000	I	Y	Dbg	M1
Quiesce Timeout	QUIESCE_TIMEOUT	Y	0 0111 000	M	Y	Dbg	M1
Virtual Address	VA	Y	0 0100 001	I	Y	Dbg	M1
Virtual Address Format	VA_FORM	Y	0 0100 011	—	Y	—	—
Watch Physical Address	WATCH_PHYS_ADDR	Y	—	I		Dbg	—
Cbox (Scache Control) IPRs							
Hardware Interrupt Clear	HW_INT_CLR	Y		M	N		
Int. Enable and Current Mode	IER_CM	Y		M	Y		
Interrupt Summary	ISUM	Y					
Software Interrupt Request	SIRR	Y		M	Y		
Rbox (External Router Unit) IPRs							
Router Configuration 1	R_CFG1						
Router Configuration 2	R_CFG2						
Router Interrupt Mask	R_INT_MASK						
Router Interrupt Queue	R_INT_QUE						
Router Interrupt Queue Add	R_INT_QUEADD						
Router Interrupt Request	R_INT_REQ						
Router Interrupt Status	R_INT_STAT						
Router Interval Timer	R_INTER_TIM						
Router IO Port Buffer Size	R_IO_BUFSIZ						
Router IO Port Config 1	R_IO_CFG1						
Router IO Port Config 2	R_IO_CFG2						

Table 16–1 Internal Processor Register Summary (Continued)

Name	Mnemonic	Per-TPU	Index	Writer Class[1]	Read	Init[2]	Grp
Router IO Port Error Status	R_IO_ERR						
Router IO Port Perf. Counter	R_IO_PERF						
Router IO Port Timer1 Config	R_IO_T1CFG						
Router IO Port Timer2 Config	R_IO_T2CFG						
Router Local Port Error Status	R_LOC_ERR						
Router Channel*n* Config 1	R_*n*_CFG1						
Router Channel*n* Config 2	R_*n*_CFG2						
Router Channel*n* Error Status	R_*n*_ERR						
Router Channel*n* Perf Count	R_*n*_PERF						
Router Channel*n* Timer1 Config	R_*n*_T1CFG						
Router Channel*n* Timer2 Config	R_*n*_T2CFG						
Router Overall Timer-Control	R_OVER						
Router Routing Table	R_ROUT						
Router Scratch 1	R_SCRATCH1						
Router Scratch 2	R_SCRATCH2						
Router Who-Am-I?	R_WHOAMI						
Zbox (External Memory Controller Unit) IPRs							
DIFT Control	ZBOX*n*_DIFT_CTL						
DIFT Error Status	ZBOX*n*_DIFT_ERR_STATUS						
DIFT Timeout	ZBOX*n*_DIFT_TIMEOUT						
DRAM Calibration Control 1	ZBOX*n*_DRAM_CALIB_CTL1						
DRAM Calibration Control 2	ZBOX*n*_DRAM_CALIB_CTL2						
DRAM Error Address	ZBOX*n*_DRAM_ERR_ADR						
DRAM Error Status 1	ZBOX*n*_DRAM_ERR_STATUS1						
DRAM Error Status 2	ZBOX*n*_DRAM_ERR_STATUS2						
DRAM Error Status 3	ZBOX*n*_DRAM_ERR_STATUS3						
DRAM Error Control	ZBOX*n*_DRAM_ERROR_CTL						
DRAM Initialization Control	ZBOX*n*_DRAM_INIT_CTL						
DRAM Mapper Control	ZBOX*n*_DRAM_MAPPER_CTL						
DRAM Refresh Control	ZBOX*n*_DRAM_REFR_CTL						
DRAM Refresh Row	ZBOX*n*_DRAM_REFRESH_ROW						
DRAM Sweep Directory Bits	ZBOX*n*_DRAM_SWEEP_DIR						
DRAM Timing Control 1	ZBOX*n*_DRAM_TIMING_CTL1						
DRAM Timing Control 2	ZBOX*n*_DRAM_TIMING_CTL2						
DRAM Timing Control 3	ZBOX*n*_DRAM_TIMING_CTL3						

Table 16–1 Internal Processor Register Summary (Continued)

Name	Mnemonic	Per-TPU	Index	Writer Class[1]	Read	Init[2]	Grp
DRAM Timing Control 4	ZBOX*n*_DRAM_TIMING_CTL4						
Force Error Address	ZBOX*n*_FRC_ERR_ADR						
RAC Control	ZBOX*n*_RAC_CTL						
Performance Counter 1	ZBOX*n*_ZPM_CTL1						
Performance Counter 0	ZBOX*n*_ZPM_CTR0						

[1] M = MTPR; I = Implict; E = Event

[2] See Table 16–2.

[3] Chapter 19 contains the information for Performance Monitoring IPRs.

16.1.1 PALcode Coding Rules

PALcode coding rules are described in Section 17.5.

16.1.2 IPR Issues:

- The 21264 had the behavior that an implicitly written register would read as zero if read while being written. Will the 21464 have the same behavior? Should we define a valid bit in each of the implicitly-written registers to explicitly flag this case? All registers except VA have bit<63> available.
- Need to better understand SLEEP modes and GCLK PLL programming. This is also tied into how to bring the chip alive. What state must be preserved when entering/exiting sleep mode?
- What does I_CTL[CHIP_ID] do? If it cannot be written, how is it different than AMASK/IMPLVER?
- *Disruptions and PALmode in the Ibox* describes several cases where a combination of traps within traps overwrites implicitly written IPRs. Can these cases be enumerated to form guidelines or define a rule?

16.1.3 Reset

This section should be moved to the Init chapter when more known....

The 21464 will have at least three major reset modes (with maybe a fourth for manufacturing test):

1. Cold

   Power-on full-reset. Initialize all IPRs.

2. Fast

   Quick, complete reset for Tandem synchronization. Initialize all IPRs.

3. Debug

Programmable reset for debug. Initialize only required IPRs; the required subset being defined as IPRs that contain bits that could alter the initial post-reset code flow.

IPRs fall into two basic categories:

- Registers that must be set by hardware to an initial value for all reset flows.
- Registers that can be initialized by software (PALcode) during the flow.

The primary reason to NOT initialize an IPR is for debug. Manufacturing test patterns, Tandem synchronization, and general chip simulation and verification benefit from hardware initialization of most or all IPR values. In addition, implicitly written and event-written IPRs that are not also writable by a HW_MTPR can be difficult to initialize with software.

It should also be noted that if a destructive scan dump operation precedes a debug reset, the contents of all uninitialized IPRs are potentially unknown random values.

Table 16–2 defines the classes of initialization.

Table 16–2 IPR Initialization Classification

Class	Meaning
All	The value is initialized by hardware for all reset flows.
Dbg	The value is initialized by hardware for Cold and Fast reset flows but left to software to initialize for the debug reset flows.
No	The value is not initialized by hardware during any reset flow. The goal is to eliminate this class.

16.2 Ibox IPRs

This section describes the Ibox IPRs.

The IPR reserved fields can have the following type:

Table 16–3 IPR Reserved Field Type Definitions

Type	Meaning
MBZ	Must be zero when written and always read as zero.
RAZ	Ignored for writes and always read as zero.
X	Ignored for writes and reads.

16.2.1 Cycle Counter Register — CC[tpu]

The process cycle counter consists of two fields. The COUNT field is an unsigned, wrapping counter, the OFFSET field is an operating-system specific offset which, when added to the wrapping counter, forms a per-process or per-thread cycle count. The ENABLE field is used to enable/disable the counter.

The RPCC instruction is used to read the process cycle counter. It is TBD whether a MFPR instruction will also read the cycle counter.

Section (II–A) 2.1.12 of the Alpha SRM requires a mechanism to cause the RPCC instruction to read-as-zero, writing CC_CTL with a zero achieves that result.

Notes:

- Most event-written IPRs in the 21464 will have a valid bit because they may not read correctly when being updated by an event. This register must read correctly even if the counter is being incremented.

Read:	RPCC
Written	HW_MTPR
Index	0xB8-0xBB

The low two index bits allow for selective writing of fields.

00	write nothing
01	write OFFSET and ENABLE fields only
10	write COUNT field only
11	write all fields

Figure 16–1 Cycle Counter Register — CC[tpu]

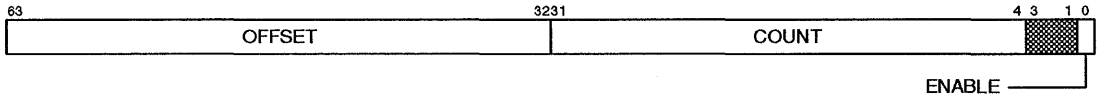


Table 16–4 Cycle Counter Register Fields Description

Field Name	Extent	Type	Description
OFFSET	63:32	RW,0	OS specific value that is added to PCC_CNT to derive the per-process cycle count. Overflow of PCC_CNT does not alter PCC_OFF.
COUNT	31:4	RW,0	Wrapping counter which increments once every 16th CPU cycle.
Reserved	3:1	RAZ	
ENABLE	0	RW,0	Cycle counter enable. The COUNT field increments monotonically when enabled and remains unchanged when disabled.

16.2.2 DTB Single-Miss Return Address Register - DTBMS_RET_ADDR[tpu]

Stores the return PC for single-level DTB miss traps. On the 21264, the return address was stored in EXC_ADDR, but saving the value in a separate register avoids cases where disruptions during the single miss flow cause the EXC_ADDR register to be modified and the return PC for the DTB miss flow to be lost.

The traps that set DTBMS_RET_ADDR are:

DTBMS_SINGLE

DTBMS_SINGLE_CONS

DTBMS_RET_ADDR is readable at two different locations. The first location (0xA6) is a general location with no side-effects. The second location (0xA7) has the side-effect of setting a issue block against load and store instructions and is intended to only be used within the block of instructions that modifies a DTB entry.

Written	Implicitly written when a DTB single miss trap occurs.
Readable	HW_MFPR
Index	0xA6, 0xA7
Reset	Unchanged for debug.

Figure 16–2 DTB Single-Miss Return Address Register — DTBMS_RET_ADDR[tpu]

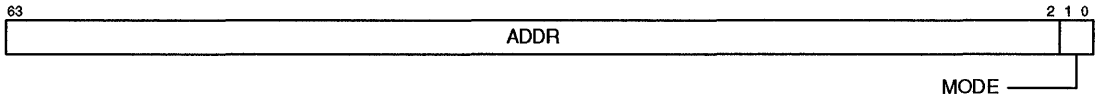


Table 16–5 DTB Miss Return Address Register Field Descriptions

Field Name	Extent	Type	Description
ADDR	63:2	IR,0	Sign-extended PC of the instruction that caused the TB miss, where ADDR is SEXT(PC<51:2>).
MODE	1:0	IR,0	Mode of the trapping instruction: 00 - Normal 01 - PALmode 11 - SuperPALmode

16.2.3 Exception Address Register — EXC_ADDR[tpu]

Implicitly written with the expected restart PC for most PALmode traps.

The only PALmode traps that do not write EXC_ADDR are:

DTBM_SINGLE

DTBM_SINGLE_CONS

IMCHK

For Interrupts, this register contains the PC of the next instruction that would have executed had the interrupt not occurred. PALcode uses this address as the return address from the interrupt handler.

Written	Implicitly written when a trap occurs.
Readable	HW_MFPR
Index	0xA1
Reset	Unchanged for debug.

Figure 16–3 Exception Address Register — EXC_ADDR[tpu]

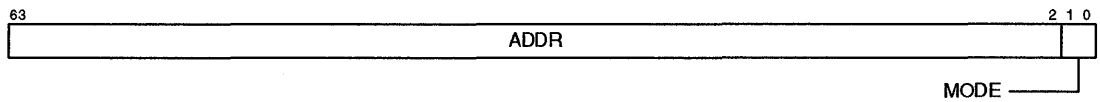


Table 16–6 Exception Address Register Field Descriptions

Field Name	Extent	Type	Description
ADDR	63:2	IR,0	For all traps except ARITH and MT_FPCR, the restart PC written into EXC_ADDR is the sign-extended PC, SEXT(PC<51:2>), of the instruction that caused the trap. For ARITH and MT_FPCR, the restart address is the PC of the next instruction
MODE	1:0	IR,0	Mode of the trapping instruction: 00 - Normal 01 - PALmode 11 - SuperPALmode

16.2.4 Exception Summary Register — EXC_SUM[tpu]

The exception summary register is an implicitly written register that contains trap status information and any register specifiers present in the original instruction.

The traps that set EXC_SUM are:

ARITH

DFAULT

UNALIGN

DTBM_SINGLE

DTBM_SINGLE_CONS

BAD_JMP_IVA

The register fields actually reflect bits <26:16> and <4:0> of the instruction longword independent of the type of operation. The fields are not qualified in any way for instructions that lack one or more of the register fields.

Written	Implicitly written when a trap occurs.
Readable	HW_MFPR
Index	0xA0
Reset	Unchanged for debug.

Figure 16–4 Exception Summary Register — EXC_SUM[tpu]

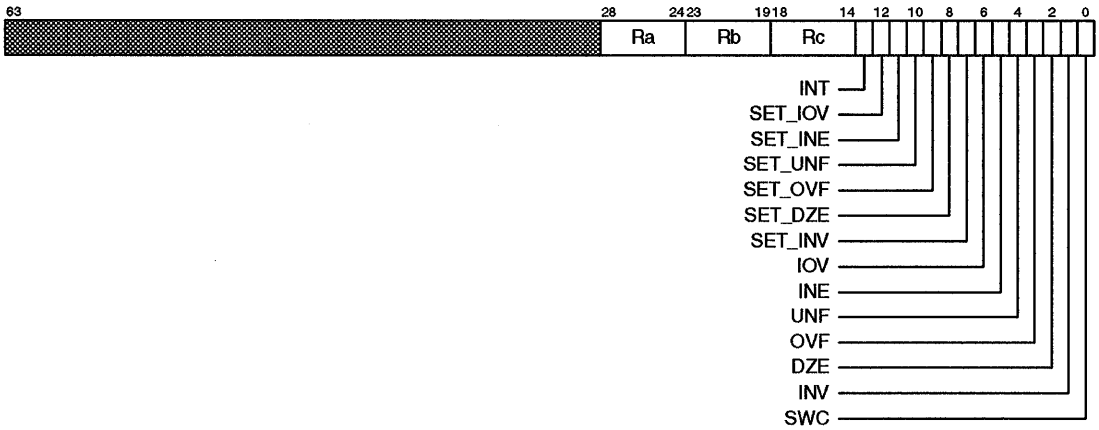


Table 16–7 Exception Summary Register Field Descriptions

Field Name	Extent	Type	Description
Reserved	63:29	RAZ	
Ra	28:24	IR,0	Instruction bits<25:21> of the trapping instruction
Rb	23:19	IR,0	Instruction bits<20:16> of the trapping instruction
Rc	18:14	IR,0	Instruction bits<4:0> of the trapping instruction
INT	13	IR,0	Integer overflow/underflow trap
SET_IOV	12	IR,0	PALcode should set FPCR[IOV]
SET_INE	11	IR,0	PALcode should set FPCR[INE]
SET_UNF	10	IR,0	PALcode should set FPCR[UNF]
SET_OVF	9	IR,0	PALcode should set FPCR[OVF]
SET_DZE	8	IR,0	PALcode should set FPCR[DZE]
SET_INV	7	IR,0	PALcode should set FPCR[INV]
IOV	6	IR,0	Floating convert to integer trap
INE	5	IR,0	Floating inexact error trap
UNF	4	IR,0	Floating underflow trap
OVF	3	IR,0	Floating overflow trap
DZE	2	IR,0	Divide by zero trap
INV	1	IR,0	Invalid operation trap
SWC	0	IR,0	Software completion possible/requested. Set if the instruction that triggered the trap contained the /S specifier.

16.2.5 Ibox CPU Configuration Register — CPU_CNFG

Per-chip configuration register. Settings apply to all TPUs.

Written	HW_MTPR
Readable	HW_MFPR
Index	0xC8
Reset	All modes.

Figure 16–5 Ibox CPU Configuration Register — CPU_CNFG

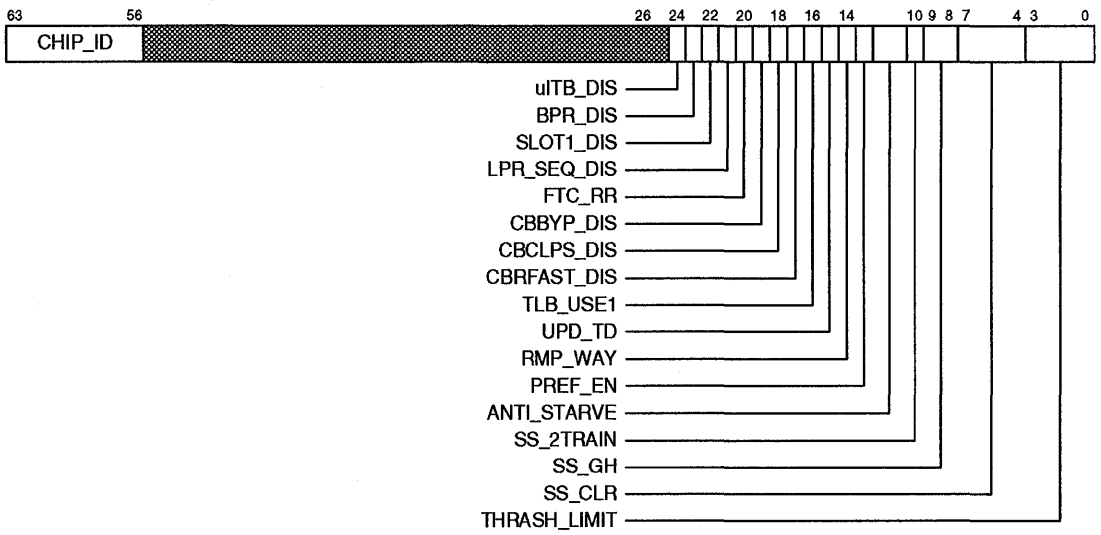


Table 16–8 CPU Configuration Register Fields Description

Field Name	Extent	Type	Description
CHIP_ID	63:56	RO,0	Read-only CHIP_ID code
Reserved	55:25	MBZ	
µITB_DIS	24	RW,0	Disable the µITB performance feature (a debug mode).
BPR_DIS	23	RW,0	Disable the branch predictor. When set, all branches will be predicted not-taken.
SLOT1_DIS	22	RW,0	Disable the use of slot 1 fetching.
LPR_SEQ_DIS	21	RW,0	Disable sequential training, predict non-sequential
FTC_RR	20	RW,0	Force the fetch thread chooser into round-robin mode
CBBYP_DIS	19	RW,0	Disable bypasses around the collapsing instruction buffer
CBCLPS_DIS	18	RW,0	Disable the collapsing capability of the collapsing buffer
CBRFAST_DIS	17	RW,0	Disable Oldest CBR mispredict fast restart optimization
TLB_USE1	16	RW,0	Use only one entry in the ITB.
UPD_TD	15	RW,0	Update the thrash detector array
RMP_WAY	14	RW,0	Enable remapping the Icache way when a thrash has been detected by the thrash detector.

Table 16–8 CPU Configuration Register Fields Description (Continued)

Field Name	Extent	Type	Description
PREF_EN	13	RW,0	Enable the prefetch hardware.
ANTI_STARVE	12:11	RW,0	Controls the fetch starvation threshold detection. If a TPU does not retire an instruction for the selected number of instructions, the other TPUs will be suspended until the starving threads have retired at least one good instruction. 00 Off (anti-starvation detection disabled) 01 1K cycle non-retire threshold 10 16K cycle non-retire threshold 11 128K cycle non-retire threshold
SS_2TRAIN	10	RW,0	
SS_GH	9:8	RW,0	By enabling these bits, one or both of the upper two bits of the Store Set array index will include LGHIST information.
SS_CLR	7:4	RW,0	The Store Set array is cleared whenever the bit defined by this field overflows in a free-running counter. The bit monitored is bit (9 + SS_CLR) creating a clear frequency of $2^{(9+SS\_CLR)}$. A SS_CLR value of zero disables the Store Set array. The recommended value of SS_CLR is ten causing the Store Sets to be cleared every 512K cycles.
THRASH_LIMIT	3:0	RW,0	Number of thrashes before the entry is remapped to a set location in the Icache.

16.2.6 Ibox TPU Configuration Register — TPU_CNFG

Icache/Ibox configuration register.

Written	HW_MTPR
Readable	HW_MFPR
Index	0xD8
Reset	All modes.

Figure 16–6 Ibox TPU Configuration Register — TPU_CNFG

Table 16–9 Ibox TPU Configuration Register Field Descriptions

Field Name	Extent	Type	Description
Reserved	63:58	MBZ	
TPU_ID	57:56	RO,0-3	Read-only ID of the current TPU. The 21464 has four TPUs numbered 0,1,2,3.
Reserved	55:8	MBZ	

Table 16–9 Ibox TPU Configuration Register Field Descriptions

Field Name	Extent	Type	Description
MCHK_EN	7	RW,0	Enable machine-check interrupts to this TPU.
PREF_RANGE	6:4	RW,0	Number of Icache blocks to prefetch beyond a demand miss
Reserved	3:0	MBZ	

16.2.7 Ibox Control Register — I_CTL[tpu]

The per-TPU I_CTL register controls Istream memory management functions.

Most fields in I_CTL are replicated in M_CTL. It is expected (but not required) that these registers will be typically written together.

Written	HW_MTPR
Readable	HW_MFPR
Index	0x80-0x83
Reset	Unchanged for debug.

The low two index bits allow for selective writing of fields.

00	Write nothing
01	Write VPTE_BASE only
10	Write SUPERPAGE, VA_SIZE and PAGE_SIZE fields
11	Write all fields

Figure 16–7 Ibox Control Register — I_CTL[tpu]

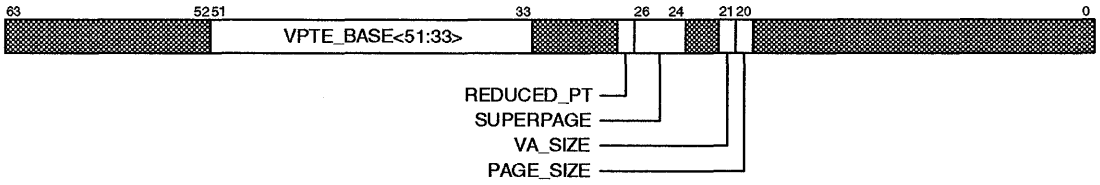


Table 16–10 Ibox Control Register Field Descriptions

Field Name	Extent	Type	Description
Reserved	63:52	MBZ	
VPTE_BASE	51:33	RW,0	Virtual Page Table Base. See IVA_FORM, Section 16.2.17, for details.
Reserved	32:28	MBZ	
REDUCED_PT	27	RW	See Appendix C.3.

Table 16–10 Ibox Control Register Field Descriptions

Field Name	Extent	Type	Description
SUPERPAGE	26:24	RW,0	Istream Super Page mode enables. Any combination of bits can be set at once. Any non-kernel mode access to an enabled superpage region must result in an access violation. SPE[2] Enables super page mapping when PC[63:50] = 0x3FFE. In this mode PC[47:0] is mapped directly to PA[47:0]. SPE[1] Enables super page mapping when PC[63:41] = 0x7FFFFE. In this mode PA[47:0] = SEXT(PC[40:0]). SPE[0] Enables super page mapping when PC[63:30] = 0x3FFFFFFFE. In this mode PA[47:0] = ZEXT(PC[29:0]).
Reserved	23:22	RAZ	
VA_SIZE	21	RW,0	Defines the I-Stream Virtual address size. Controls the IVA_FORM register and sign extension checking. VA_SIZE = 0 — 43-bit addressing VA_SIZE = 1 — 52-bit addressing (invalid if PAGE_SIZE = 0)
PAGE_SIZE	20	RW,0	Defines the I-Stream page size. Controls the IVA_FORM register PAGE_SIZE = 0 — 8KB pages PAGE_SIZE = 1 — 64KB pages
Reserved	19:0	As follows:	

Bits	Type
19	X
18	MBZ
17:8	X
7:6	RAZ
5:3	X
2:0	MBZ

16.2.8 Ibox Process Mode Register — I_MODE[tpu]

The Ibox process mode register specifies the console and current process mode. These mode bits shadow the bits in M_MODE and exists on a per-TPU basis.

Written	HW_MTPR
Read	HW_MFPR
Index	0x88-0x8B
Reset	Unchanged for debug.

The low two index bits allow for selective writing of fields.

00	Write nothing
01	Write CURRENT field only
10	Write CONSOLE field only
11	Write all fields

Figure 16–8 Ibox Process Mode Register — I_MODE[tpu]

Table 16–11 Ibox Process Mode Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:6	As follows:	
		Bits	**Type**
		63:52	MBZ
		51:33	X
		32:28	MBZ
		27:19	X
		18	MBZ
		17:6	X
CONSOLE	5	RW,0	ITB traps in console mode are reported to the trap handler separately from non-console mode traps so they can be vectored to different addresses.
CURRENT	4:3	RW,0	The CURRENT mode field is encoded as follows 00 — Kernel 01 — Executive 10 — Supervisor 11 — User
Reserved	2:0	MBZ	

16.2.9 Ibox Process Context Register — I_PCTX[tpu]

The process context register contains information associated with the context of the process currently running on the TPU.

Written	HW_MTPR
Readable	HW_MFPR
Index	0x90-0x97
Reset	Unchanged for debug.

The low three index bits allow for selective writing of fields.

000	Write nothing
001	Write ASN field only
010	Write TPU_GRP field only
100	Write FP_ENABLE field only
111	Write all fields

Figure 16–9 Ibox Process Context Register — I_PCTX[tpu]

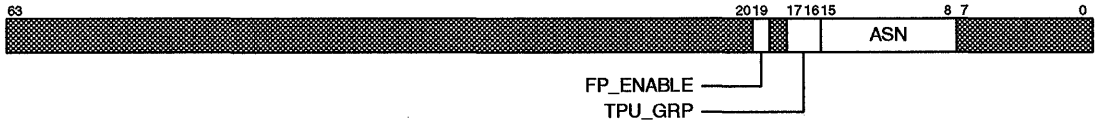


Table 16–12 Ibox Process Context Register Field Descriptions

Field Name	Extent	Type	Description
Reserved	63:20	As follows:	
		Bits	**Type**
		63:52	MBZ
		51:33	X
		32:28	MBZ
		27:20	X
FP_ENABLE	19	RW,0	If clear, floating-point instructions generate FEN exceptions. Used at process context switch time to detect if any FP state exists. Software clears the bit when a process is initially created and only sets it when the first FP instruction traps.
Reserved	18	MBZ	
TPU_GRP	17:16	RW,0	TPU group number associated with this TPU. Allows TB entries to be collectively allocated/invalidated for all TPUs that belong to the same group.
ASN	15:8	RW,0	Address space number. Stored in TBs and compared during invalidate operations to minimize the number of entries invalidated on a process context switch. See SRM Section (II-B) 3.8 for details.
Reserved	7:0	As follows:	
		Bits	**Type**
		7:3	X
		2:0	MBZ

16.2.10 Icache Status Register — IC_STAT[tpu]

The Ibox status register is a read/write-1-to-clear register that contains Ibox status information.

Written	Set implicitly by Icache data or tag parity error, W1C
Readable	HW_MFPR
Index	0xC9
Reset	Unchanged for debug.

Figure 16–10 Icache Status Register — IC_STAT[tpu]

Table 16–13 Icache Status Register Fields Descriptions

Field Name	Extent	Type	Description
Reserved	63:16	RAZ	
INDEX	15:5	RO,0	Cache line index that caused the PAR_ERROR bit to be set.
Reserved	4:2	RAZ	
MULTIPLE	1	RO,0	Set when a data or tag parity error occurs and the PAR_ENABLE bit is already set. Cleared whenever the PAR_ERROR bit is cleared.
PAR_ERROR	0	W1C,0	Set when a data or tag parity error occurs, Cleared when wtten with a 1. Writing a 1 also clears the MULTIPLE field.

16.2.11 Icache Flush Register — IC_FLUSH[tpu]

When a write to the IC_FLUSH pseudo-register retires, all Icache blocks that match the group number of this TPU (in I_PCTX) are marked invalid.

Written	HW_MTPR
Readable	No
Index	0xCC
Reset	N/A

Figure 16–11 Icache Flush Register — IC_FLUSH[tpu]

Table 16–14 Icache Flush Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:0	X	

16.2.12 Icache Flush (ASM=0) Register — IC_FLUSH_ASM[tpu]

When a write to the IC_FLUSH_ASM pseudo-register retires, all Icache blocks that match the group number of this TPU (in I_PCTX) and have their ASM bit cleared are marked invalid.

The current implementation actually performs an IC_FLUSH when this IPR is written.

Written	HW_MTPR
Readable	No
Index	0xCD
Reset	N/A

Figure 16–12 Icache Flush (ASM = 0) Register — IC_FLUSH_ASM[tpu]

Table 16–15 Icache Flush (ASM = 0) Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:0	X	

16.2.13 ITB Invalidate Multiple Register — ITB_IM[tpu]

The ITB Invalidate Multiple register is a write-only pseudo-register. When write instructions to this register retire, all ITB entries matching the criteria specified by the mode bits are invalidated. An explicit write to IC_FLUSH is required to flush the Icache of the corresponding blocks.

Written	HW_MTPR
Readable	NO
Index	0xC0
Reset	N/A

Figure 16–13 ITB Invalidate Multiple Register — ITB_IM[tpu]

Table 16–16 ITB Invalidate Multiple Register Fields Descriptions

Field Name	Extent	Type	Description
Reserved	63:5	MBZ	
MODE	4:0	WO,0	Defines the invalidation mode, as follows:

Value	Mnemonic	Description
0x00	IAG	Invalidate all entries independent of group
0x01	IA	Invalidate all entries that match the current TPUGRP field in M_PCTX.
0x03	IASM	Invalidate all entries with the ASM bit set that also match the current TPUGRP.
0x05	IAP	Invalidate all entries with the ASM bit clear that also match the current TPUGRP
0x0D	IASN	Invalidate all non-ASM entries that match the current TPUGRP and ASN fields.
0x10	IAG	Invalidate all entries independent of group and reset the write pointer
0x16	IWRP	Reset the write pointer only. Nothing is invalidated

16.2.14 ITB Invalidate Single Register — ITB_IS[tpu]

When a write to the ITB_IS pseudo-register retires, all ITB entries that match the group number and address space number of this TPU (in I_PCTX) and match the tag value supplied are marked invalid.

The implementation physically shares storage for this register with the ITB_TAG register so writes to these registers must be separated by a IFETCHB instruction to ensure correct ordering.

Written	HW_MTPR
Readable	No
Index	0xC2
Reset	No

Figure 16–14 ITB Invalidate Single Register — ITB_IS[tpu]

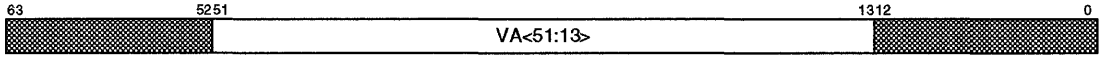


Table 16–17 ITB Invalidate Single Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:52	MBZ	
VA<51:13>	51:13		
Reserved	12:0	MBZ	

16.2.15 Instruction PTE Array Write Register — ITB_PTE[tpu]

The ITB PTE array is written by way of this register. A write transaction to the ITB_TAG writes a register outside of the ITB array. When a write to the ITB_PTE register is retired, the contents of both the ITB_TAG and ITB_PTE registers are written into the ITB entry. The specific ITB entry written is determined by the round-robin algorithm described above.

Written	HW_MTPR
Readable	No
Index	0xC4
Reset	No

Figure 16–15 Instruction PTE Array Write Register — ITB_PTE[tpu]

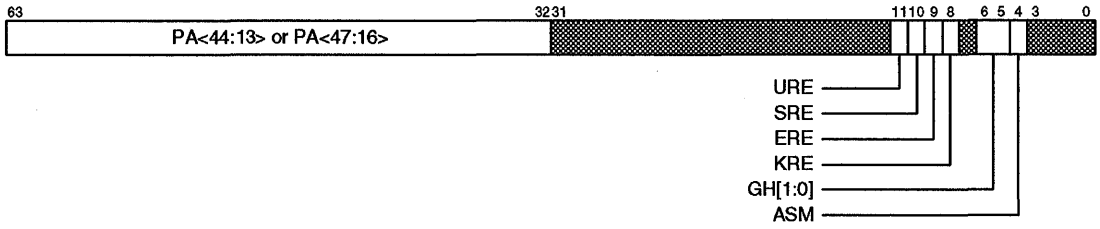


Table 16–18 Instruction PTE Array Write Register Field Descriptions

Field Name	Extent	Type	Description
PA<44:13> or PA<47:16>	63:32	W,?	The physical page number is PA<44:13> when in 8K page mode and PA<47:16> when in 64K page mode. The 21464 cannot address more than 16TB when in 8K page mode.
Reserved	31:12	MBZ	
URE	11	W,?	User write enable. When process context is User mode, this bit must be set to write this entry.
SRE	10	W,?	Supervisor write enable. When process context is Supervisor mode, this bit must be set to write this entry.
ERE	9	W,?	Executive write enable. When process context is Executive mode, this bit must be set to write this entry.
KRE	8	W,?	Kernel write enable. When process context is Kernel mode, this bit must be set to write this entry.
Reserved	7	MBZ	
GH[1:0]	6:5	W,?	Granularity hint.
ASM	4	W,?	Address space match bit. When set, this PTE matches all address space numbers.
Reserved	3:0	MBZ	

16.2.16 Instruction Tag Array Write Register — ITB_TAG[tpu]

The ITB tag array is written by way of this register. A write transaction to the ITB_TAG writes a register outside of the ITB array. When a write to the ITB_PTE register is retired, the contents of both the ITB_TAG and ITB_PTE registers are written into the ITB entry. The specific ITB entry written is determined by a round-robin algorithm; the algorithm writes to entry number 0 as the first entry after the 21464 is reset.

The implementation shares the physical register with the ITB_IS register so a IFETCHB instruction must separate writes to the ITB_TAG and ITB_IS registers.

Written	HW_MTPR
Readable	No
Index	0xC6
Reset	No

Figure 16–16 Instruction Tag Array Write Register — ITB_TAG[tpu]

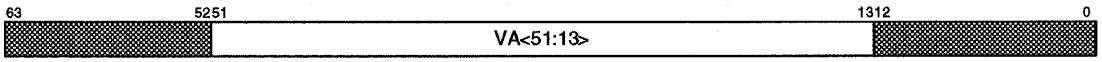


Table 16–19 Instruction Tag Array Write Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:52	MBZ	
VA<51:13>	51:13		
Reserved	12:0	MBZ	

16.2.17 Instruction Virtual Address Format Register — IVA_FORM[tpu]

The read-only virtual address format register contains the virtual page table entry address derived from the faulting virtual address stored in the EXC_ADDR register along with the virtual page table base and associated control bits stored in the I_CTL register.

Written	N/A (Derived from other implicitly and explicitly written registers)
Readable	HW_MFPR
Index	0xA3
Reset	N/A

Figure 16–17 Instruction Virtual Address Format Register — IVA_FORM[tpu]

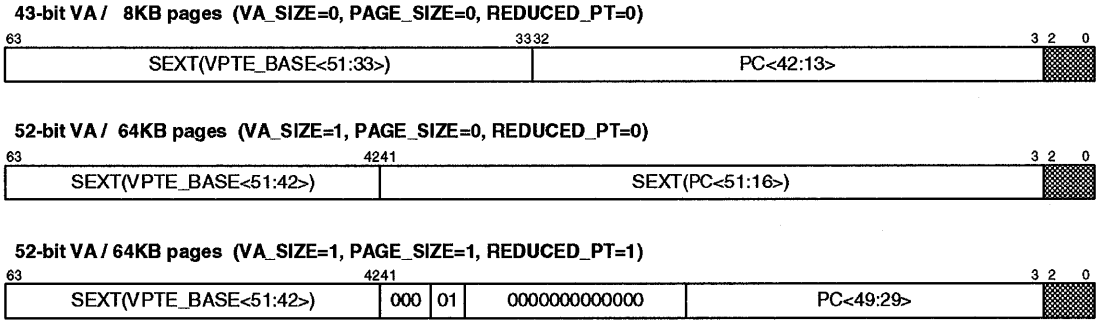


Table 16–20 Instruction VA Format Register (43-Bit VA) Fields Description

Field Name	Extent	Type	Description
SEXT(VPTE_BASE<51:33>)	63:33		
PC<42:13>	32:3		
Reserved	2:0	RAZ	—

Table 16–21 Instruction VA Format Register (52-Bit VA, REDUCED-PT=0) Fields Description

Field Name	Extent	Type	Description
SEXT(VPTE_BASE<51:42>)	63:42		
SEXT(PC<51:16>)	41:3		
Reserved	2:0	RAZ	—

Table 16–22 Instruction VA Format Register (52-Bit VA, REDUCED-PT=1) Fields Description

Field Name	Extent	Type	Description
SEXT(VPTE_BASE<51:42>)	63:42		
	41:39		
	38:37		
	36:24		
PC<49:29>	23:3		
Reserved	2:0	RAZ	—

16.2.18 PALcode Base Address Register - PAL_BASE[tpu]

Based on the type of fault, the hardware vectors into the appropriate PALcode handler as an offset from the physical address in PAL_BASE.

The specific PALcode entry points and offsets are:

Table 16–23 PALcode Base Address Entry Points and Offsets

RESERVED	PB + x000
DTBM_DOUBLE	PB + x100
DTBM_DOUBLE_ALT	PB + x180
FEN	PB + x200
UNALIGN	PB + x280
DTBM_SINGLE	PB + x300
DFAULT	PB + x380
OPCDEC	PB + x400
IACV	PB + x480
MCHK	PB + x500
ITB_MISS	PB + x580
ARITH	PB + x600
INTERRUPT	PB + x680
MT_FPCR	PB + x700
IMCHK	PB + x780
DTBM_SINGLE_CONS	PB + x800
ITB_MISS_CONS	PB + x880

Table 16–23 PALcode Base Address Entry Points and Offsets

RESERVED	PB + x000
BAD_JUMP_IVA	PB + x900
FAULT_RESET	PB + x980
WAKEUP	PB + xA00
IP_RESET	PB + xA80
RESET	PB + xB00

PAL_BASE is also used in the computation of CALL_PAL branches. The 21464 computes the target PC of a CALL_PAL instruction as follows:

Bits	Contents
PC<51:15>	PAL_BASE<51:15>
PC<14>	0
PC<13>	1
PC<12:12>	CallPal function<7>
PC<11:6>	CallPal function<5:0>
PC<5:2>	0
PC<1>	Current PC<1>
PC<0>	1

The SRM does not actually define the behavior of PAL_BASE<63:52>, but reading anything other than zero feels unwise.

Written	HW_MTPR
Readable	HW_MFPR, Implicitly by trap to PALmode.
Index	0xA8

Figure 16–18 PALcode Base Address Register — PAL_BASE[tpu]

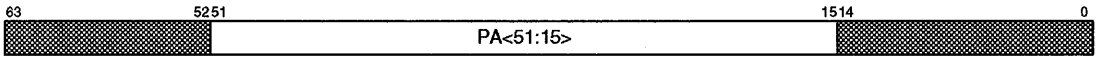


Table 16–24 PALcode Base Address Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:52	MBZ	
VA<51:15>	51:15		
Reserved	14:0	MBZ	

16.2.19 PALcode Temp Registers — PAL_TEMP1[tpu], PAL_TEMP2[tpu]

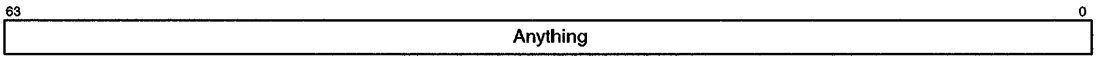
The PAL_TEMP registers are for miscellaneous use by PALcode. The primary intention is not to use these registers for save/restore type sequences within normal PALcode flows, but as infrequently written holders of important state.

The 21464 does not define a specific use for these registers but discussed uses include:

- Hold the physical address of a scratch area of memory where this TPU can save and restore values. During the PALcode initialization sequence, each CPU/TPU would do a calculation to produce a unique pal_temp_address. On previous processors, this value would have been stored in a PALcode shadow register, but given the lack of shadow registers in the 21464, this might be a good use for the scratch registers.
- Use during PALcode flows, where the ability to reliably access memory is questionable. In this case, HW_ST/HW_LD would not be an option so the overhead of synchronizing writes and reads to these registers is reasonable.

Written	HW_MTPR
Readable	HW_MFPR
Index	0xA9, 0xAA

Figure 16–19 PALcode Temp Registers — PAL_TEMP1[tpu], PAL_TEMP2[tpu]



16.3 Mbox IPRs

16.3.1 Dcache Control Register — DC_CTL

The Dcache control register is a chip-wide register controlling Dcache state.

There are many open issues relating to the structure of this register. One new bit, when written with a one, will reset the DTB write pointer to zero. Is this the flush bit?

Written	HW_MTPR
Read:	HW_MFPR
Index	0x48

Figure 16–20 Dcache Control Register — DC_CTL

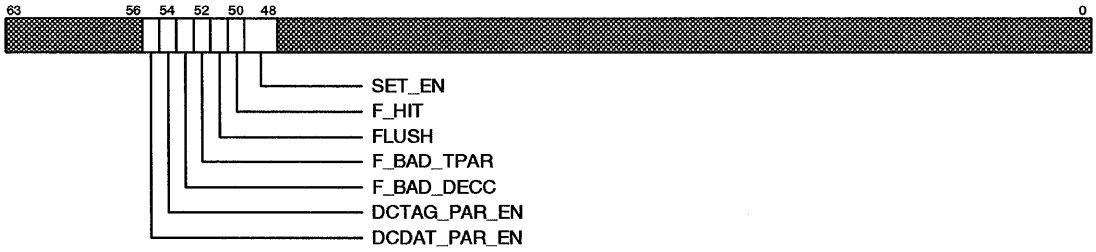


Table 16–25 Dcache Control Register Field Descriptions

Field Name	Extent	Type	Description
Reserved	63:56	MBZ	
DCDAT_PAR_EN	55	RW,0	Dcache data parity error enable
DCTAG_PAR_EN	54	RW,0	Dcache tag parity enable

Table 16–25 Dcache Control Register Field Descriptions (Continued)

Field Name	Extent	Type	Description
F_BAD_DECC	53	RW,0	Force Bad Data ECC. When set, ECC data is not written into the cache along with the block that is loaded by the fill or store.
F_BAD_TPAR	52	RW,0	Force Bad Tag Parity. When set this bit cause bad tag parity to be put in the Dcache tag array during Dcache fill operations
FLUSH	51	RW,0	
F_HIT	50	RW,0	Force Hit. When set, this bit causes all memory space load and store instructions to hit in the Dcache, independent of the Dcache tag address compare.
SET_EN	49:48	RW,0	Dcache Set Enable. At least one set must be enabled.
Reserved	47:0	MBZ	

16.3.2 Dcache Status Register — DC_STAT[tpu]

The Dcache status register is a per-TPU read-write register containing information about Dcache parity and ECC errors.

The status bits indicate an error when set and must be explicitly written with a 1 to clear.

Written	Set when the parity or ECC error event occurs.
Read	HW_MFPR
Index	0x49
Reset	Unchanged for debug modes.

Figure 16–21 Dcache Status Register — DC_STAT[tpu]

Table 16–26 Dcache Status Register Field Descriptions

Field Name	Extent	Type	Description
SEO	63	W1C,0	A second ECC error occurred within N cycles of a previous ECC error.
ECC_ERR_ST	62	W1C,0	An ECC error occurred when processing a store
ECC_ERR_LD	61	W1C,0	An ECC error occurred when processing a load from the Dcache or any fill

Table 16–26 Dcache Status Register Field Descriptions (Continued)

Field Name	Extent	Type	Description
TPERR_P2	60	W1C,0	A Dcache tag probe from pipe 2 resulted in a tag parity error. The error is uncorrectable.
TPERR_P1	59	W1C,0	A Dcache tag probe from pipe 1 resulted in a tag parity error. The error is uncorrectable.
TPERR_P0	58	W1C,0	A Dcache tag probe from pipe 0 resulted in a tag parity error. The error is uncorrectable.
Reserved	57:0	RAZ	

Figure 16–22 DTB Invalidate Address Space Register — DTB_IASN[tpu]

16.3.3 DTB Invalidate Multiple Register — DTB_IM[tpu]

The DTB Invalidate Multiple register is a write-only pseudo-register. When write instructions to this register retire, all DTB entries matching the criteria specified by the mode bits are invalidated.

Written	HW_MTPR
Readable	No
Index	0x40

Figure 16–23 DTB Invalidate Multiple Register — DTB_IM[tpu]

Table 16–27 DTB Invalidate Multiple Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:5	MBZ	
MODE	4:0	WO,0	Defines the invalidation mode, as follows:

Value	Mnemonic	Description
0x00	IAG	Invalidate all entries independent of group
0x01	IA	Invalidate all entries that match the current TPUGRP field in M_PCTX.
0x03	IASM	Invalidate all entries with the ASM bit set that also match the current TPUGRP.
0x05	IAP	Invalidate all entries with the ASM bit clear that also match the current TPUGRP
0x0D	IASN	Invalidate all non-ASM entries that match the current TPUGRP and ASN fields.
0x10	IAG	Invalidate all entries independent of group and reset the write pointer
0x16	IWRP	Reset the write pointer only. Nothing is invalidated

16.3.4 DTB Invalidate Single Register — DTB_IS[tpu]

The DTB Invalidate Single register is a write-only pseudo-register. Write instructions to this register invalidate any DTB entries that would match the virtual page number written, given the current values of the TPUGRP and DTB_ASN registers.

Written	HW_MTPR
Readable	No
Index	0x42

Figure 16–24 DTB Invalidate Single Register — DTB_IS[tpu]

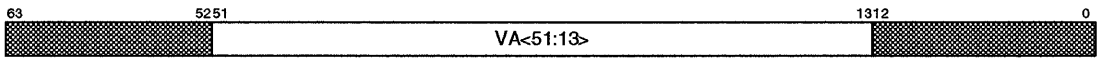


Table 16–28 DTB Invalidate Single Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:52	MBZ	
VA<51:13>	51:13		
Reserved	12:0	MBZ	

16.3.5 DTB PTE Array Write Registers — DTB_PTE0[tpu], DTB_PTE1[tpu]

The DTB PTE write register is a write-only register used to write the PTE part of the DTB array. It contains the physical page mapping and protection bits for the array. When a write to it retires, the PTE, along with the DTB_TAG, DTB_ASN and TPUGRP registers, are written to the DTB. PALcode must perform two consecutive writes to DTB_PTE to guarantee both copies of the TB are updated together.

The bits in this register are also specified in Section (II–A) 3.6 of the Alpha SRM.

Written	HW_MTPR
Readable	No
Index	0x44

Figure 16–25 DTB PTE Array Write Registers — DTB_PTE0[tpu], DTB_PTE1[tpu]

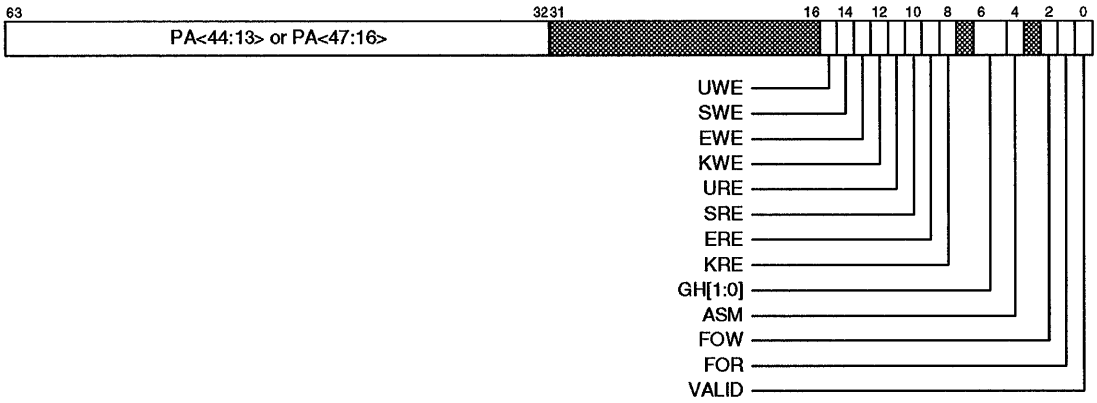


Table 16–29 DTB_PTE Array Write Registers Fields Descriptions

Field Name	Extent	Type	Meaning
PA<44:13> or PA<47:16>	63:32	WO, 0	The physical page number is PA<44:13> when in 8K page mode and PA<47:16> when in 64K page mode. The 21464 cannot address more than 16 TB when in 8K page mode.
Reserved	31:16	MBZ	
UWE	15	WO, 0	User write enable. When process context is User mode, this bit must be set to write this entry.
SWE	14	WO, 0	Supervisor write enable. When process context is Supervisor mode, this bit must be set to write this entry.
EWE	13	WO, 0	Executive write enable. When process context is Executive mode, this bit must be set to write this entry.
KWE	12	WO, 0	Kernel write enable. When process context is Kernel mode, this bit must be set to write this entry.
URE	11	WO, 0	User read enable. When process context is User mode, this bit must be set to read this entry.

Table 16–29 DTB_PTE Array Write Registers Fields Descriptions

Field Name	Extent	Type	Meaning
SRE	10	WO, 0	Supervisor read enable. When process context is Supervisor mode, this bit must be set to read this entry.
ERE	9	WO, 0	Executive read enable. When process context is Executive mode, this bit must be set to read this entry.
KRE	8	WO, 0	Kernel read enable. When process context is Kernel mode, this bit must be set to read this entry.
Reserved	7	MBZ	
GH[1:0]	6:5	WO, 0	Granularity Hint.
ASM	4	WO, 0	Address space match bit. When set, this PTE matches all address space numbers.
Reserved	3	MBZ	
FOW	2	WO, 0	Fault-on-write control bit.
FOR	1	WO, 0	Fault-on-read control bit
VALID	0	WO, 0	Valid bit. Not actually written to the TB, but is used to prevent the writer block interlock from being lifted unitl the MTPR instruction is killed. The DTB miss PALcode must include a branch-on-invalid check before the DTB write or the 21464 hangs if an invalid TB entry is written.

16.3.6 DTB Tag Array Write Registers — DTB_TAG0[tpu], DTB_TAG1[tpu]

The DTB Tag write register is a write-only register used to write the DTB tag array. It contains the virtual page number of the entry currently being written to the DTB, and will be committed to the DTB array when the corresponding write to DTB_PTE retires. The DTB_ASN and TPUGRP registers are also implicitly included in the data written to the DTB when the write to DTB_PTE retires. PALcode must perform two consecutive writes to DTB_PTE to guarantee both copies of the TB are updated together.

Written	HW_MTPR
Readable	No
Index	0x46

Figure 16–26 DTB Tag Array Write Registers — DTB_TAG0[tpu], DTB_TAG1[tpu]

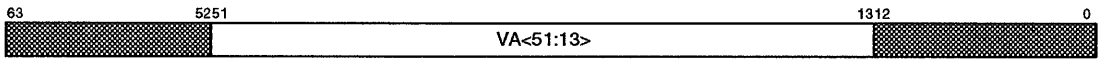


Table 16–30 DTB Tag Array Write Registers Fields Description

Field Name	Extent	Type	Description
Reserved	63:52	MBZ	
VA<51:13>	51:13		
Reserved	12:0	MBZ	

16.3.7 Mbox Control Register — M_CTL[tpu]

The Mbox control register was a write-only register in the 21264, but is proposed to be read-write in the 21464. M_CTL controls the formatting of the VA_FORM register by storing the virtual page table base, along with control bits for big-endian and 64K page modes.

Written	HW_MTPR
Read	HW_MFPR
Index	0x00-0x03

The low two index bits allow for selective writing of fields.

00	Write nothing
01	Write VPTE_BASE field only
10	Write SUPERPAGE, BIG_ENDIAN, VA_SIZE and PAGE_SIZE only
11	Write all fields

Figure 16–27 Mbox Control Register — M_CTL[tpu]

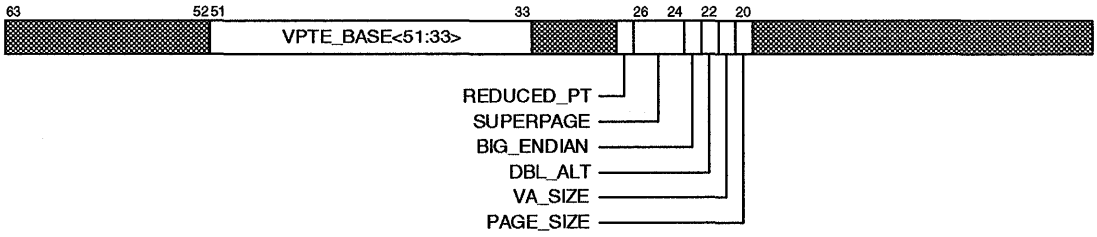


Table 16–31 Mbox Control Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:52	MBZ	
VPTE_BASE<51:33>	51:33	RW,0	Virtual Page Table Base. See the VA_FORM register section for details.
Reserved	32:28	MBZ	
REDUCED_PT	27		

Table 16–31 Mbox Control Register Fields Description

Field Name	Extent	Type	Description
SUPERPAGE	26:24	RW,0	Dstream Super Page mode enables. Any combination of bits can be set at once. Any non-kernel mode access to an enabled superpage region must result in an access violation. **Bits** **Meaning** SPE[2] Enables super page mapping when VA[63:50] = 0x3FFE. In this mode VA[47:0] is mapped directly to PA[47:0]. SPE[1] Enables super page mapping when VA[63:41] = 0x7FFFFE. In this mode PA[47:0] = SEXT(VA[40:0]). SPE[0] Enables super page mapping when VA[63:30] = 0x3FFFFFFFE. In this mode PA[47:0] = ZEXT(VA[29:0]).
BIG_ENDIAN	23	RW,0	When set, the lower bits of the physical address for Dstream accesses are inverted based upon the length of the datatype referenced. Also, the shift amount (Rbv[2:0]) is inverted for EXTxx, INSxx and MSKxx instructions.
DBL_ALT	22	RW,0	Determines which double miss flow will be vectored to when a hw_ld/vpte misses in the TB. This bit controls the vectoring for all double TB misses -- I-Stream and D-Stream. 0 Vector to DTB_MISS_DOUBLE 1 Vector to DTB_MISS_DOUBLE_ALT DTB_MISS_DOUBLE and DTB_MISS_DOUBLE_ALT are in used in place of the 21264's DTB_MISS_DOUBLE_3 and DTB_MISS_DOUBLE_4 with the distinction being that the decision is the discretion of PALcode.
VA_SIZE	21	RW,0	Defines the D-Stream Virtual address size. Controls the VA_FORM register and sign extension checking. VA_SIZE = 0 specifies 43-bit addressing VA_SIZE = 1 specifies 52-bit addressing (invalid if PAGE_SIZE = 0)
PAGE_SIZE	20	RW,0	Defines the D-Stream page size. Controls the VA_FORM register. PAGE_SIZE = 0 specifies 8KB pages PAGE_SIZE = 1 specifies 64KB pages
Reserved	19:0	As follows:	**Bits** **Type** 19 X 18 MBZ 17:3 X 2:0 MBZ

16.3.8 Mbox Process Mode Register — M_MODE[tpu]

The Mbox process mode register specifies the console, current and alternate processor mode. These mode bits shadow the bits in I_MODE and exists on a per-TPU basis.

Written	HW_MTPR
Read	HW_MFPR
Index	0x08-0x0F

The low three index bits allow for selective writing of fields.

000	Write nothing
001	Write CURRENT field only
010	Write CONSOLE field only
100	Write ALT field only
111	Write all fields

Figure 16–28 Mbox Process Mode Register — M_MODE[tpu]

Table 16–32 Mbox Process Mode Register Field Descriptions

Field Name	Extent	Type	Description
Reserved	63:8	As follows:	
		Bits	**Type**
		63:52	MBZ
		51:33	X
		32:28	MBZ
		27:19	X
		18	MBZ
		17:8	X
ALT	7:6	RW,0	The ALT field is encoded as follows: 00 — Kernel 01 — Executive 10 — Supervisor 11 — User
CONSOLE	5	RW,0	DTB traps in console mode are reported to the trap handler separately from non-console mode traps so they can be vectored to different addresses.
CURRENT	4:3	RW,0	The CURRENT field is encoded as follows: 00 — Kernel 01 — Executive 10 — Supervisor 11 — User
Reserved	2:0	MBZ	

16.3.9 Mbox Process Context register — M_PCTX[tpu]

The Mbox process context register is a copy of the Ibox process context register stored locally to the Mbox for implementation convenience.

Written	HW_MTPR
Readable	HW_MFPR
Index	0x10-0x13

The low two index bits allow for selective writing of fields.

00	Write nothing
01	Write ASN field only
10	Write TPU_GRP field only
11	Write all fields

Figure 16–29 Mbox Process Context Register — M_PCTX[tpu]

Table 16–33 Mbox Process Context Register Field Descriptions

Field Name	Extent	Type	Description
Reserved	63:18	As follows:	
		Bits	**Type**
		63:52	MBZ
		51:33	X
		32:28	MBZ
		27:19	X
		18	MBZ
TPU_GRP	17:16	RW,0	Thread Group number this TPU belongs to
ASN	15:8	RW,0	Address space number, should be identical to the value in PCTX controlling the ITB
Reserved	7:0	As follows:	
		Bits	**Type**
		7:3	X
		2:0	MBZ

16.3.10 Mbox Memory Management Status Register — M_STAT[tpu]

The memory management status register is implicitly written register containing information about the most recent Dstream TB miss or fault in the TPU.

The traps that set M_STAT are:

UNALIGN

DFAULT

MCHK

DTBM_SINGLE

DTBM_SINGLE_CONS

*** One of the bits should indicate a BAD_VA fault (sign extension check failure)???

Written	Implicitly written when a Dstream fault occurs.
Readable	HW_MFPR
Index	0x20

Figure 16–30 Mbox Memory Management Status Register — M_STAT[tpu]

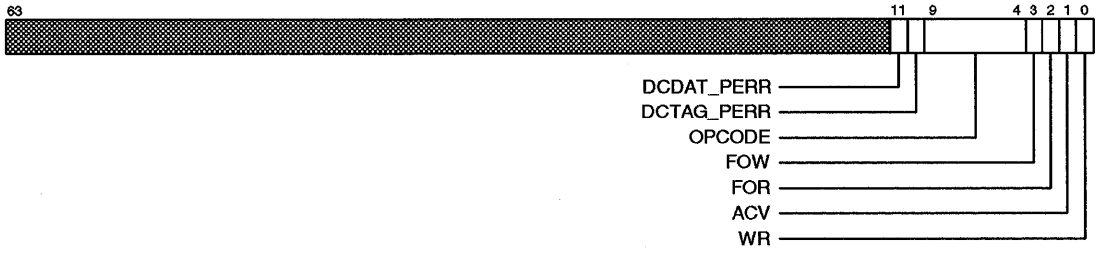


Table 16–34 Mbox Memory Management Status Register Field Descriptions

Field Name	Extent	Type	Description
Reserved	63:12	RAZ	
DCDAT_PERR	11	IR,0	Set when a Dcache data parity error occurs during the initial tag probe of a load or store instruction. A DFAULT PALmode trap is generated.
DCTAG_PERR	10	IR,0	Set when a Dcache tag parity error occurs during the initial tag probe of a load or store instruction. A DFAULT PALmode trap is generated.
OPCODE	9:4	IR,0	The opcode of the instruction that generated the error.
FOW	3	IR,0	Set when a fault-on-write error occurs and PTE[FOW] was set
FOR	2	IR,0	Set when a fault-on-read error occurs and PTE[FOR] was set
ACV	1	IR,0	Set when an access violation occurs. This includes bad virtual addresses
WR	0	IR,0	Set when an error occurs during a write operation

16.3.11 Quiesce Timeout Register — QUIESCE_TIMEOUT[tpu]

This IPR specifies a limit to the number of CPU cycles that may elapse between issuing the QUIESCE instruction and the watch_flag (see WATCH_PHYS_ADDR) being cleared. The value in this register is used to load the quiesce timer.

Allowing this value to be set per-TPU is necessary to give the 21464 the capability of running virtual machines, i.e., the ability for different TPUs to run different O/S's simultaneously.

Does the counter wrap or saturate and what is the behavior of a zero value?

The background for booting document assumes the ability to define an infinite wait condition in the Suspending TPUs section. Disabling the timer should suffice.

Written	HW_MTPR
Readable	HW_MFPR, Implicitly by Quiesce instruction
Index	0x38

Figure 16–31 Quiesce Timeout Register — QUIESCE_TIMEOUT[tpu]

Table 16–35 Quiesce Timeout Register Field Descriptions

Field Name	Extent	Type	Description
Reserved	63:20	MBZ	
TIMEOUT	19:4	RW, 0x280	Number of CPU cycles to wait before clearing the watch_flag of a quiesced TPU. What does zero do???
Reserved	3:2	MBZ	
WATCH_EN	1	RW,0	Enables comparison against the physical address specified by the LDx_ARM instruction. If disabled, a TPU will not awaken when an access to the WATCH_PHYS_ADDR is detected.
TIMEOUT_EN	0	RW,0	Enables the timeout counter. If disabled, a TPU will not timeout from a quiesce operation.

The actual timer value is not currently readable. While the process is quiesced, that is not important, but would the ability to read the timer value when awaken by an interrupt or address match be useful?

16.3.12 Virtual Address Register — VA[tpu]

When a Dstream fault occurs, the associated virtual address is stored in the VA register. The VA is not written when an LD_VPTE gets a DTB miss or Dstream fault.

Traps that cause VA to be written are:

UNALIGN

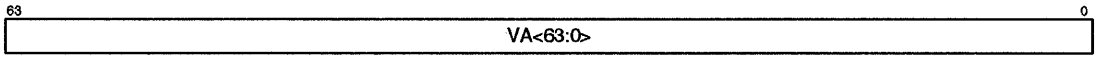
DFAULT

DTBM_SINGLE

DTBM_SINGLE_CONS

Written	Implicitly by instruction that caused the miss.
Readable	HW_MFPR
Index	0x21

Figure 16–32 Virtual Address Register — VA[tpu]



NOTE: The SRM states that the pre-endian adjusted address (va, not va') is reported for memory management faults. The 21464 stores va' and requires PALcode to adjust back to va where necessary.

16.3.13 Virtual Address Format Register — VA_FORM[tpu]

The read-only virtual address format register contains the virtual page table entry address derived from the faulting virtual address stored in the VA register along with the virtual page table base and associated control bits stored in the VA_CTL register.

Written	N/A (Derived from address in VA and control bits in M_CTL)
Readable	HW_MFPR
Index	0x23

Figure 16–33 Virtual Address Format Register — VA_FORM[tpu]

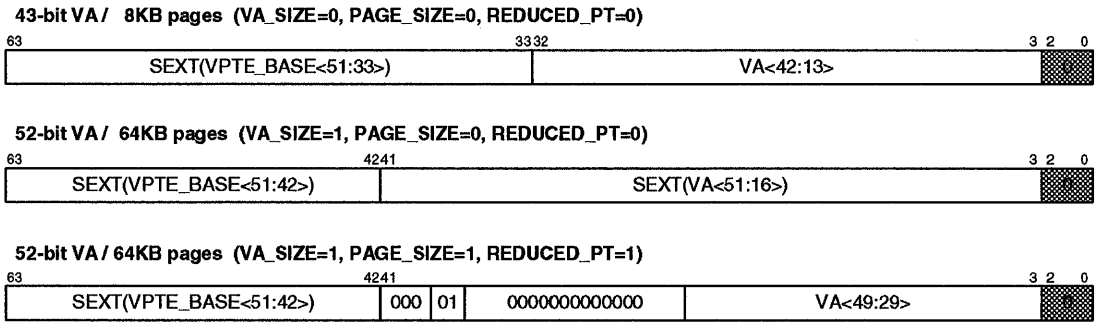


Table 16–36 Instruction VA Format Register (43-Bit VA) Fields Description

Field Name	Extent	Type	Description
SEXT(VPTE_BASE<51:33>)	63:33		
VA<42:13>	32:3		
Reserved	2:0	RAZ	—

Table 16–37 Instruction VA Format Register (52-Bit VA, REDUCED-PT=0) Fields Description

Field Name	Extent	Type	Description
SEXT(VPTE_BASE<51:42>)	63:42		
SEXT(VA<51:16>)	41:3		
Reserved	2:0	RAZ	—

Table 16–38 Instruction VA Format Register (52-Bit VA, REDUCED-PT=1) Fields Description

Field Name	Extent	Type	Description
SEXT(VPTE_BASE<51:42>)	63:42		
	41:39		
	38:37		
	36:24		
VA<49:29>	23:3		
Reserved	2:0	RAZ	—

16.3.14 Watch Physical Address Register — WATCH_PHYS_ADDR[tpu]

When a LDx_ARM instruction retires, the physical address specified is loaded into this register and the watch flag is set. If the watch flag is still set when a Quiesce instruction to the TPU retires, the TPU is put to sleep until the flag is cleared.

The watch flag is cleared by a memory write to the physical address, an interrupt to the TPU or when the quiesce timer expires.

Written	Implicitly by LDx_ARM instruction
Readable	No
Index	N/A

Figure 16–34 Watch Physical Address Register — WATCH_PHYS_ADDR[tpu]

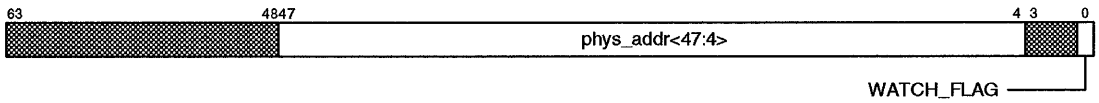


Table 16–39 Watch Physical Address Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:48	MBZ	
PHYS_ADDR<47:4>	47:4		
Reserved	3:1	MBZ	
WATCH_FLAG	0		

16.4 Cbox IPRs

16.4.1 Hardware Interrupt Clear Register — HW_INT_CLR[tpu]

The hardware interrupt clear register is a write-only register used to clear edge-sensitive interrupt requests.

I believe this register is moving to the Cbox and will be completely reworked given how the 21364/21464 handle interupts, as opposed to the 21264.

Note: The FBTP bit will be move.

Written ??
Readable No

Figure 16–35 Hardware Interrupt Clear Register — HW_INT_CLR[tpu]

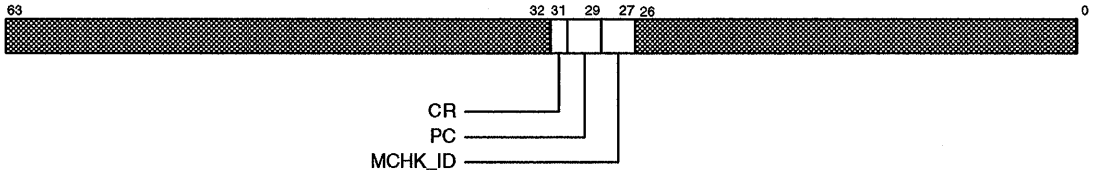


Table 16–40 Hardware Interrupt Clear Register Fields Description

Field Name	Extent	Type	Description
Reserved	63:32	—	
CR	31		Clears a corrected read error interrupt request.
PC	30:29		Clears a performance counter interrupt request.
MCHK_ID	28:27		Clears a Dstream machine check interrupt request.
Reserved	26:0	—	

16.5 Rbox IPRs

This section describes the Rbox IPRs.

16.5.1 Router Configuration1 (R,W) — R_CFG1

Table 16–41 shows the router configuration register fields.

Table 16–41 Router-Configuration1 Register Fields Description

Bit	Field	Value	Meaning	Comments
<31>	IRW			If set then ignore writes to the Router Table. This bit helps reduce the risk that an errant IPR write will corrupt the Routing table. Table 2: Router-Configuration register (Part 2)
<30:25>	reserved			
<24:22>	DRI<2:0>	0 1 2 3 4 5 6 7	0 3 15 63 255 1023 409 516383	Drain Interval. Indicates how many cycles after the drain interval starts, before the router forces out the starved packet. The drain interval starts once an input-buffer slot becomes available for the starved packet.
<21>	DRE			Enable Drain Mode

Table 16–41 Router-Configuration1 Register Fields Description

Bit	Field	Value	Meaning	Comments
<20:18>	STI	0 1 2 3 4 5 6 7	0 3 15 63 255 1023 4095 16383	Starvation Interval. Indicates how many cycles the starvation token can last in the header queue before it triggers the starvation mode. When in starvation mode, the router treats all packets in the header queue in front of the token as starved.
<17>	STE			Enable Starvation mode
<16:14>	SYF<2:0>	N (range is 0:7)		SYNCH frequency: Interval = (N+1) * 4096 cycles Period = (N+1) * (1024)2 cycles
<13>	SYE			Enable SYNCHs
<12:10>	reserved			
<9>	ADA			Enable adaptive routing when set.
<8:7>	SHB<1:0>	0 1 2 3	0 64 128 256	Size of Packet queue in ticks. A value of zero, and ADA = 0 forces deterministic routing. The other values allow performance experiments.
<6:3>	BRO<3:0>	<0> <1> <2> <3>	North South East West	Determines which output ports to send the broadcast packet that are in the local input port.
<2:1>	TRT	0 1 2 3	North South East West	Turn Route Type: Selects whether the routing type is North-last, etc. North-last and South-Last imply XY-routing for the Starvation-recovery routine. East-last and West-last imply YX-routing.
<0>	ECB	01	Normal Low	ECC Bypass: If set enable the low latency ECC checker. (Currently not implemented)

16.5.2 Router Configuration2 (R, W) — R_CFG2

Table 16–42 shows the Router Configuration2 register fields.

Table 16–42 Router-Configuration2 Register Fields Description

Bit	Field	Value	Meaning	Comments
<31:22>	reserved			
<21:20>	TG1	0 1 2 3	0% 33% 67% 100%	Toggle rate for Header queue entries 4 through 15. Same as TG0.
<19:18>	TG0	0 1 2 3	0% 33% 67% 100%	Toggle rate for the first four Header-Queue entries. Chooses fifth channel over the Adaptive-route field.

Table 16–42 Router-Configuration2 Register Fields Description

Bit	Field	Value	Meaning	Comments
<17:14>	WAL<3:0>	<0> <1> <2> <3>	North O/P South O/P East O/P West O/P	Wall: The output port is a wall. Packets with the Bounce bit set will turn on encountering the wall.
<13:12>	DNC	0 1 2 3	16 32 64 128	De-allocate NOP Counter: Number of network cycles before the 21464 tries to force a De-allocate NOP on the links. This NOP will dispatch as soon as any packet currently on the link completes.
<11>	WID	0 1	Narrow Wide	Width: Selects the width of the network links.
<10:9>	FDR<1:0>	0 1 2 3	No force every 4th every 8th every 16th	Force Deterministic Route: Force every 4th to 16th cycle to route deterministically. This is a performance tweak.
<8>	TCB	0 1	normal 2-cycle	Two Cycle Bid: The local arbiter does not bid in the cycle after issuing a bid. Setting this bit prevents the local arbiter from bidding in the next two cycles, which allows packets to route in order.
<7:6>	reserved			
<5:0>	DRM	<0> <1> <2> <3> <4> <5>	Request Forward Block-Response Victim-Block Non-Block Release	Deterministically Route Message class: Setting a bit causes all packets in the selected message class to route deterministically. This is an insurance policy in case we find a ships-passing-in-the-night problem

16.5.3 Router Channel {N,S,E,W} Configuration1 (R,W) — R_*n*_CFG1

There are four such registers - one per network port - called R_N_CFG1, R_S_CFG1, R_E_CFG1, R_W_CFG1.

Table 16–43 Router-{N,S,E,W}-Configuration1 Register Fields Description

Bit	Field	Value	Meaning	Comments
<31:26>	reserved			
<25>	NUL			When set, the Output-Port issues Null ticks.
<24:21>	SPD<3:0>	0	1:1	The clock ratio between the interface and the CPU. For instance, a ratio of 3:2 means that there are three CPU clocks for every 2 interface clocks.
		1	3:2	
		2	2:1	
		3	5:2	
		4	3:1	This table is still TBD. Need some slower ratios for test purposes.
		5	7:2	
		6	4:1	
		7	9:2	
		8	5:1	
		9	11:2	
		10	6:1	
		11	13:2	
		12	7:1	
		13	15:2	
		14	8:1	
		15	reserved	
<20>	IGD			Ignore incoming de-allocate NOP packets. This is a test bit, used to ensure that the router timers expire.
<19:18>	FEM<1:0>	0	Normal	Force-Error mode:
		1	Error-D	Error-D: force 1-shot, double-bit error
		2	Error-C	Error-C: force a continuous stream of 1-bit errors
		3	Error-S	Error-S: force 1-shot, single-bit error
				A hidden, 20-bit counter triggers the forced error modes. Once per million GCLK clock cycles, this counter forces the error on a random, outward-bound packet tick.
				The Router-TCTL IPR defines this counter. A write to this IPR clears the hidden counter, and clears the force-error conditions.
				The interval timer, the SYNC timer, and the time-out timer, all share this hidden counter.
<17>	INI			Initialize Mode: Causes the port to go through a clock-forward init on the next fast reset. Also causes the output to send true NOP packets rather than NUL-NOP packets, until the clock initialization sequence completes. The hardware clears this bit at the end of a clock-forward initialization sequence.
<16>	SYC			Run the port input in synchronous mode
<15:14>	UNI			Unload pointer init value (for clock-forward reset)
<13>	SYE			Enable the port to respond to a SYNCH.

Table 16–43 Router-{N,S,E,W}-Configuration1 Register Fields Description

Bit	Field	Value	Meaning	Comments
<12>	FCC			Enable the port to check the Forwarded clock. This check logic confirms that the clocks are at the expected rate, and that the clocks are in synchronization.
<11>	ECC			Enable the ECC checking/correction logic.
<10>	SAE			Enable the port to respond to a SW alert
<9>	HAE			Enable the port to respond to a HW alert
<8>	reserved			
<7>	reserved			
<6:3>	BRO<3:0>	<0> <1> <2> <3>	North South East West	Broadcast Output port: These bits direct the broadcast packet on the Local Port to the enabled output ports.
<2>	OE			Output Port Enable: If clear, the router discards any packet destined for this port. The hardware clears this bit when the channel goes down.
<1>	ICO			Input Connected to Output: If set the input port is connected to an output port (i.e., another node). If it is not connected then the hardware disables the port logic to minimize power, noise, etc.
<0>	IE			Input Port Enable: When clear the router ignores any packets on this input port. The hardware clears this bit when the channel goes down.

16.5.4 Router Channel {N,S,E,W} Configuration2 (R,W) — R_*n*_CFG2

Table 16–44 shows the Router Channel Configuration2 register fields.

Table 16–44 Router Channel {N,S,E,W} Configuration2 Register Fields Description

Bit	Field	Value	Meaning	Comments
<31:9>	reserved			
<8:6>	FOF<2:0>			Output-FIFO Fullness offset for fifth channel see SOF<2:0>
<5:3>	TOF<2:0>			Output-FIFO Fullness offset for turning path: see SOF<2:0>
<2:0>	SOF<2:0>			Output-FIFO-Fullness offset for straight-through path: This value is added to the actual Output Buffer fullness amount in the pre-decode logic. The pre-decode uses this result to determine which output FIFO is the least heavily used, and it routes new packets to this output port.

16.5.5 Router Channel {N,S,E,W} Timer1 Configuration (R,W) — R_*n*_T1CFG

There are four such registers - one per network port - called R_N_T1CFG, R_S_T1CFG, R_E_T1CFG, R_W_T1CFG.

Table 16–45 Router {N,S,E,W} Timer1 Configuration Register Fields Description

Bit	Field	Value	Meaning	Comments
<31:28>	reserved			
<27>	WITE			Enable the Write-IO timer
<26:21>	WITV<5:0>			Write-IO message-class timer value
<20>	RITE			Enable the Read-IO timer
<19:14>	RITV<5:0>			Read-IO message-class timer value
<13>	FWTE			Enable the Forward timer
<12:7>	FWTV<5:0>			Forward message-class timer value
<6>	RSTE			Enable the Response timer
<5:0>	RSTV<5:0>			Response message-class (both block and non-block) timer value

16.5.6 Router Channel {N,S,E,W} Timer2 Configuration (R,W) — R_*n*_T2CFG

There are four such registers - one per network port - called R_N_T2CFG, R_S_T2CFG, R_E_T2CFG, R_W_T2CFG.

Table 16–46 Router {N,S,E,W} Timer2 Configuration Register Fields Description

Bit	Field	Value	Meaning	Comments
<31:21>	reserved			
<20>	FITE			Enable the Fan-in timer
<19:14>	FITV<5:0>			Broadcast-Acknowledge Fan-in class timer value
<13>	FOTE			Enable the Fan-out timer
<12:7>	FOTV<5:0>			Broadcast Fan-out class timer value
<6>	RETE			Enable the Request timer
<5:0>	RETV<5:0>			Request message-class timer value

16.5.7 Router Channel {N,S,E,W} Error Status (R, W1C) — R_*n*_ERR

There are four such registers - one per network port - called R_N_ERR, R_S_ERR, R_E_ERR, R_W_ERR.

Table 16–47 Router {N,S,E,W} Error Status Register Fields Description

Bit	Field	Value	Meaning	Comments
<31:19>	reserved			
<18>	FITX			Broadcast-Acknowledge Fan-in Timer expired
<17>	FOTX			Broadcast Fan-out Timer expired
<16>	WITX			Write-IO Timer expired
<15>	RITX			Read-IO Timer expired
<14>	FWTX			Forward Timer expired
<13>	RETX			Request Timer expired
<12>	RSTX			Response Timer expired
<11>	reserved			
<10>	FCE			Forwarded-clock error: To determine the presence of a clock, then clear this bit are read it again (while the forward-clock checking is enabled).
<9>	DBE			Double-bit error
<8:2>	SYN<6:0>			ECC syndrome
<1>	MSE			Multiple, single-bit errors
<0>	SBE			Single-bit error

Notes:

- The syndrome reflects the first error condition. For example, if the double-bit error bit is set then the syndrome is for the first occurrence of the double-bit error.
- The hardware will disable the input and output ports on the failing compass point, for the following errors:
  - Double-bit error
  - Forward-clock error
  - The expiration of any timer.

In addition, the hardware will force the adjacent node to shut-down its port by forcing a double-bit error in the first tick of the next packet heading to the adjacent node.

16.5.8 Router Channel {N,S,E,W} Performance Counter (R, W) — R_*n*_PERF

There are four such registers - one per network port - called R_N_PERF, R_S_PERF, R_E_PERF, R_W_PERF.

The PCV counter stops incrementing when it reaches the maximum value (all ones). It also sets an interrupt at this point if the interrupt mask has enabled this interrupt.

Table 16–48 Router {N,S,E,W} Performance Counter Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:3>	PCV<27:0>			Counter value
<2:0>	PCC			Performance Counter Selection:
		0		Port usage 0 - increment the count for every outward tick.
		1		Port usage 1 - increment the count for every outward packet
		2		TBD
		3		TBD
		4		TBD
		5		TBD
		6		TBD
		7		TBD

16.5.9 Router I/O-Port Configuration1 Register (R, W) — R_IO_CFG1

Table 16–49 shows the Router I/O Port Configuration register fields.

Table 16–49 Router I/O-Port Configuration Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:27>	reserved			
<26>	KCL			Keep Clock running: The IO-ASIC may derive its clock from the forwarded clock sent by the 21464. If this bit is set, then keep the forwarded clock running, and instead set the DTN (Drive True-NOP) bit (see above).
<25>	NUL			When set, the Output-Port issues Null ticks.
<24:21>	SPD<3:0>	0	3:2	The clock ratio between the interface and the CPU. For instance, a ratio of 3:2 means that there are three CPU clocks for every 2 interface clocks.
		1	2:1	
		2	5:2	
		3	3:1	
		4	7:2	
		5	4:1	
		6	9:2	
		7	5:1	
		8	11:2	
		9	6:1	
		10	13:2	
		11	7:1	
		12	15:2	
		13	8:1	
		14	reserved	
		15		
		reserved		
<20>	IGD			Ignore incoming de-allocate NOP packets. This is a test bit, used to ensure that the router timers expire.

Table 16–49 Router I/O-Port Configuration Register Fields Description

Bits	Field	Value	Meaning	Comments
<19:18>	FEM<1:0>	0 1 2 3	Normal Error-D Error-C Error-S	Force error mode: Error-D: force 1-shot, double-bit error Error-C: force a continuous stream of 1-bit errors Error-S: force 1-shot, single-bit error A hidden, 20-bit counter triggers the forced error modes. Once per million G clocks, this counter forces the error on a random, outward-bound packet tick. The Router_TCTL IPR defines this counter. A write to this IPR clears the hidden counter, and clears the force-error conditions. The interval timer, the SYNC timer, and the time-out timer, all share this hidden counter.
<17>	DTN			Drive True-NOP packet. Software should first assert this bit before pulsing the UNU bit (see previous field). Hardware will then reset the bit. Hardware will set this bit whenever it disables the I/O port because of an error condition, and if the KCL (Keep clock - see below) bit is set. If this bit is set then the port will discard any messages that are attempting to travel through the port.
<16>	UNU (RAZ, RW)			Unload Pointer Update (for lock-step synchronous mode). Setting this bit: Causes the unload pointer to initialize to the UNI value and to start counting when the clock-forward initialize sequence begins (i.e., when the NUL-NOP is sent). Causes the DTN bit (see next field) to transition from one to zero
<15:14>	UNI<1:0>			Unload pointer init value (for lock-step synchronous mode).
<13>	SYE			Synchronous input mode enabled (for lock-step)
<12>	FCC			Enable the port to check the Forwarded clock. This check logic confirms that the clocks are at the expected rate, and that the clocks are in synchronization.
<11>	ECC			Enable ECC checking and correcting.
<10>	reserved			
<9>	HAE			Enable the port to respond to a HW alert
<8:3>	reserved			
<2>	OE			Output Port Enable. If clear, the router discards any packet destined for this port. The hardware clears this bit when the channel goes down.
<1>	reserved			
<0>	IE			Input Port Enable. When clear the router ignores any packets on this input port. The hardware clears this bit when the channel goes down.

16.5.10 Router I/O-Port Configuration2 Register (R, W) — R_IO_CFG2

Table 16–50 shows the Router I/O Port Configuration2 register fields.

Table 16–50 Router I/O-Port Configuration 2 Register Field Description

Bits	Field	Value	Meaning	Comments
<31:13>	reserved			
<13>	FTG	01	Toggle No toggle	Disable toggling of fifth channel in the local arbiter when selecting routing direction.
<12>	FCW	01	East West	Fifth channel is wired in an East or West direction
<11>	FCS	01	North South	Fifth channel is wired in a North or South direction
<10:6>	FEW<4:0>			East-West Coordinates of node at the other end of the fifth channel.
<5:1>	FNS<4:0>			North-South Coordinates of node at the other end of the fifth channel.
0	FIO	01	IO-ASIC 5th	I/O Channel usage.

16.5.11 Router I/O-Port Buffer Size (R,W) — R_IO_BUFSIZ

This register only applies when the I/O-port is acting as an I/O channel. When it is acting as a fifth channel then it ignores this register. The buffer sizes correspond to the size of the input buffers inside the I/O ASIC. (Hence there is no request buffer because the network does not send request packets to the I/O ASIC.)

Table 16–51 Router I/O-Port Buffer Size Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:15>	reserved			
<14:12>	WIB<2:0>			Number of Write-IO buffers in IO-ASIC
<11:9>	RIB<2:0>			Number of Read-IO buffers in IO-ASIC
<8:6>	FWB<2:0>			Number of Forward buffers in IO-ASIC
<5:3>	NSB<2:0>			Number of Non-Block-response buffers in IO-ASIC
<2:0>	RSB<2:0>			Number of Block-response buffers in IO-ASIC

16.5.12 Router I/O-Port Timer1 Configuration (R,W) — R_IO_T1CFG

These timers apply to the out-going port. Whenever the timers expire then the I./O-port starts to discard the packets. These values apply whether the I/O-port is acting as the I/O-channel or the Fifth-channel. However, the fifth channel never transfers Read-IO or Write-IO packets, and thus these counters should not be enabled.

Table 16–52 RouterI/O-Port Timer1 Configuration Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:28>	reserved			
<27>	WITE			Enable the Write-IO timer
<26:21>	WITV<5:0>			Write-IO message-class timer value
<20>	RITE			Enable the Read-IO timer
<19:14>	RITV<5:0>			Read-IO message-class timer value
<13>	FWTE			Enable the Forward timer
<12:7>	FWTV<5:0>			Forward message-class timer value
<6>	RSTE			Enable the Response timer
<5:0>	RSTV<5:0>			Response message-class (both block and non-block) timer value

16.5.13 Router I/O-Port Timer2 Configuration (R,W)— R_IO_T2CFG

This register only applies when the I/O-port is acting as a fifth channel. When the I/O-port is acting as a I/O-channel then it disables these timers.

Table 16–53 Router I/O-Port Timer2 Configuration Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:7>	reserved			
<6>	RETE			Enable the Request timer
<5:0>	RETV<5:0>			Request message-class timer value

16.5.14 Router I/O-Port Error Status (R, W1C) — R_IO_ERR

The RETX field only applies to the fifth channel. The RITX and WITX only apply to the I/O channel.

Table 16–54 Router I/O-Port Error Status Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:17>	reserved			
<16>	WITX			Write-IO Timer expired (I/O channel only)
<15>	RITX			Read-IO Timer expired (I/O channel only)
<14>	FWTX			Forward Timer expired
<13>	RETX			Request Timer expired (Fifth channel only)
<12>	RSTX			Response Timer expired
<11>	reserved			

Table 16–54 Router I/O-Port Error Status Register Fields Description

Bits	Field	Value	Meaning	Comments
<10>	FCE			Forwarded-clock error: To determine the presence of a clock, then clear this bit are read it again (while the forward-clock checking is enabled).
<9>	DBE			Double-bit error
<8:2>	SYN<6:0>			ECC syndrome
<1>	MSE			Multiple, single-bit errors
<0>	SBE			Single-bit error

Notes:

- The syndrome reflects the first error condition. For example, if the double-bit error bit is set then the syndrome is for the first occurrence of the double-bit error.
- The hardware will disable the input and output ports on the failing compass point, for the following errors:
  - Double-bit error
  - Forward-clock error
  - The expiration of any timer.

In addition, the hardware will force the adjacent node to shut-down its port by forcing a double-bit error in the first tick of the next packet heading to the adjacent node.

16.5.15 Router I/O-Port Performance Counter (R, W) — R_IO_PERF

The PCV counter stops incrementing when it reaches the maximum value (all ones). It also sets an interrupt at this point if the interrupt mask has enabled this interrupt.

Table 16–55 Router I/O-Port Performance Counter Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:3>	PCV<27:0>			Counter value
<2:0>	PCC			Performance Counter Selection:
		0		Port usage 0 - increment the count for every outward tick.
		1		Port usage 1 - increment the count for every outward packet
		2		TBD
		3		TBD
		4		TBD
		5		TBD
		6		TBD
		7		TBD

16.5.16 Router Local-Port Error Status Register (R, W1C) — R_LOC_ERR

The 21464 does not check the interface between the router and the Scache/memory for errors. The packets travel to and from the Router and the C-box or Z-box without ECC or parity bits. However, the interface ports perform error checking, as follows: The input ports write a reserved-double-bit error pattern into the packet tick on detecting a double-bit error. When this

packet arrives at the local-output port, the router sends the packet to the C-box with the error signal asserted.

Table 16–56 Router I/O-Port Error Status Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:9>	reserved			
<8>	RTP			Router-Table parity error
<7:1>	reserved			
<0>	RES			Reserved Double-bit error code detected

16.5.17 Router Routing Table Register (R,W) — R_ROUT

This table holds the routing information the packet needs to reach the destination processor. There are 532 routing table entries, There is one for each of the 512 nodes, and 20 for each sharing mask bit in the directory mask. These 20 entries define which node the router sends the SharedInvalBroadcast packets.

Table 16–57 Router Routing Table Register Fields Description

Bits	Field	Value	Meaning	Comments
<23>	PAR			Parity
<22>	reserved			
<21:20>	IND	0 1 2 3	North South East West	Initial Direction: This field defines which direction the packet will leave the source node if CAD = 0. If CAD= 1 then hardware will ignore this field if it chooses to adaptively route. The Initial direction should always be on the deterministic path (even if CAD = 1).
<19:16>	DNS<3:0>			North-South (Y) coordinate of the destination
<15:12>	DEW<3:0>			East-West (X) coordinate of the destination
<11>	REW	0 1	East West	Route in the East-West direction
<10>	RNS	0 1	North South	Route in the North-South direction
<9>	STF	0 1	Cut-thru. Store/Fwd	Store-&-Forward. If set then packet destined for an I/O ASIC waits in destination node until complete packet is in the output FIFO. Only required if packet crosses a slow link.
<8>	BOU			Bounce: If set the packet will turn on encountering the wall; otherwise it will pass through the wall.
<7>	CAD			Can-adapt: When set the packet can adapt (i.e., use the A-route paths). Typically, this bit is set. The error-recovery code clears this bit when it needs to prevent a packet from adaptively routing into a faulty region of the network.
<6>	PME			If set, request packet can access memory at this destination node.
<5>	PIO			If set, I/O-packets can access the IO-ASIC at this destination node.

Table 16–57 Router Routing Table Register Fields Description

Bits	Field	Value	Meaning	Comments
<4>	PIP			If set, I/O-packets can access the IPRs at this destination node.
<3>	IME			If set then the IO-ASIC (on this node) can access memory at this destination node.
<2>	IIO			If set then the IO-ASIC (on this node) can access the IO-ASIC at this destination node (for peer-to-peer transactions).
<1>	IIP			If set then the IO-ASIC (on this node) can access the IPRs at this destination node.
<0>	VAL			Destination is valid for all transfers

Occasionally, a torus offers two paths to the destination node from the current node that are equidistant. The routing algorithm should favor both paths equally, because this increases the network performance. The 21464 uses the 21363 approach, which assumes that the software creating the routing table has balanced the equidistant paths amongst the nodes. Thus an individual node will favor a particular direction, but a different node in the network will favor the opposite direction. Hence, overall, the network will exhibit no particular bias.

16.5.18 Router WHOAMI Register (R,W) — R_WHOAMI

This contains a 10-bit address defining the nodes address.

Table 16–58 WhoAmI Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:10>	reserved			
<9:5>	EW<4:0>			East-West (x-axis) coordinate
<4:0>	NS<4:0>			North-South (y-axis) coordinate

16.5.19 Router Overall-Timer-Control Register (R,W) — R_OVER

This register specifies the period between increment pulses to the port, fan-in, and fan-out timers.

Table 16–59 Router Overall-Timer-Control Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:21>	TIM			Timer Value: Software visible portion.
<20:0>	HID			Hidden portion: Only the hardware can write to this portion

16.5.20 Router Interrupt Status (R, WIC) — R_INT_STAT

TBD:

16.5.21 Router Interrupt Mask (R, W) — R_INT_MASK

TBD:

16.5.22 Router Interrupt Request (WO) — R_INT_REQ

TBD:

16.5.23 Router Interrupt Queue Register (RO) — R_INT_QUE

A read of this register reads the head of the interrupt queue. This read is non-destructive - to remove the entry at the head of the queue, software must write an arbitrary value to this register.

TBD:

16.5.24 Router Interrupt Queue Add Register (WO) —R_INT_QUEADD

This register is typically used by I/O devices to post interrupts. A write to this register attempts to add an interrupt identifier to the interrupt queue (via an write-IO command). If the queue is full then the 21464 discards the interrupt identifier and returns a WrION-Ack packet. Otherwise, the write succeeds and the 21464 returns a WrIOAck packet.

The interrupt queue is two-entries deep.

TBD:

16.5.25 Router Interval Timer Register (R,W) — R_INTER_TIM

TBD: Do we need this?

Table 16–60 Router Overall-Timer-Control Register Fields Description

Bits	Field	Value	Meaning	Comments
<31:8>	reserved			
<7:0>	ITV			Interval Timer value

16.5.26 Router Scratch Register 1 (R,W) — R_SCRATCH1

TBD: Do we need this?

16.5.27 Router Scratch Register 2 (R,W) — R_SCRATCH2

TBD: Do we need this?

16.6 Zbox IPRs

This section describes the internal processor registers that control Zbox functions. These registers are duplicated for each of the two memory controllers.

16.6.1 DRAM Error Status 1 – ZBOX*n*_DRAM_ERR_STATUS1

There are two DRAM error status 1 registers; ZBOX0_DRAM_ERR_STATUS1 and ZBOX1_DRAM_ERR_STATUS1.

Figure 16–36 shows the DRAM error status 1 register.

Figure 16–36 DRAM Error Status 1

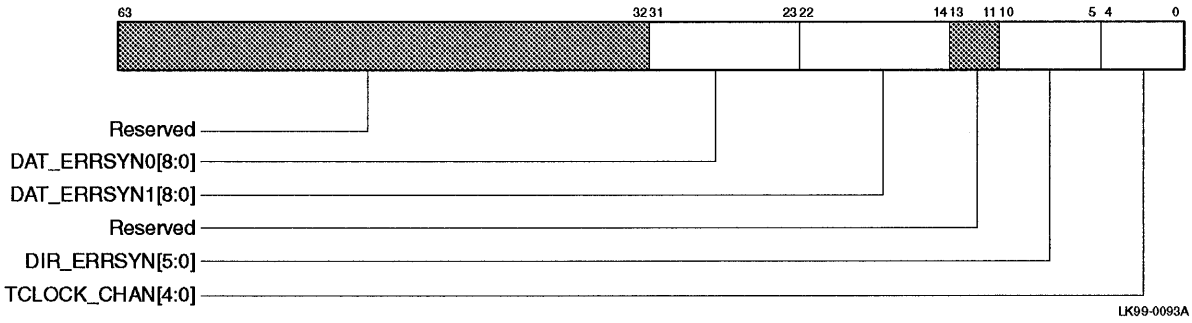


Table 16–61 describes the DRAM error status 1 register fields.

Table 16–61 DRAM Error Status 1 Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RO, MBZ	—
DAT_ERRSYN0[8:0]	[31:23]	RWRC	ECC syndrome for octaword 0 – valid only when RAID, RMAP, SGL, DBL, GE3, or PAR set
DAT_ERRSYN1[8:0]	[22:14]	RWRC	ECC syndrome for octaword 1 – valid only when RAID, RMAP, SGL, DBL, GE3, or PAR set
Reserved	[13:11]	MBZ, WAC	—
DIR_ERRSYN[5:0]	[10:5]	RWRC	Directory syndrome for single-bit ECC error
TCLOCK_CHAN[4:0]	[4:0]	RWAC	Bit mask of which channels had tclock errors – only valid when TCLK error bit set

Any write to DRAM_ERR_STATUS1 forces a load of TCLOCK_CHAN. This should clear it if no errors are occurring.

See ????, which describes error syndromes while a channel is being remapped.

16.6.2 DRAM Error Status 2 – ZBOX*n*_DRAM_ERR_STATUS2

There are two DRAM error status 2 registers; ZBOX0_DRAM_ERR_STATUS2 and ZBOX1_DRAM_ERR_STATUS2.

Figure 16–37 shows the DRAM error status 2 register.

Figure 16–37 DRAM Error Status 2

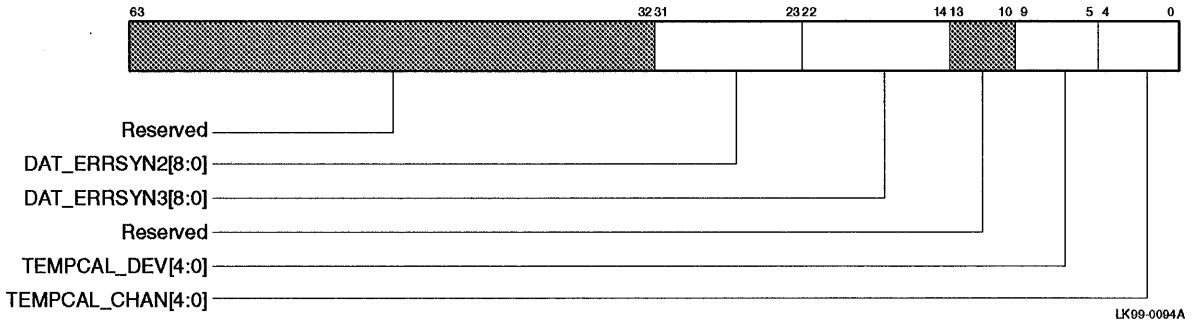


Table 16–62 describes the DRAM error status 2 register fields.

Table 16–62 DRAM Error Status 2 Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RO, MBZ	—
DAT_ERRSYN2[8:0]	[31:23]	RWAC	ECC syndrome for octaword 2 – valid only when RAID, RMAP, SGL, DBL, GE3, or PAR set
DAT_ERRSYN3[8:0]	[22:14]	RWAC	ECC syndrome for octaword 3 – valid only when RAID, RMAP, SGL, DBL, GE3, or PAR set
Reserved	[13:10]	MBZ, WAC	—
TEMPCAL_DEV[4:0]	[9:5]	RWAC	Identifies which device had a temperature calibration error – valid only when TCALERR is set
TEMPCAL_CHAN[4:0]	[4:0]	RWAC	A bit mask of those channels that had temperature calibration errors – valid only when TCALERR is set

Any write to DRAM_ERR_STATUS2 register will:

- Force a reload of the syndrome registers DAT_ERRSYN0, DAT_ERRSYN1, DAT_ERRSYN2, DAT_ERRSYN3, and DIR_ERRSYN (if ECC_COR_ENABLED = 0, the syndromes are defaulted to 0).
- Force a write of TEMPCAL_CHAN (if any read has been performed after reset, this will be some bits of the data read, which will be the same on replicated systems). **Writer note: Unclear – needs updating/augmenting.**
- Clear TEMPCAL_DEV

See ?????, which describes error syndromes while a channel is being remapped.

16.6.3 DRAM Error Status 3 – ZBOX*n*_DRAM_ERR_STATUS3

There are two DRAM error status 3 registers; ZBOX0_DRAM_ERR_STATUS3 and ZBOX1_DRAM_ERR_STATUS3.

Figure 16–38 shows the DRAM error status 3 register.

Figure 16–38 DRAM Error Status 3

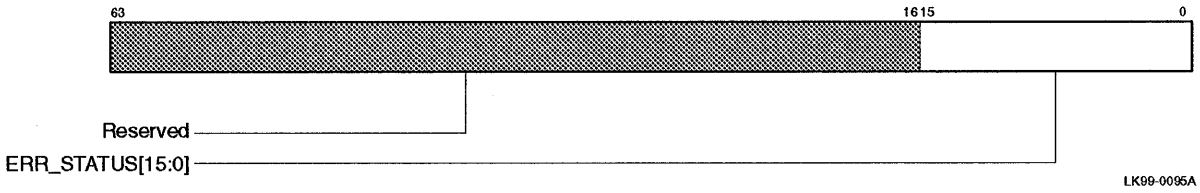


Table 16–63 describes the DRAM error status 3 register fields.

Table 16–63 DRAM Error Status 3 Register Fields Description

Name	Extent	Type	Description
Reserved	[63:16]	RO, MBZ	—
ERR_STATUS[15:0]	[15: 0]	RW1C	Bitmask of DRAM error conditions:

Name	Bit	Meaning When Set
SWP	[15]	An error occurred during sweep mode read
SEO	[14]	A second uncorectable error occurred for which no physical address was saved
MEO	[13]	A second correctable error occurred for which no Phys Addr. was saved
Reserved	[12]	—
Reserved	[11]	—
OLCK	[10]	A DLL had an out-of-lock condition
TIME	[9]	A DIFT timeout occurred
TCAL	[8]	Some channel had a over temperature fault
TCLK	[7]	Some channel had a clock fault
D21	[6]	Directory[21] was read as 1
DBL	[5]	A double ECC error was detected on a read
GE3	[4]	Three or more single ECC errors were detected on a read
MAPF	[3]	A raid-remap occurred, and no unique best remapping was found
RAID	[2]	A raid-remap occurred, and a remapping was selected
SGL	[1]	One or two single bit ECC errors were detected on a read
PAR	[0]	One or more parity errors were detected on a read

See ?????, which describes error syndromes while a channel is being remapped.

16.6.4 DRAM Error Control – ZBOX*n*_DRAM_ERROR_CTL

There are two DRAM error control registers; ZBOX0_DRAM_ERROR_CTL and ZBOX1_DRAM_ERROR_CTL.

Figure 16–39 shows the DRAM error control register.

Figure 16–39 DRAM Error Control

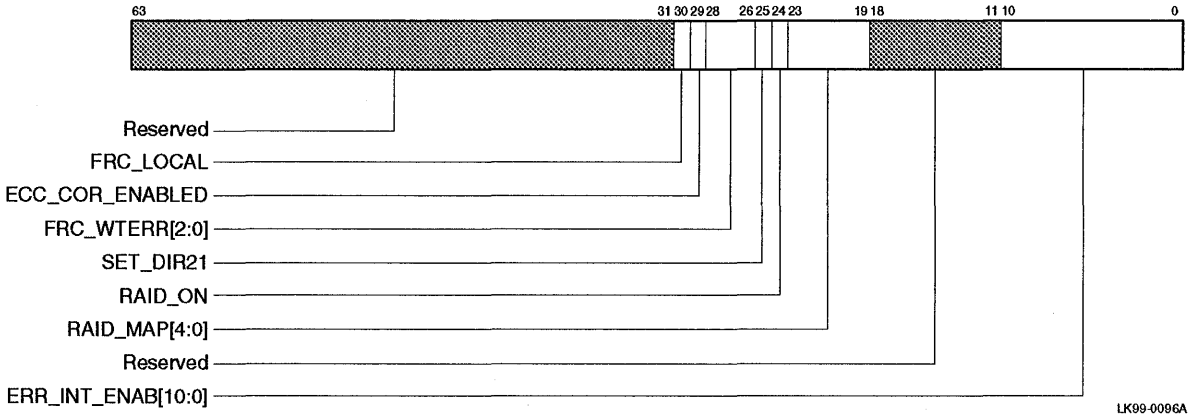


Table 16–64 describes the DRAM error control register fields.

Table 16–64 DRAM Error Control Register Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RW, MBZ	—
FIFTH_CH_ENA	[31]	RW	RAID channel has power and is enabled.
FRC_LOCAL	[30]	RW	If set, forces directory data being sent to the DIFT to be read as zero (local). Directory data that is stored into the DRAM_SWEEP_DIR register when SWEEP_ON is set is not affected. No directory ECC errors are reported when FRC_LOCAL is set.
ECC_COR_ENABLED	[29]	RW	Used to disable ECC correction of fill data to fill buffer. If clear, data is not corrected, errors are not reported, and syndromes are forced to zero.
FRC_WTERR[2:0]	[28:26]	RW	If the address matches the value of Zbox force-error address register, then do one of the following, depending on the value in this field::
			Bits **Meaning** 000 Do nothing 001 Substitute victim_data[27:0] for Dir[21,E5:0,20:0] 010 Force COL_ADR[0] to 0 on channel 4 only 011 Force COL_ADR[0] to 0 on all channels 100 Force COL_ADR[0] to 0 on channel 0 only 101 Force COL_ADR[0] to 0 on channel 1 only 110 Force COL_ADR[0] to 0 on channel 2 only 111 Force COL_ADR[0] to 0 on channel 3 only
SET_DIR_21	[25]	RW	If set, forces the spare directory bit to be set based on address-match when matched block is written to memory (Section 6.7.22). RAID_ON must be set if this function is enabled.

Table 16–64 DRAM Error Control Register Fields Description (Continued)

Name	Extent	Type	Description
RAID_ON	[24]	RW	Used to disable byte writes – If set: 1. FIFTH_CH_ENA must be set 2. Directory byte writes are disabled
RAID_MAP[4:0]	[23:19]	RW	Indicates which channel should be remapped (one-hot encoded). There are only eight legal encodings for the combination of RAID_ON, ECC_COR_ENABLED, and RAID_MAP:

RAID_ON	ECC_COR_ ENABLED	RAID_MAP	
1	1	00000	Use raid channel, nothing mapped out yet
1	1	10000	Raid channel exists, some channel mapped out (used for chanel 4 observation port).
1	1	01000	Raid channel exists, some channel mapped out.
1	1	00100	Raid channel exists, some channel mapped out.
1	1	00010	Raid channel exists, some channel mapped out.
1	1	00001	Raid channel exists, some channel mapped out.
0	1	10000	No raid channel, ECC enabled.
0	0	10000	No raid channel, no ECC ch/corr.

Reserved	[18:11]	RO, MBZ	—
ERR_INT_ENAB[10:0]	[10:0]	RW	If set, enables appropriate interrupt to be generated when the error status bit in the corresponding bit position of ERR_STATUS[10:0] is being set. Setting the enable after the ERR_STATUS bit is already set does not cause an interrupt to be generated.

16.6.5 DRAM Timing Control 1 – ZBOX*n*_DRAM_TIMING_CTL1

There are two DRAM timing control 1 registers; ZBOX0_DRAM_TIMING_CTL1 and ZBOX1_DRAM_TIMING_CTL1.

Figure 16–40 shows the DRAM timing control 1 register.

Figure 16–40 DRAM Timing Control 1

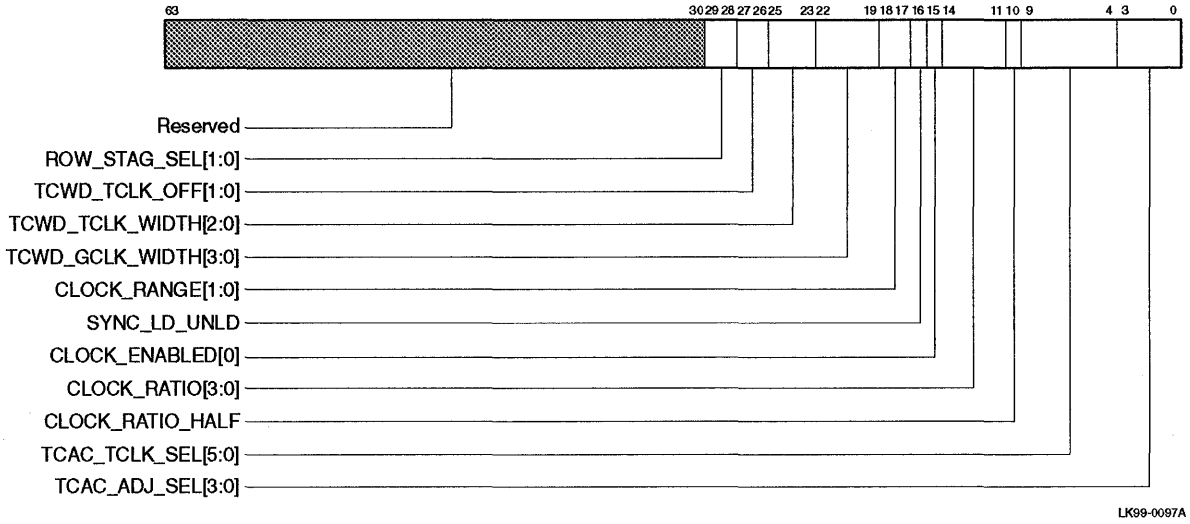


Table 16–65 describes the DRAM timing control 1 register fields.

Table 16–65 DRAM Timing Control 1 Fields Description

Name	Extent	Type	Description
Reserved	[63:30]	RW, MBZ	—
ROW_STAG_SEL[1:0]	[29:28]	RW	Row stagger select for refresh (0, 1, 2 or 4 TLCKs). The refresh operations to the ROW bus can be staggered from channel to channel by 0 (no stagger), 1, 2 or 4 TCLKs, controlled by ROW_STAG_SEL[1:0], encoded as follows: **Value** — **Meaning** 0 — No stagger 1 — 1 TCLK stagger 2 — 2 TCLKs stagger 3 — 4 TCLKs stagger
TCWD_TCLK_OFF[1:0]	[27:26]	RW	Signal **tcwd_off_a_h** adjusts for tCWD less than 6 by starting write_sent pulse at an early TCLK. Values are as follows: **RDRAM tCWD** — **TCWD_TCLK_OFF** ≥6 — 0 5 — 1 4 — 2
TCWD_TCLK_WIDTH[2:0]	[25:23]	RW	See table under TCWD_GCLK_WIDTH[3:0] for tCWD = 4, 5 or 6 values. For tCWD = 7, use the value from the table +1.

Table 16–65 DRAM Timing Control 1 Fields Description (Continued)

Name	Extent	Type	Description
TCWD_GCLK_WIDTH[3:0]	[22:19]	RW	Use the following table for any tCWD value of 4...7: .

Clock Ratio	TCWD_TCLK_WIDTH[2:0]	TCWD_GCLK_WIDTH[3:0]
2	3	0
2.5	3	2
3	4	1
3.5	3	6
4	5	0
4.5	5	1
5	5	2
5.5	5	3
6	5	4
6.5	5	5
7	5	6
7.5	5	7
8	6	0
16	6	8

For RDRAM tCWD < 6, program TCWD_TCLK_WIDTH and TCWD_GCLK_WIDTH as above, and adjust TCWD_TCLK_OFF. For tCWD = 7, add 1 to the TCWD_TCLK_WIDTH value above.

CLOCK_RANGE[1:0]	[18:17]	RW	Tell DLL min/max clk freq range (encoding **TBF**).
SYNC_LD_UNLD	[16]	RW	Synchronize silos.
CLOCK_ENABLED[0]	[15]	RW,0	Rambus clocks are **not** enabled until this bit is set.
CLOCK_RATIO[3:0]	[14:11]	RW	This is the GCLK to TCLK ratio according to this table:

Clock_Ratio[3:0]	GCLK:TCLK Ratio
0	16:1
1	Illegal
2*	2:1
3*	3:1
4*	4:1
5*	5:1
6*	6:1
7*	7:1
8	8:1
9...15	Illegal

CLOCK_RATIO_HALF	[10]	RW	This adds 0.5 to the GCLK to TCLK ratio if set. This bit may only bit set for the ratios marked with * under CLOCK_RATIO[3:0].
TCAC_TCLK_SEL[5:0]	[9:4]	RW	Selects TCAC RDRAM delay parameter (COL = RD→Data). Set this value according to table under TCAC_ADJ_SEL[3:0].

Table 16–65 DRAM Timing Control 1 Fields Description (Continued)

Name	Extent	Type	Description
TCAC_ADJ_SEL[3:0]	[3:0]	RW	Used for fine (GCLK) adjustment of the tCAC parameter. The supported range is 0...15. This is used to: • Compensate for the runway depth • Center the read-strobe in the clock-forward silo data-valid window The following table lists base values (no delay compensation). Baseline CSR values for TCAC (RDRAM tCAC=8 to 0 fine adjustment). For RDRAM tCAC>8, add offset to TCAC_TCLK_SEL baseline value (support for RDRAM tCAC from 7 to 39). For fine adjustment, add needed cycles to TCAC_ADJ_SEL baseline value

Clock Ratio	TCAC_TCLK_SEL[5:0]	TCAC_ADJ_SEL[3:0]
2	9	0
2.5	9	2
3	9	2
3.5	9	2
4	9	2
4.5	9	2
5	9	2
5.5	9	2
6	9	2
6.5	9	2
7	A	6
7.5	9	D
8	9	2
16	9	7

16.6.6 DRAM Timing Control 2 – ZBOX*n*_DRAM_TIMING_CTL2

There are two DRAM timing control 2 registers; ZBOX0_DRAM_TIMING_CTL2 and ZBOX1_DRAM_TIMING_CTL2.

Figure 16–41 shows the DRAM timing control 2 register.

Figure 16–41 DRAM Timing Control 2

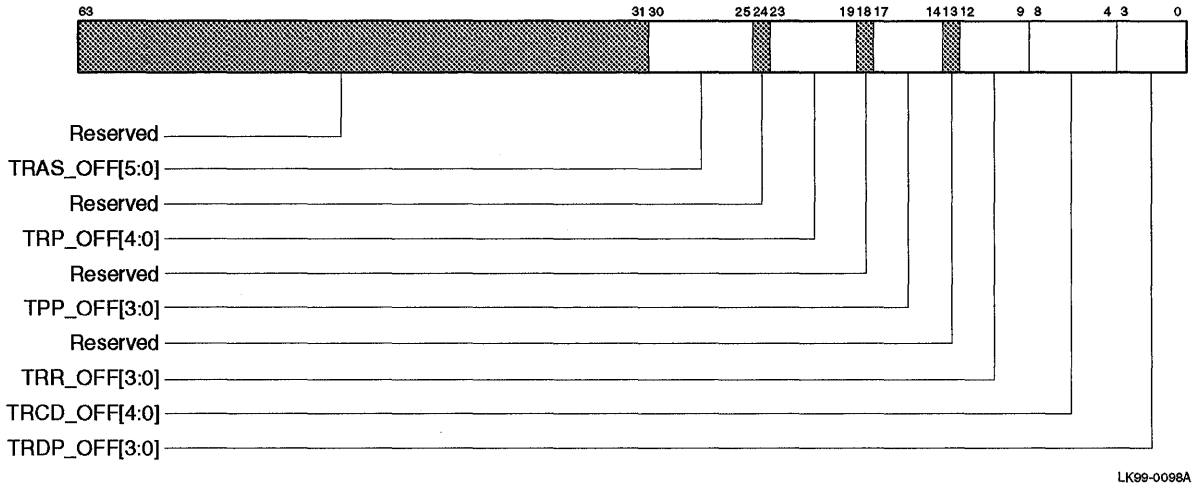


Table 16–66 describes the DRAM timing control 2 register fields.

Table 16–66 DRAM Timing Control 2 Fields Description

Name	Extent	Type	Description
Reserved	[63:31]	RW,MBZ	—
TRAS_OFF[5:0]	[30:25]	RW	Used to determine tRAS (RAS-PRE) TRAS_OFF = Rambus tRAS
Reserved	[24]	RW,MBZ	—
TRP_OFF[4:0]	[23:19]	RW	Used to determine tRP (PRE-RAS) TRP_OFF = Rambus tRP
Reserved	[18]	RW,MBZ	—
TPP_OFF[3:0]	[17:14]	RW	Used to determine tPP (PRE-PRE to same device) TPP_OFF = Rambus tPP
Reserved	[13]	RW,MBZ	—
TRR_OFF[3:0]	[12:9]	RW	Used to determine tRR (RAS-RAS to same device) TRR_OFF = Rambus tRR
TRCD_OFF[4:0]	[8:4]	RW	Used to determine tRCD (RAS-CAS delay) TRP_OFF = Rambus tRP
TRDP_OFF[3:0]	[3:0]	RW	Used to determine tRDP (CAS=RD-PRE delay) TRDP_OFF = Rambus tRDP

16.6.7 DRAM Timing Control 3 – ZBOX*n*_DRAM_TIMING_CTL3

There are two DRAM timing control 3 registers; ZBOX0_DRAM_TIMING_CTL3 and ZBOX1_DRAM_TIMING_CTL3.

Z_SLT attempts to gang commands of like type (rd or wr) in consecutive COLC packets. To prevent locking out the command of unlike type, there is logic which monitors the number of pending and consecutively slotted transactions. Based on thresholds programmed by means of CSRs, the slotter switches to the other command type.

Figure 16–42 shows the DRAM timing control 3 register.

Figure 16–42 DRAM Timing Control 3

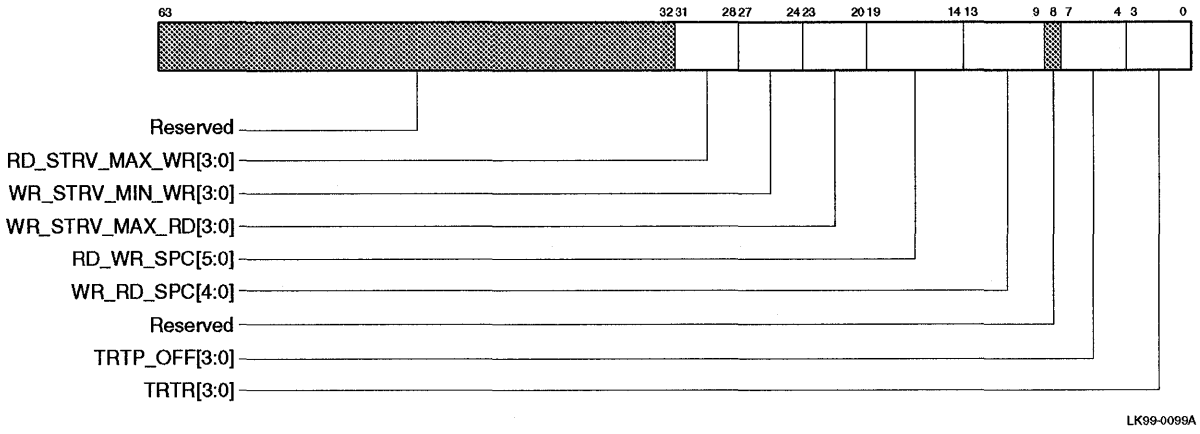


Table 16–67 describes the DRAM timing control 3 register fields.

Table 16–67 DRAM Timing Control 3 Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RO, MBZ	—
RD_STRV_MAX_WR [3:0]	[31:28]	RW	This is used by the read starvation logic. It is the maximum number of consecutive writes that may be slotted to the COLC bus while a read is pending in the ZRQ-CSQ. Supported range is 0...15. This is the maximum number of consecutively-slotted writes that are tolerated before considering reads to be starved. The additional criterion is that there must be at least one (not programmable) read pending in the ZRQ. The CSR is programmed with the desired maximum – 1. Legal values are 0...15, yielding 1...16 maximum number of consecutively slotted writes while a read is pending.
WR_STRV_MIN_WR [3:0]	[27:24]	RW	This is the minimum number of writes that must be pending before interrupting a stream of reads. Used to enforce the Alpha architecture write-timeliness requirement. Supported range is 0...7, yeilding a pending write count of 1...8. To disable interrupting the read stream, use a value of 8 in this field. Values in the range of 9...15 can result in UNPREDICTABLE operation. This is the minimum number of writes that must be pending in the ZRQ before considering writes to be starved by reads.
WR_STRV_MAX_RD [3:0]	[23:20]	RW	This is the maximum number of consecutive reads allowed before attempting to satisfy the write timeliness clause. This is the maximum number of consecutively-slotted reads that will be tolerated before considering writes to be starved. The additional criterion is that there must be at least `z_csrN→wr_strv_min_wr_a_h + 1` writes pending in the ZRQ. The CSR is programmed with the desired maximum – 1. Legal values are 0...15, yielding 1...16 maximum number of consecutively slotted reads while the minimum number of writes are pending.
RD_WR_SPC[5:0]	[19:14]	RW	This is the number of TCLKs of dead time that must be inserted between COLC=RD and COLC=WR packets.
WR_RD_SPC[4:0]	[13:9]	RW	This is the number of TCLKs of dead time that must be inserted between COLC=WR and COLC=RD packets.
Reserved	[8]	RW, MBZ	—
TRTP_OFF[3:0]	[7:4]	RW	Used to determine tRTP (COLC=RET-ROWR=PRER) TRTP_OFF = Rambus tRTP
TRTR[3:0]	[3:0]	RW	Specifies tRTR value (COLC=WR-COLC=RET) or COLC=WR-COLC=BMSK)

16.6.7.1 Calculating Read to Write and Write to Read Spacing

There are CSRs to control the number of "gap" cycles to inject between adjacent read and write transaction packet pairs. These gap cycles are required to avoid driver overlap on the Rambus data lines.

In an ideal (zero skew, zero CFM→CTM delay) arrangement, the CSRs can be programmed such that the last data cycle of the first transaction can be followed immediately (the next cycle) by the first data cycle of the second transaction.

16.6.7.2 Terminology

Timing parameter tCAC is the CAS access delay, and specifies the number of TCLK cycles between the end of a CAS=READ packet and its corresponding data cycles. Direct RDRAMs support tCAC values of 7 to 12 cycles. The ZBox supports a higher tCAC limit to allow for repeater chips, which add approximately 30nS of delay.

Timing parameter tCWD is the CAS write delay, and specifies the number of TCLK cycles between the end of a CAS=WRITE packet and its corresponding data cycles. The Direct RDRAM tCWD parameter is defined as 4 + tCLS. tCLS is a RDRAM core parameter and, for example, has the value of 2 for the 256k × 18 × 16d device, yielding tCWD = 6. tCLS is a 2 bit field, so tCWD can vary from 4 to 7.

16.6.7.3 Ideal Rambus

In an ideal arrangement, the minimum read to write spacing that must be injected (expressed in TCLKs) is tCAC – tCWD.

The formula for calculating the (ideal) read to write spacing CSR in the ZBox, namely, RD_WR_SPC[5:0] is:

```
RD_WR_SPC (IDEAL) = (tCAC-tCWD) + tPACKET
```

similarly for wr_rd spacing:

```
WR_RD_SPC (IDEAL) = (tCWD-tCAC) + tPACKET
```

Note: Because tCWD is typically less than tCAC, a negative result should be that a tPKT value is placed in the CSR (where tPKT = 4 TCLKs).

16.6.7.4 Non-Ideal Rambus

Skews and delays in actual (non-ideal) designs must be accounted for in determining the values of the spacing CSRs. Thus, a read to write transition might need to have further dead cycles injected (for example, to account for delay that could make a read's data collide with a subsequent write's data if IDEAL values are programmed in the CSRs).

System designers must determine these additional delays, round up to the nearest TCLK, and add them to the ideal calculated values. These delays are called:

tRES	tRead Extra Spacing
tWES	tWrite Extra Spacing

Furthermore, when the 21464 is issuing write-data to the channel, there must be 2 TCLKs of dead space before a read transaction can supply its data, to allow ringing on the channel to settle, and not disrupt read data arriving at the 21464. Thus, if tCAC – tCWD < 2, then tWRA (Write Ringing Avoidance) cycles must be added to the write to read spacing to enforce the 2 TCLK minimum.

The CSR equations now become:

```
RD_WR_SPC[5:0] = (tCAC-tCWD) + tRES
```

```
WR_RD_SPC[4:0] = (tCWD-tCAC) + tWES + tWRA
```

(If negative result, set CSR to 4)

16.6.8 DRAM Refresh Control – ZBOX*n*_DRAM_REFR_CTL

There are two DRAM refresh control registers; ZBOX0_DRAM_REFR_CTL and ZBOX1_DRAM_REFR_CTL.

Figure 16–43 shows the DRAM refresh control register.

Figure 16–43 DRAM Refresh Control

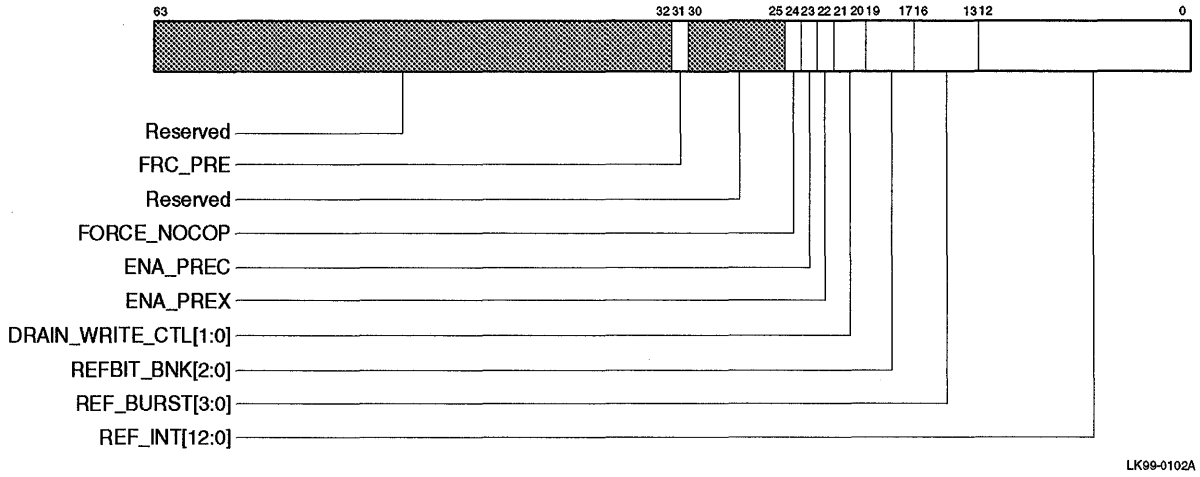


Table 16–68 describes the DRAM refresh control register fields.

Table 16–68 DRAM Refresh Control Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RO, MBZ	—
FRC_PRE	[31]	RW	When set, disables page-hit logic in memory controller, forcing full PRE-RAS-CAS for every access. Set to 1 on cold or fast reset. Firmware must clear this bit to enable use of the ZBox page table.
Reserved	[30:25]	RW, MBZ	—
FORCE_NOCOP	[24]	RW	When set, forces nocop to have higher priority than READS during retire slot.
ENA_PREC	[23]	RW	Enables the slotter to use a COLC_PREC precharge packet.
ENA_PREX	[22]	RW	Enables the slotter to use a COLX-PREX precharge packet.
DRAIN_WRITE_CTL[1:0]	[21:20]	RW	Controls when to force write drains. Encoded as: **Bits** — **Meaning when set** 00 — Drain write timer disabled, and reset to 0 01 — Drain writes every 64 TCLKs 10 — Drain writes every 128 TCLKs 11 — Reserved
REFBIT_BNK[2:0]	[19:17]	RW	A 3-bit mask that corresponds to the BNK[5:3] address bits that are to be ignored during refresh (REFA/REFP) in support of multi-bank refresh. **REFBIT_BNK[2:0]** — **BNK[5:3]** 000 — No mask 100 — Mask BNK[5] 110 — Mask BNK[5:4] 111 — Mask BNK[5:3]
REF_BURST[3:0]	[16:13]	RW	The number of refresh commands in a burst. The number is encoded as REF_BURST[3:0] + 1
REF_INT[12:0]	[12:0]	RW	The number of Tpkts (each = 4 Rambus clocks) to wait between refresh intervals. The number of refreshes that will be serviced within a given TREF_INT interval is determined by REF_BURST as described in text following this table.

The refresh interval (expressed in TCLKs) is programmed by using REF_INT[12:0].

The formula is:

```
REF_INT[12:0] = MIN(uREF_INT, uRAS_INT) - 1
```

Where:

- uREF_INT is the "micro" refresh interval
- uRAS_INT is the constraint that guarantees all banks are precharged (due to refresh operations) such that the RDRAM tRAS,MAX parameter is satisfied
- uREF_INT = INT((.25 * tREF_T * nBURST) / (2**(b+r))
- uRAS_INT = INT((.25 * tRASMAX_T * nBURST))/(2**b)
- tREF_T is RDRAM refresh interval expressed as a number of TCLKs
- b = number of refresh bank bits (may not be equal to number of bank-address bits due to multibank refresh)
- r = number of row address bits
- nBURST = number of refresh operations/interval. This should be set to the number of REFP-REFA transactions that can be issued within the tRAS,MIN interval. This reduces the overhead of refresh activity
- tRASMAX_T is the RDRAM tRAS,MAX parameter expressed as a number of TCLKs
- A REF_INT value of 0 disables memory refresh.

The number of refresh operations issued each interval is programmed by means of REF_BURST[3:0]. This register serves as an offset, in that the actual burst length is 1 more than the programmed value. Legal range for the CSR is 0...15, yielding burst length range of 1...16.

16.6.9 DRAM Calibration Control 1 – ZBOX*n*_DRAM_CALIB_CTL1

There are two DRAM calibration control 1 registers; ZBOX0_DRAM_CALIB_CTL1 and ZBOX1_DRAM_CALIB_CTL1.

Figure 16–44 shows the DRAM calibration control 1 register.

Figure 16–44 DRAM Calibration Control 1

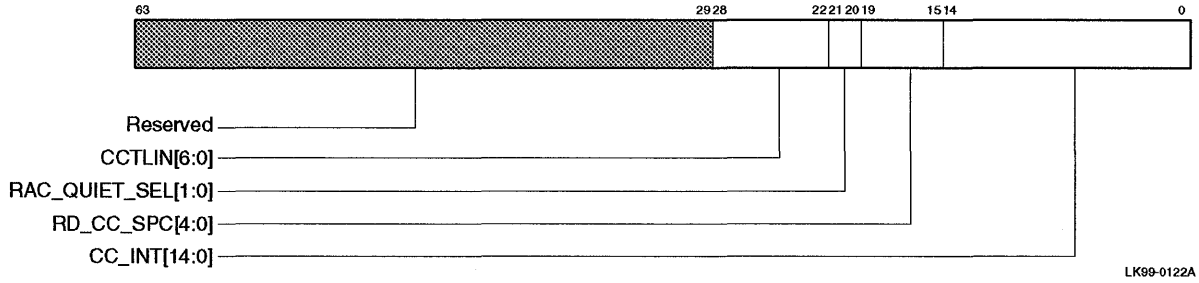


Table 16–69 describes the DRAM calibration control 1 register fields.

Table 16–69 DRAM Calibration Control 1 Fields Description

Name	Extent	Type	Description
Reserved	[63:30]	RW, MBZ	—
TC_INT[14:0]	[29:15]	RW	The number of refresh intervals between temperature calibrations.
CC_INT[14:0]	[14:0]	RW	The number of refresh intervals between current calibrations.

16.6.9.1 Temperature Calibration Interval

The Temp Calibrate interval (expressed in number of refresh intervals) is programmed by means of TC_INT[14:0]. It should be set to the number of intervals contained in one-half of the RDRAM's temperature calibrate (tTCAL) parameter. The one-half arises from the fact that each temperature calibration sequence consists of 2 commands, TCEN followed by TCAL, such that the TCAL is issued once per RDRAM tTCAL interval.

16.6.9.2 Current Control Interval

The current calibrate interval (expressed in number of refresh intervals) is programmed by means of field CC_INT[14:0]. It should be set to the number of refresh intervals in one RDRAM current calibrate interval, divided by the number of devices on the channel. The 21464 RAC performes its current calibration immediately following the calbiration of RDRAM device #0.

16.6.10 DRAM Calibration Control 2 – ZBOX*n*_DRAM_CALIB_CTL2

There are two DRAM calibration control 2 registers; ZBOX0_DRAM_CALIB_CTL2 and ZBOX1_DRAM_CALIB_CTL2.

Figure 16–45 shows the DRAM calibration control 2 register.

Figure 16–45 DRAM Calibration Control 2

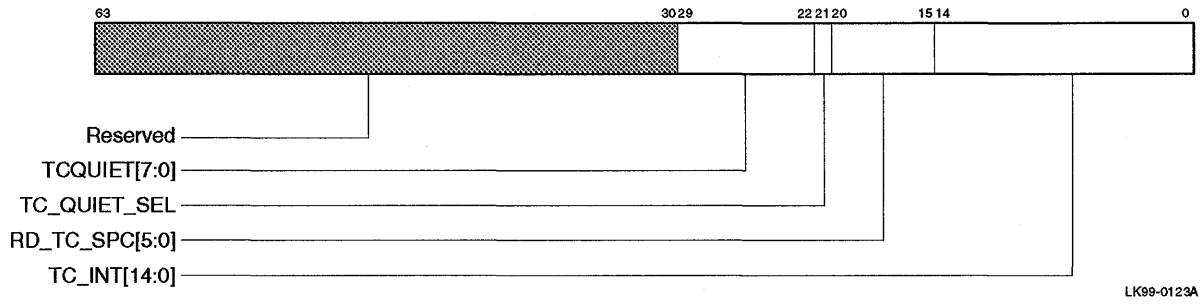


Table 16–70 describes the DRAM calibration control 2 register fields.

Table 16–70 DRAM Calibration Control 2 Fields Description

Name	Extent	Type	Description
Reserved	[63:30]	RW, MBZ	—
CCTLIN[6:0]	[29:23]	RW	Used for manual RAC current control update. The value of CCTLIN[6:0] is copied to the die bumps.
RAC_QUIET_SEL [1:0]	[22:21]	RW	This field selects the number of tPKTs (1 tPKT = 4 TCLKs) of quiet period (no Rambus activity) after performing current calibration of 21464 internal RACs.

RAC_QUIET_SEL[1:0]	Amount of additional delay
0	0
1	1 tPKT
2	2 tPKTs
3	4 tPKTs

RD_CC_SPC[4:0]	[20:15]	RW	This is the number of TCLKs of dead time that must be inserted between COLC=CC and COLC=RD packets.
TCQUIET[7:0]	[14:7]	RW	Number of read cycles prohibited after tcal.
TC_QUIET_SEL	[6]	RW	Enables optional all-quiet period after TempCal command is issued to RDRAMs. When set, the quiet period as specified by TCQUIET[7:0] causes ROW, COL and DATA buses not to be driven. When clear, the quiet period only applies to a period of prohibition of reads.
RD_TC_SPC[5:0]	[5:0]	RW	This is the number of TCLKs of dead time that must be inserted between COLC=RD and COLC=TC packets.

16.6.10.1 Read to Current Control Transition

Prior to executing a current calibration packet to a given device, that device may not have been read READTOCC TCLKs prior. The field that controls this quiet period is RD_CC_SPC. This CSR should be programmed to the RDRAM's READTOCC parameter.

16.6.10.2 Temperature Calibrate to Read transition

A gap must be injected on the Rambus to enforce the "quiet" period between temperature calibrate (TCAL) and the next read transaction. TCQUIET[7:0] controls the length of the quiet period. The formula for calculating TCQUIET[7:0] is:

```
tCAL + tTCQUIET - tPACKET - tCAC
```

16.6.10.3 Read to Temperature Calibrate transition

One must make sure that the the beginning of the tTCQUIET period is not violated by a prior read or CAL/SAM operation. To do this, the TCAL packet must be delayed to a certain point past the last previous READ or CAL/SAM packet. RD_TC_SPC[5:0] controls the length of this period. The formula for calculating RD_TC_SPC[5:0] is:

```
tCAC - tTCAL + tPACKET + tPACKET
```

16.6.11 DRAM Timing Control 4 – ZBOX*n*_DRAM_TIMING_CTL4

There are two DRAM timing control 4 registers; ZBOX0_DRAM_TIMING_CTL4 and ZBOX1_DRAM_TIMING_CTL4.

Figure 16–46 shows the DRAM timing control 4 register.

Figure 16–46 DRAM Timing Control 4

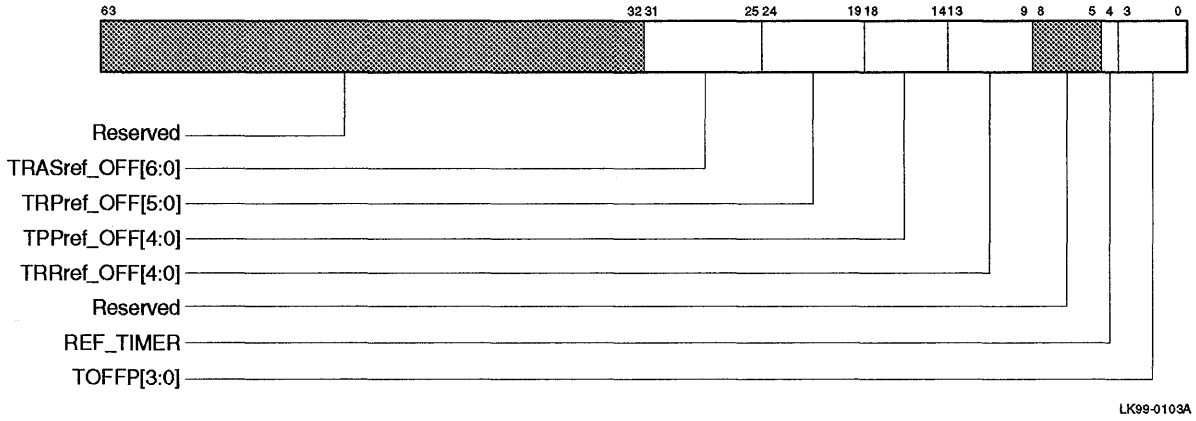


Table 16–71 describes the DRAM timing control 4 register fields.

Table 16–71 DRAM Timing Control 4 Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RO, MBZ	—
TRASref_OFF[6:0]	[31:25]	RW	Used to determine tRAS (RAS-PRE) during burst refresh. TRASref_OFF = Rambus tRASref
TRPref_OFF[5:0]	[24:19]	RW	Used to determine tRP (PRE-RAS) during burst refresh. TRPref_OFF = Rambus tRPref
TPPref_OFF[4:0]	[18:14]	RW	Used to determine tPP during burst refresh (PRE-PRE to same device). TPPref_OFF = Rambus tPPref
TRRref_OFF[4:0]	[13:9]	RW	Used to determine tRR during burst refresh (RAS-RAS to same device). TRRref_OFF = Rambus tRRref
Reserved	[8:5]	RW, MBZ	—
REF_TIMER	[4]	RW	If set, enables alternate parameters during refresh.
TOFFP[3:0]	[3:0]	RW	Specifies tOFFP value (COLX = PREX to "implied" ROWR = PRER)

16.6.12 DRAM Refresh Row – ZBOX*n*_DRAM_REFRESH_ROW

There are two DRAM refresh row registers; ZBOX0_DRAM_REFRESH_ROW and ZBOX1_DRAM_REFRESH_ROW.

Figure 16–47 shows the DRAM refresh row register.

Figure 16–47 DRAM Refresh Row

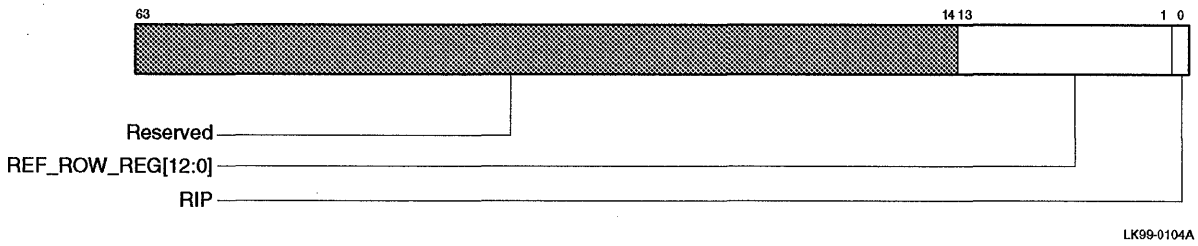


Table 16–72 describes the DRAM refresh row register fields.

Table 16–72 DRAM Refresh Row Fields Description

Name	Extent	Type	Description
Reserved	[63:14]	RO, MBZ	—
REF_ROW_REG[12:0]	[13:1]	RW	The next row to be refreshed – the refresh row number should be written to zero for the refresh all memory function (110 for CMD in DRAM_INIT_CTL).
RIP	[0]	RO	Refresh-all-mem_In_Progress status (polled to control PDN entry/exit).

16.6.13 DRAM Initialization Control – ZBOX*n*_DRAM_INIT_CTL

There are two DRAM initialization control registers; ZBOX0_DRAM_INIT_CTL and ZBOX1_DRAM_INIT_CTL.

Figure 16–48 shows the DRAM initialization control register.

Figure 16–48 DRAM Initialization Control

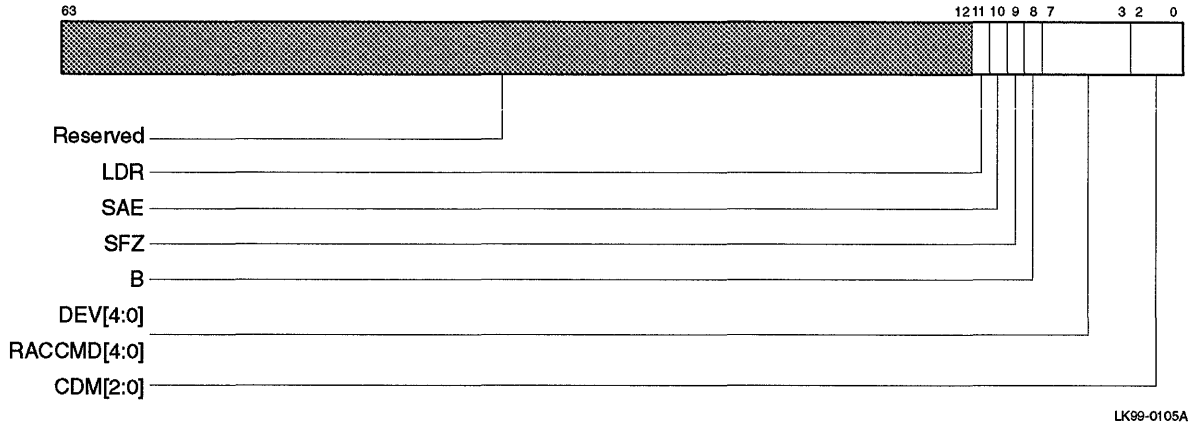


Table 16–73 describes the DRAM initialization control register fields.

Table 16–73 DRAM Initialization Control Fields Description

Name	Extent	Type	Description
Reserved	[63:11]	RO, MBZ	—
SAE	[10]	WO	Stop refresh-all-mem after refreshing bank=max, row=max
SFZ	[9]	WO	Start from bank=0, row=0 for Refresh-All-mem
B	[8]	WO	Broadcast to all devices. Cannot be used for ATTN
DEV[4:0]	[7:3]	WO	If non-RAC function (CMD!=0b111), device number targeted by CMD
RACCMD[4:0]	[7:3]	WO	If RAC function (CMD=0b111), encoded as follows:

Bits	Meaning	Bits	Meaning
0000	NOP	0001	NOP
0010	Assert PWRUP	0011	Deassert PWRUP
0100	Assert RESET	0101	Deassert RESET
0110	Reserved	0111	Reserved
1000	Assert CCTLAUTO	1001	Deassert CCTLAUTO
1010	Assert CCTLEN	1011	Deassert CCTLEN
1100	Assert CCTLLD	1101	Deassert CCTLLD
1110	Reserved	1111	Reserved

CMD[2:0] [2: 0] WO Function encoded as follows:

Bits	Meaning
000	NOP
001	TCAL device
010	TCEN device
011	Current calibrate
100	PDNR
101	ATTN
110	Refresh all mem
111	RAC Function

16.6.14 DIFT Control – ZBOX*n*_DIFT_CTL

There are two DIFT control registers; ZBOX0_DIFT_CTL and ZBOX1_DIFT_CTL.

Figure 16–49 shows the DIFT control register.

Figure 16–49 DIFT Control

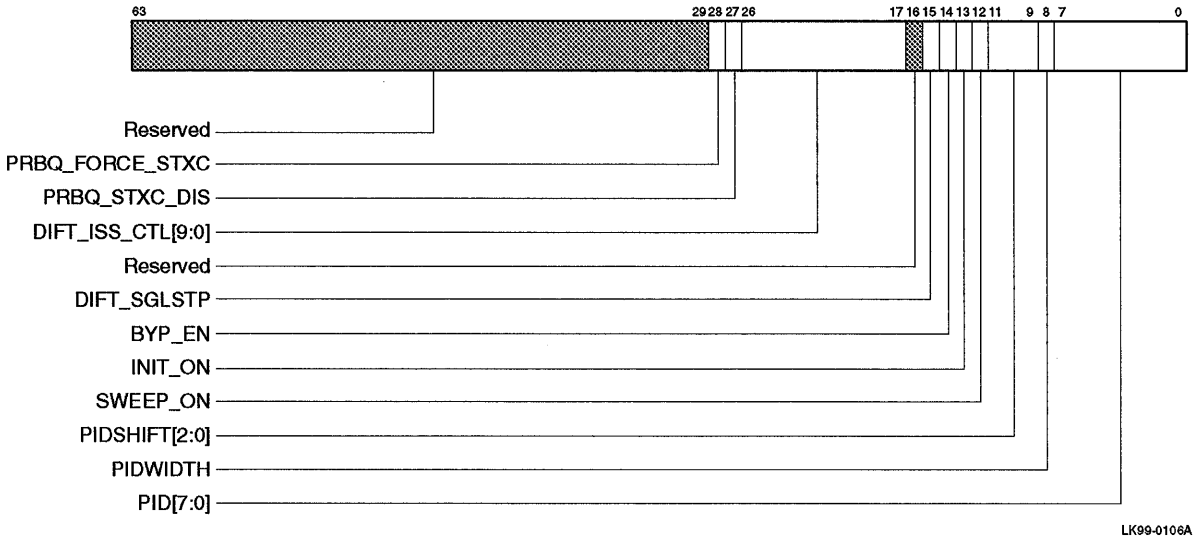


Table 16–74 describes the PID control register fields.

Table 16–74 PID Control Fields Description

Name	Extent	Type	Description
Reserved	[63:29]	RW, MBZ	—
PRBQ_FORCE_STXC	[28]	RW	Mimics functions of signal with same name in Cbox CSR and should be set to the same value.
PRBQ_STXC_DIS	[27]	RW	Mimics functions of signal with same name in Cbox CSR and should be set to the same value.
DIFT_ISS_CTL[9:0]	[26:17]	RW	Maximum value to initialize Softsnap counter.
Reserved	[16]	RW, MBZ	—
DIFT_SGLSTP	[15]	RW	When set, BeginQueue is stalled until the DIFT is idle (that is, all previous transactions have retired).
BYP_EN	[14]	RW	Bypass enable from allocation to ZRQ. This bit, when set, enables bypassing directly from DIFT allocation directly into the Zbox middle (ZRQ).
INIT_ON	[13]	RW	Put DIFT into init mode. INIT mode is used to initialize memory. When the Zbox is in this mode the processor is expected to submit inval_to_dirty requests to the zbox. The zbox ignores the current memory state and responds success to the inval_to_dirty command. Once the block is victimized, memory is initialized.
SWEEP_ON	[12]	RW	Put DIFT into sweep mode.
PIDSHIFT[2:0]	[11:9]	RW	Processor Shift value. Specifies how many bits to right shift the PID. Supported Range [0...4].
PIDWIDTH	[8]	RW	Processor ID width mode: **Pidwidth** **Mode** 0 6bit PID 1 8bit PID
PID[7:0]	[7:0]	RW	Processor ID value (loaded from module CSR)

16.6.15 DRAM Error Address – ZBOX*n*_DRAM_ERR_ADR

There are two DRAM error address registers; ZBOX0_DRAM_ERR_ADR and ZBOX1_DRAM_ERR_ADR.

Figure 16–50 shows the DRAM error address register.

Figure 16–50 DRAM Error Address

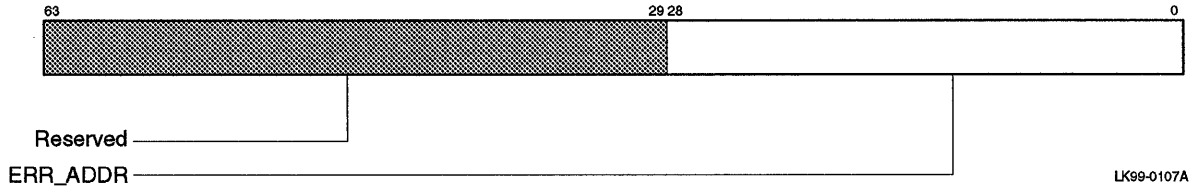


Table 16–75 describes the DRAM error address register fields.

Table 16–75 DRAM Error Address Fields Description

Name	Extent	Type	Description
Reserved	[63:29]	RO, MBZ	—
ERR_ADDR	[28:0]	RWAC	Memory Address of the first (more serious) error encountered. A correctable error address can be overwritten by an uncorrectable error address. (Memory Address is Physical Address with PID and byte-in-Cache_line address bits PA[5:0] removed.

16.6.16 DIFT Timeout – ZBOX*n*_DIFT_TIMEOUT

There are two DIFT timeout registers; ZBOX0_DIFT_TIMEOUT and ZBOX1_DIFT_TIMEOUT.

Figure 16–51 shows the DIFT timeout register.

Figure 16–51 DIFT Timeout

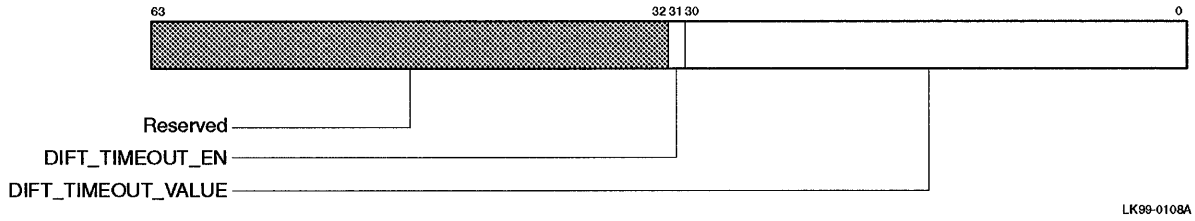


Table 16–76 describes the DIFT timeout register fields.

Table 16–76 DIFT Timeout Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RO, MBZ	—
DIFT_TIMEOUT_EN	[31]	RW	Enables DIFT timeout interrupts if set.
DIFT_TIMEOUT_VALUE	[30:0]	RW	Value to reload DIFT timer when it counts down to 0.

This register specifies a timeout value for the overall DIFT timer. This timer sends pulses to the 5 bit timers held with each DIFT entry. The value in this register specifies the period of the pulses.

When the N bit timer with a given DIFT entry cycles through all 2^N states, the timer expires. Each timer cycles to the next state with each pulse from the overall DIFT timer. This allows DIFT timeouts in the range of 2^6 to 2^{36} cycles.

The DIFT timer is reloaded when it counts to zero, or when DIFT_TIMEOUT_EN transitions from a 0 to a 1.

16.6.17 DRAM Mapper Control – ZBOX*n*_DRAM_MAPPER_CTL

There are two DRAM mapper control registers; ZBOX0_DRAM_MAPPER_CTL and ZBOX1_DRAM_MAPPER_CTL.

The system programmer is given the flexibility to extract Rambus BANK, DEVICE, ROW and COLUMN fields from the Memory Address (MA) to reduce the overall number of bank conflicts (that is, the unnecessary act of closing pages).

Physical → Memory Address Mapping

There are four modes of interpretation for the physical → memory address mapping, based on the small_addr and striped mode enable bits in the Cbox. ??????? describes and illustrates these modes, and the following table summarizes them.

small_addr	striped mode PA[36]	mem_adr[28:0]
0	0	MA[28:0] = PA[42] ! PA[33:6]
0	1	MA[28:0] = PA[42] ! PA[35:9] ! PA[6]
1	0	MA[28:0] = PA[42] ! PA[37] ! PA[35] ! PA[31:6]
1	1	MA[28:0] = PA[42] ! PA[33:32] ! PA[37] ! PA[35] ! PA[31:9] ! PA[6]

Figure 16–52 shows the DRAM mapper control register.

Figure 16–52 DRAM Mapper Control

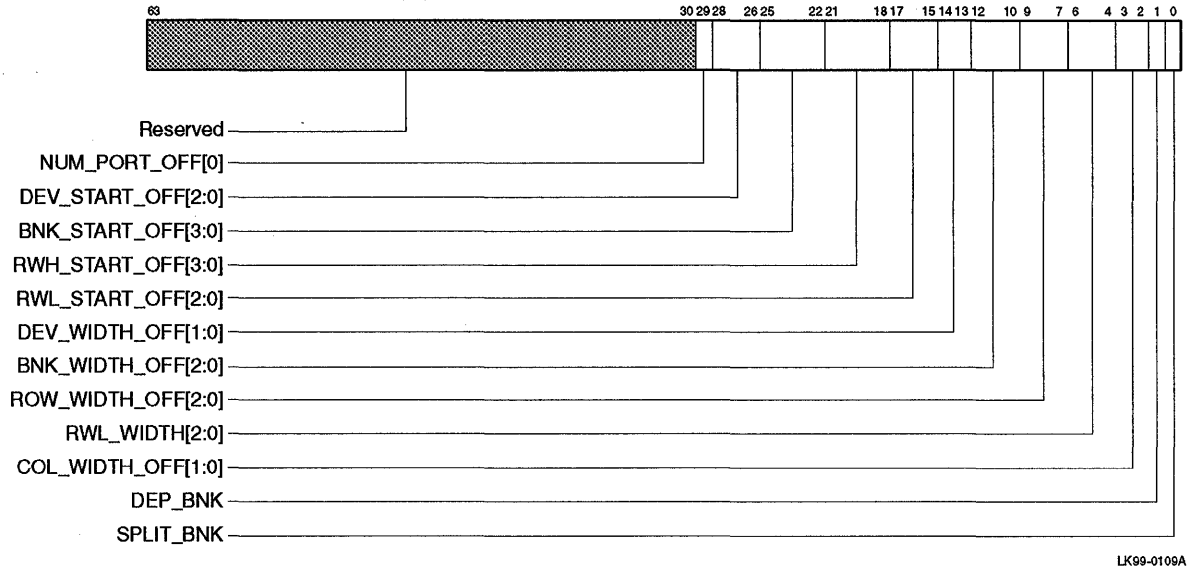


Table 16–77 describes the DRAM mapper control register fields.

Table 16–77 DRAM Mapper Control Fields Description

Name	Extent	Type	Description
Reserved	[63:30]	RW, MBZ	—
NUM_PORT_OFF[0]	[29]	RW	This offset value specifies the number of memory ports that will be active. If one port is specified (num_port_off=0), the colstart is implied to be MA[0]. If two ports are specified (num_port_off=1), the colstart is implied to be MA[1].
DEV_START_OFF[2:0]	[28:26]	RW	This offset value specifies the bit position within the MA to start the DEVICE field extraction. Valid ranges are 7...0, with the device start = MA[DEV_START_OFF+5]

DEV_START_OFF[2:0]	Dev_start
0	MA[5]
1	MA[6]
2	MA[7]
3	MA[8]
4	MA[9]
5	MA[10]
6	MA[11]
7	MA[12]

Table 16–77 DRAM Mapper Control Fields Description (Continued)

Name	Extent	Type	Description
BNK_START_OFF[3:0]	[25:22]	RW	This offset value specifies the bit position within the MA to start the BANK field extraction. The BANK field is extracted from the MA in reverse bit order (eg: B[0:n]) to minimize bank conflicts in dependent bank Direct RDRAM devices. Unlike the other start fields, BNK_START_OFF describes which MA bit to start B[0] extraction. Valid ranges are 14...0, with the bank start = MA[DEV_START_OFF+8]

BNK_START_OFF[3:0]	Bnk_start B[0]
0	MA[8]
1	MA[9]
2	MA[10]
3	MA[11]
4	MA[12]
5	MA[13]
6	MA[14]
7	MA[15]
8	MA[16]
9	MA[17]
10	MA[18]
11	MA[19]
12	MA[20]
13	MA[21]
14	MA[22]

RWH_START_OFF[3:0] [21:18] RW

This offset value specifies the bit position within the MA to start the high order ROW field extraction. This field must compensate for any low order ROW bits that have already been extracted from the row hole. If none of the row address bits are used in the row hole (rwl_width = 0), then the values of 0 and 10 correspond to bit positions MA[9] and MA[19] respectivly. The corresponding values increase by one for each bit taken into the hole. Valid ranges are 10...0, with RowN starting at bit position RWH_START_OFF+9+n. The following table shows the appropriate values. Figure 16–53 shows an intpretation of Row High.

	RWH_START_OFF										
RWL_WIDTH	0	1	2	3	4	5	6	7	8	9	10
0	9	10	11	12	13	14	15	16	17	18	19
1	10	11	12	13	14	15	16	17	18	19	20
2	11	12	13	14	15	16	17	18	19	20	21
3	12	13	14	15	16	17	18	19	20	21	22
4	13	14	15	16	17	18	19	20	21	22	—
5	14	15	16	17	18	19	20	21	22	—	—

The chart shows the initial high order bit to be extracted. For example if a three bit row hole (2) is found in the lower bits, then the "Three bit hole(R3)" is consulted. To begin the upper order extraction from bit position 22, a RWH_START_OFF of 4 is required.

Table 16–77 DRAM Mapper Control Fields Description (Continued)

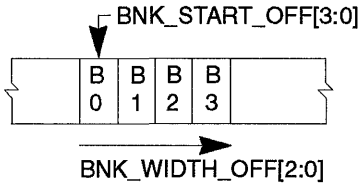
Name	Extent	Type	Description
RWL_START_OFF[2:0]	[17:15]	RW	This offset value specifies the bit position within the MA to start the low order ROW field extraction. Valid ranges are 4...0, with the row lower start = MA[5+RWL_START_OFF]
			RWL_START_OFF[3:0] Rwl_start B[0]
			0 MA[5]
			1 MA[6]
			2 MA[7]
			3 MA[8]
			4 MA[9]
DEV_WIDTH_OFF[1:0]	[14:13]	RW	This offset value specifies the number of bits to extract from the MA for the DEVICE field.
			DEV_WIDTH_OFF[1:0] Dev_width
			0 2b 4 devices
			1 3b 8 devices
			2 4b 16 devices
			3 5b 32 devices

Table 16–77 DRAM Mapper Control Fields Description (Continued)

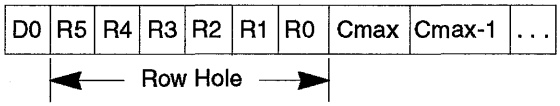
Name	Extent	Type	Description
BNK_WIDTH_OFF[2:0]	[12:10]	RW	This offset value specifies the number of bits to extract from the MA for the BANK field. The BANK field is extracted from the MA in reverse bit order (eg: B[0:n]) to minimize bank conflicts in dependent bank Direct RDRAM devices. The BNK_WIDTH_OFF field describes how many bits to extract to the right of the MA starting bit position. The BNK bits are extracted [LSB:MSB]. To enable 64 bank mode (BNK_WIDTH_OFF=4), software must guarantee that Dependent Bank mode is also enabled (DEP_BNK=1).

BNK_WIDTH_OFF[2:0]	Bnk_width
0	2b 4 banks
1	3b 8 banks
2	4b 16 banks
3	5b 32 banks
4	6b 64 banks

An example of how BANK is extracted from MA:



The system programmer is given the ability to extract lower order ROW bits from the MA between the DEVICE and COLUMN extraction points. This "row hole" is required if the DEVICE starting MA bit position does not fall exactly next to the COLUMN ending MA bit position, as shown in the following figure.



ROW_WIDTH_OFF[2:0]	[9:7]	RW	This offset value specifies the total number of row bits to extract from the MA (including those from the row hole). For instance, If row_width_off=0 (row size=9b) and rwl_width_off=5 (row hole=5b), then (9b-5b)=4b are extracted for the high order row. See table under BNK_WIDTH_OFF[2:0] to determine where the high order row bits are extracted based on the size of the row hole. The Rambus ROW packet protocol allows extensions for 12- and 13-bit rows by using bits defined as bank bits.

ROW_WIDTH_OFF[2:0]	Row_width
0	9b
1	10b
2	11b
3	12b [bank(5) = row(11)]
4	13b [bank(5,4) = row(11,12)]

Table 16–77 DRAM Mapper Control Fields Description (Continued)

Name	Extent	Type	Description
RWL_WIDTH[2:0]	[6:4]	RW	This value specifies the number of bits to extract from the MA for the low order ROW field. **RWL_WIDTH[2:0] — Rwl_width** 0 — 0b (no row hole) 1 — 1b 2 — 2b 3 — 3b 4 — 4b 5 — 5b
COL_WIDTH_OFF[1:0]	[3:2]	RW	This offset value specifies the number of bits to extract from the PA for the COLUMN field. **COL_WIDTH_OFF[1:0] — Col_width** 0 — 5b 1 — 6b 2 — 7b
DEP_BNK	[1]	RW	Dependent Bank Mode. When set, assumes dependent bank devices.
SPLIT_BNK	[0]	RW	Split Bank Mode. When set, assumes a split bank device. When SPLIT_BNK is set, DEP_BNK must also be set because a split bank device implies that the banks are dependent.

Figure 16–53 Interpretation of Row High

Figure TBS (LK99-0110A.WMF).

Need clarification on the ASCII figure below in order to create useful line art.

```
 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                          <---| rwh_start_off
                                  <-------| rwl_width
                                  ^ Row-High starting bit =
                                     MA<9+rwh_start_off+rwl_width>
```

16.6.18 Zbox Performance Counter 0 – ZBOX*n*_ZPM_CTR0

There are two Zbox performance counter 0 registers; ZBOX0_ZPM_CTR0 and ZBOX1_ZPM_CTR0.

This is a 31-bit event counter and an underflow bit. ZBOXn_ZPM_CTR0 can be programmed to count one of 32 items related to the Zbox middle.

The counter can be preloaded with an initial count via software. When the selected event occurs, the corresponding counter is decremented. When either counter counts below zero, the Zbox generates a performance_monitor interrupt.

Only the first underflow causes a performance_monitor interrupt, so that the interrupt can be disabled by writing a 1 to the underflow bit. The interrupt occurs on the 0→1 transition, therefore, #event–1 must be loaded into the counters.

Figure 16–54 shows the Zbox performance counter 0 register.

Figure 16–54 Zbox Performance Counter 0

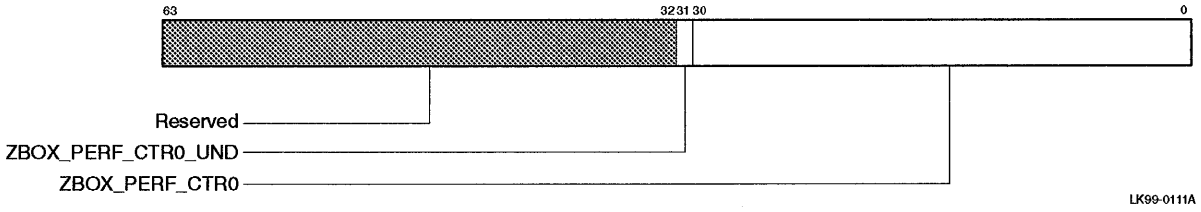


Table 16–78 describes the Zbox performance counter 0 fields.

Table 16–78 Zbox Performance Counter 0 Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RW, MBZ	—
ZBOX_PERF_CTR0_UND	[31]	RW	Indicates counter underflow.
ZBOX_PERF_CTR0	[30:0]	RW	Zbox Performance counter 0. Decrements when the condition specified by ZBOX_PERF_CTL[4:0] have been met. A performance counter interrupt is signalled when the counter underflows.

16.6.19 Zbox Performance Counter 1 – ZBOX*n*_ZPM_CTR1

There are two Zbox performance counter 1 registers; ZBOX0_ZPM_CTR1 and ZBOX1_ZPM_CTR1.

This is a 31-bit event counter and an underflow bit. ZBOXn_ZPM_CTR1 can be programmed to count one of 16 items related to the Zbox front-end (DIFT).

The counter can be preloaded with an initial count via software. When the selected event occurs, the corresponding counter is decremented. When either counter counts below zero, the Zbox generates a performance_monitor interrupt.

Only the first underflow causes a performance_monitor interrupt, so that the interrupt can be disabled by writing a 1 to the underflow bit. The interrupt occurs on the 0→1 transition, therefore, #event–1 must be loaded into the counters.

Figure 16–55 shows the Zbox performance counter 1 register.

Figure 16–55 Zbox Performance Counter 1

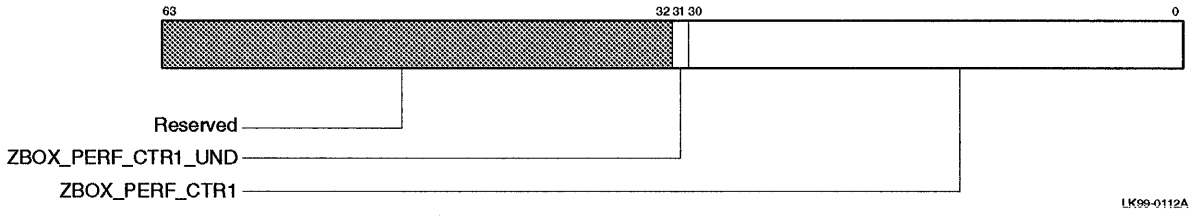


Table 16–79 describes the Zbox performance counter 1 fields.

Table 16–79 Zbox Performance Counter 1 Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RW, MBZ	—
ZBOX_PERF_CTR1_UND	[31]	RW	Indicates counter underflow.
ZBOX_PERF_CTR1	[30:0]	RW	Zbox Performance counter 1. Decrements when the condition specified by ZBOX_PERF_CTL[8:5] have been met. A performance counter interrupt is signalled when the counter underflows.

16.6.20 Zbox Performance Control – ZBOX*n*_ZPM_CTL

There are two Zbox performance control registers; ZBOX0_ZPM_CTL and ZBOX1_ZPM_CTL.

Figure 16–56 shows the Zbox performance control register.

Figure 16–56 Zbox Performance Control

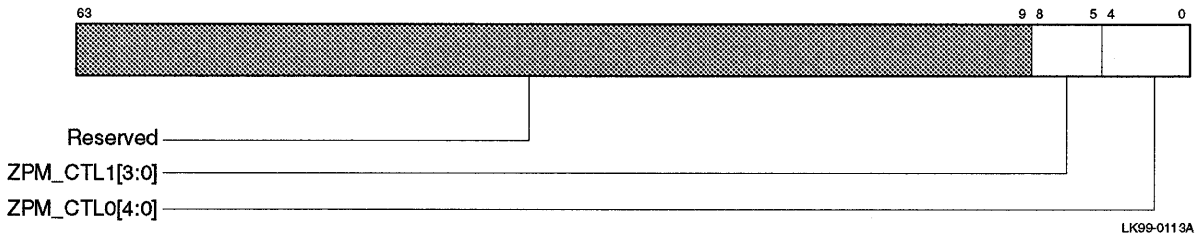


Table 16–80 describes the Zbox performance control fields.

Table 16–80 Zbox Performance Control Fields Description

Name	Extent	Type	Description
Reserved	[63:12]	RO,MBZ	—
Reserved	[11:9]	RW,MBZ	—
ZPM_CTL1[3:0]	[8:5]	RW	Control for Zbox Performance Counter 1.

CTL1	Item to count (ZPM_CTR1)
00000	nq_any – regardless of reject status
00001	nq_prq – regardless of reject status
00010	nq_rsq – regardless of reject status
00011	nq_csq – regardless of reject status
00100	nq_any – qualified by &! reject
00101	nq_prq – qualified by &! reject

Table 16–80 Zbox Performance Control Fields Description (Continued)

Name	Extent	Type	Description	
			00110	nq_rsq – qualified by &! reject
			00111	nq_csq – qualified by &! reject
			01000	nq_any – qualified by & reject
			01001	nq_prq – qualified by & reject
			01010	nq_rsq – qualified by & reject
			01011	nq_csq – qualified by & reject
			01100	nq_rej – No Fill Buffers available
			01101	nq_rej – Shadow Reject (tRR, tPP) in PDN or SHDPND Interval

Table 16–80 Zbox Performance Control Fields Description (Continued)

Name	Extent	Type	Description	
			CTL1	**Item to count (ZPM_CTR1)**
			01110	nq_rej – Page-Conflict Reject (tRP,tRCD) in PND or BLK-PND or NQPRPND Interval, (tRAS) in PND or HLDPND Interval (R/w), (tRDP) in SHDPND interval (R)
			01111	nq_rej – WRB Reject (tRTR, tRTP)
			10000	nq_rej – Queue Full Reject
			10001	nq-rej – NQ' Waterfall priority over DFT-NQ
			10010	cmd = dir_only_read
			10011	cmd = dir+data_read
			10100	cmd = dir_only_write
			10101	cmd = dir+data_write
			10110	PRER precharge
			10111	PREX precharge
			11000	PREC precharge
			11001	COL=RD
			11010	COL=WR
			11011	COL=NOCOP
			11100	COL=Any
			11101	Starvation detections
			11110	Force write retire
			11111	Deferred write retire
ZPM_CTL0[4:0]	[4:0]	RW	Control for Zbox Performance Counter 0.	
			CTL0	**Item to count (ZPM_CTR0)**
			0000	Incoming transaction (any)
			0001	Incoming ReadSharedReq
			0010	Incoming ReadModREq
			0011	Incoming ReadReq
			0100	Incoming FetchReq
			0101	Incoming SharedToDirtyReq
			0110	incoming SharedToDirtySTCReq
			0111	Incoming InvalToDirtyReq
			1000	Incoming Victim
			1001	Incoming VictimClean
			1001	Outgoing Forward (any)
			1010	Outgoing Forward=InvalSingle

Table 16–80 Zbox Performance Control Fields Description (Continued)

Name	Extent	Type	Description	
			CTL0	**Item to count (ZPM_CTR0)**
			1011	Outgoing Forward=InvalMask
			1100	Outgoing Forward=Read(anytype)Forward
			1101	Outgoing Forward=FetchForward
			1110	Outgoing Forward=ItoDForward
			1111	Forward Miss received

16.6.21 Zbox Sweep Directory Bits – ZBOX*n*_DRAM_SWEEP_DIR

There are two Zbox sweep directory bits registers; ZBOX0_DRAM_SWEEP_DIR and ZBOX1_DRAM_SWEEP_DIR.

Figure 16–57 shows the Zbox sweep directory bits register.

Figure 16–57 Zbox Sweep Directory Bits

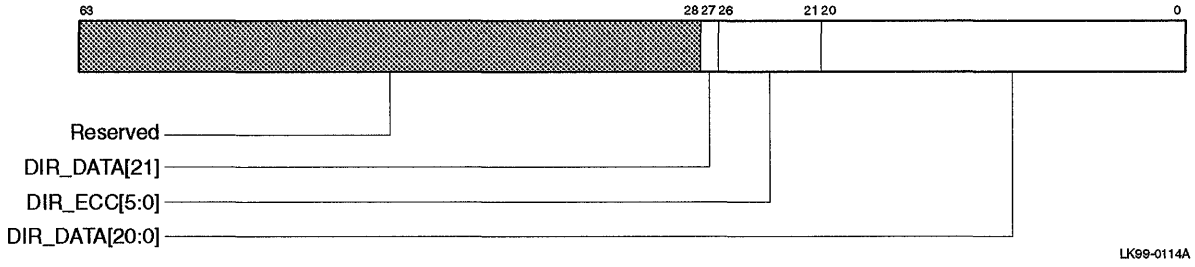


Table 16–81 describes the Zbox sweep directory bits fields.

Table 16–81 Zbox Sweep Directory Bits Fields Description

Name	Extent	Type	Description
Reserved	[63:28]	RO, MBZ	—
DIR_DATA[21]	[27]	RO	Contains directory entry data[21] from last read from memory. Valid only if SWEEP_ON is set. This bit is normally 0, and should be set only if the directory entry is written with the SET_DIR21 control bit set.
DIR_ECC[5:0]	[26:21]	RO	Contains directory entry ECC from last read from memory. Valid only if SWEEP_ON is set.
DIR_DATA[20:0]	[20:0]	RO	Contains directory entry data from last read from memory. Valid only if SWEEP_ON is set.

16.6.22 Zbox Force-Error Address register – ZBOX*n*_FRC_ERR_ADR

There are two Zbox force-error address registers; ZBOX0_FRC_ERR_ADR and ZBOX1_FRC_ERR_ADR.

The Zbox provides a means to force errors when a particular physical address is written. The address is specified in Rambus device, bank, row and column format. When the matching function is enabled (by means of DRAM_ERR_CTL[MAT_ERR_ENA]), the LSB of the Column address is cleared (as specified by DRAM_ERROR_CTL[FRC_WTERR]). The physical address match is on a cache-block granularity, with separate, independent registers for each Zbox. Use of this register requires knowledge of the address mapping scheme in use.

This register can also be used to provide a means to set directory_data[21] when an address match occurs (by means of DRAM_ERR_CTL[ADR_MAT_TRIGGER]). The Zbox can also optionally generate an interrupt (if DRAM_ERROR_CTL[ERR_INT_ENAB[6]] is set) upon reading a block of memory with directory_data[21]=1.

Figure 16–58 shows the Zbox force-error address registers.

Figure 16–58 Zbox Force-Error Address Register

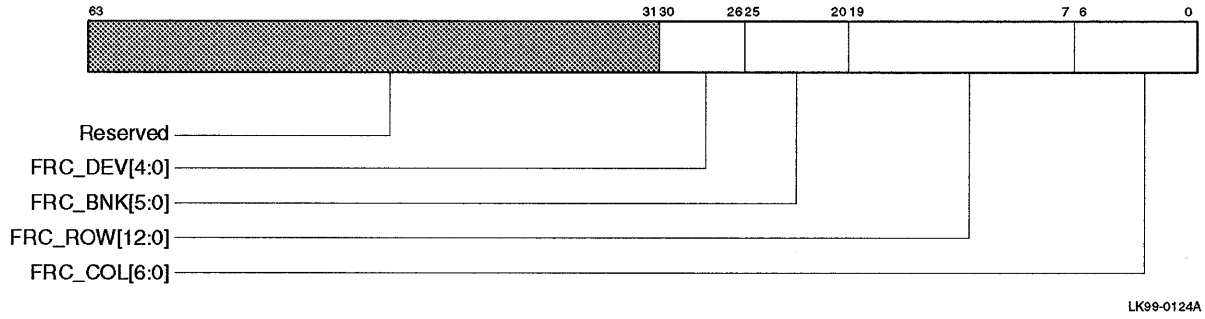


Table 16–82 describes the Zbox force-error address fields.

Table 16–82 Zbox Force-Error Address Fields Description

Name	Extent	Type	Description
Reserved	[63:31]	RW, MBZ	—
FRC_DEV[4:0]	[30:26]	RW	Rambus Device number
FRC_BNK[5:0]	[25:20]	RW	Rambus Bank number
FRC_ROW[12:0]	[19:7]	RW	Rambus Row number
FRC_COL[6:0]	[6:0]	RW	Rambus Column number

16.6.23 Zbox DIFT Error Status – ZBOX*n*_DIFT_ERR_STATUS

There are two Zbox DIFT error status registers; ZBOX0_DIFT_ERR_STATUS and ZBOX1_DIFT_ERR_STATUS.

Figure 16–59 shows the Zbox DIFT error status registers.

Figure 16–59 Zbox DIFT Error Status Register

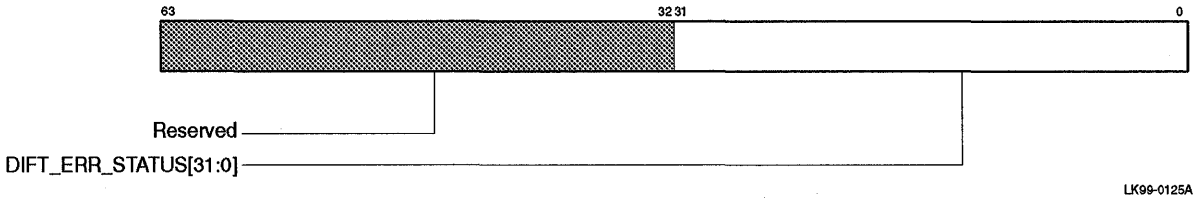


Table 16–83 describes the Zbox DIFT error status fields.

Table 16–83 Zbox DIFT Error Status Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RO, MBZ	—
DIFT_ERR_STATUS[31:0]	[31:0]	RWAC	Contains hardware error status bits from DIFT. For HW debugging only. A write clears all bits.
	[31:18]	MBZ	Reserved
	[17]		DIFT debit counter overflow
	[16]		DIFT debit counter underflow
	[15]		Block response to Rbox credit overflow
	[14]		Block response to Rbox credit underflow
	[13]		Non-block response to Rbox credit overflow
	[12]		Non-block response to Rbox credit underflow
	[11]		Forward to Rbox credit overflow
	[10]		Forward to Rbox credit underflow
	[9]		Protocol violation or unsupported case detected during DIR write
	[8]		DirWrite command logic error
	[7]		DirRead command logic error
	[6]		Simultaneous issue of Forward and Invalidate to FORWARD channel
	[5]		Simultaneous issue of DirRead and DirWrite to MEM channel
	[4]		Incoming ACK failed to merge to any DIFT entry
	[3]		New entry allocation failed due to empty freelist
	[2]		Unknown command decoded by packet accumulator
	[1]		Framing error on incoming packet from Rbox
	[0]		z_dft_acc->err_cbad_frame_a_h = framing error on incoming packet from Cbox

16.6.24 Zbox RAC Control – ZBOX*n*_RAC_CTL

There are two Zbox RAC control registers; ZBOX0_RAC_CTL and ZBOX1_RAC_CTL.

Figure 16–60 shows the Zbox RAC control registers.

Figure 16–60 Zbox RAC Control Register

Figure TBS when bits are defined.

Table 16–84 describes the Zbox RAC control fields.

Table 16–84 Zbox RAC Control Fields Description

Name	Extent	Type	Description
Reserved	[63:32]	RO, MBZ	—
RAC_CTL[31:0]	[31:0]	RW	Contains control bits for RAC. No bits are currently defined.

17
Privileged Architecture Library Code

17.1 HW_LD and HW_ST Instructions

PALcode uses the HW_LD and HW_ST instructions to access memory outside the realm of normal Alpha memory management and perform special Dstream load and store transactions. The data conversions are identical to byte, word, long and quad integer counterparts.

Data alignment traps are disabled for all forms of the HW_LD and HW_ST instructions and the effective address is forced to match the specified data size.

The instruction format of the HW_LD and HW_ST instructions is:

Figure 17–1 HW_LD/HW_ST Instruction Format

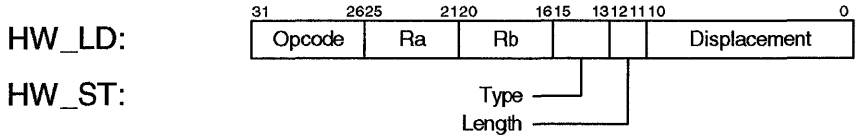


Table 17–1 HW_LD/HW_ST Instruction Fields Description

Field Name	Extent	Description
Opcode	31:26	The instruction Opcode. 0x1B HW_LD 0x1F HW_ST
Ra	25:21	The destination register number for loads or the write data for stores
Rb	20:16	Source register which holds the base address of the operation.

Table 17–1 HW_LD/HW_ST Instruction Fields Description (Continued)

Field Name	Extent	Description
Type	15:13	Type of memory reference to perform. The /PTE and /WrChk operations are valid only for HW_LD operations.

Bits Set	Type of Reference	Meaning
000	Physical	The effective address for the HW_LD/ST instruction is physical, not virtual.
010	Virtual/PTE	Valid only for HW_LD, used to fetch page table entries from memory. TB faults vector directly to the double-miss flows. Kernel mode access checks are performed.
100	Virtual	The address virtual. How is this different from LD/ST? Alignment checks are disabled.
101	Virtual/WrChk	The effective address for a HW_LD instruction is virtual. Access checks for fault-on-read, fault-on-write, read and write are performed
110	Virtual/Alt	Same as Virtual but the ALT field of the M_MODE register is used for access checking.
111	Virtual/WrChk/Alt	Same as Vitrual/WrChk but the ALT field of the M_MODE register is used for access checking.

Length	12:11	The size of the data transaction. Data alignment checks are not performed but alignment is forced to the data size.

Bits Set	Meaning
00	Byte Access
01	Word Access
10	Longword Access
11	Quadword Access

Disp	10:0	An 11-bit signed displacement that is added the value in Rb to form the effective address of the load or store.

17.2 HW_MFPR and HW_MTPR Instructions

PALcode uses the HW_MFPR and HW_MTPR instructions to access the internal processor registers. The HW_MFPR instruction reads the value from the specified IPR into the integer register specified by the Ra field. The HW_MTPR instruction writes the value from the integer register specified by the Rb field into the specified IPR.

17.2.1 HW_MFPR Instruction

The instruction format of the HW_MFPR instruction is:

Figure 17–2 HW_MFPR Instruction Format

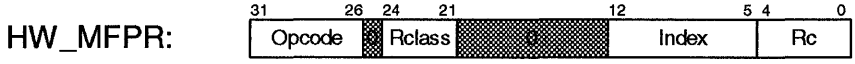


Table 17–2 HW_MFPR Fields Description

Field Name	Extent	Description
Opcode	31:26	The instruction Opcode: 0x19
Rc	4:0	Destination integer register.
Index	12:5	Identifier of the IPR to read. See the IPR table for a complete list of indexes. The MSB of the Index field differentiates between IPRs located in the Mbox and IPRs located in the Ibox. MSB = 0 Mbox MSB = 1 Ibox
Rclass	24:21	Reader class of the instruction. The reader class defines an dependency against a previous IPR writer of the same class. The reader will not issue until the writer dependency has cleared. The format of the reader class field is as follows: **Bits** **Description** 3 Valid bit. If clear, no dependency exists 2:0 Class number The currently defined/allowed values for reader class are: **Bits Set** **Meaning** 0XXX No dependency 1XX0 Dependency against an IPR writer class of 0 1XX1 Dependency against an IPR writer class of 1

17.2.2 HW_MTPR Instruction

The instruction format of the HW_MTPR instruction is:

Figure 17–3 HW_MTPR Instruction Format

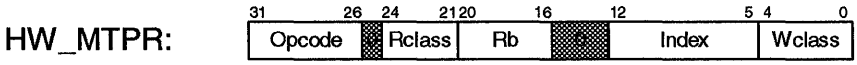


Table 17–3 MT_MTPR Instruction Fields Description

Field Name	Extent	Description
Opcode	31:26	The instruction Opcode: 0x1D.
Rb	20:16	Source integer register.
Index	12:5	Identifier of the IPR to read or write. See the IPR table for a complete list of indicies. The MSB of the Index field differentiates between IPRs located in the Mbox and IPRs located in the Ibox. MSB = 0 Mbox MSB = 1 Ibox
Rclass	24:21	Reader class of the instruction. The reader class defines an dependency against a previous IPR writer of the same class. The reader will not issue until the writer dependency has cleared. The format of the reader class field is as follows: **Bits** **Description** 3 Valid bit. If clear, no dependency exists 2:0 Class number The currently defined/allowed values for reader class are: **Bits Set** **Meaning** 0XXX No dependency 1XX0 Dependency against an IPR writer class of 0 1XX1 Dependency against an IPR writer class of 1

Table 17–3 MT_MTPR Instruction Fields Description (Continued)

Field Name	Extent	Description
Wclass	4:0	Writer class of the instruction. The writer class defines the source of a reader class dependency. HW_MTPR instructions that define a writer class create an issue dependency that must be cleared before any IPR reader (MFPR or MTPR) of the same class can issue. The dependency is cleared when the writer issues unless the bubble-bit is set, when the bubble-bit is set, the dependency does not clear until a bubble acknowledgement is received for the writer. The general format of the writer class field is: **Bits** — **Description** 4 — Bubble bit - If set, issue logic waits for notification 3 — Valid bit. - If clear, no writer dependency is set. 2:0 — Class number. The currently defined/allowed values for writer class are: **Bits Set** — **Meaning** X0XXX — No dependency 01XX0 — Set dependency for IPR writer class 0 01XX1 — Set dependency for IPR writer class 1 11XX0 — Set completion buble dependency for IPR writer class 0 11XX1 — Set completion buble dependency for IPR writer class 1 Completion bubble dependencies are only created for HW_MTPR instructions that target the Mbox IPRs. If the MSB of the Index field is set, indicating an Ibox register target, the bubble bit is ignored and an issue dependency is created.

17.3 Execution of the RET Instruction in PALmode

The special PALmode HW_RET instruction that was implemented in the 21264 is not supported by the 21464. Instead, the normal RET instruction is used to return instruction flow to a specified PC and to exit PALmode and SuperPALmode.

The RET instruction Rb field specifies an integer general-purpose register (GPR) that holds the target PC. GPR[1:0] specifies the new value of PALmode after the RET is executed, as follows:

Table 17–4 GPR[1:0] Encoding

Value	Meaning
00	Normal mode
01	PALmode
11	SuperPALmode

The only exception is that Normal mode RET instructions *cannot* cause a transition into PAlmode or SuperPALmode. Only a CALL_PAL instruction, an interrupt, or a trap condition can elevate the mode from Normal mode to PAlmode, and only a PNMI event

can cause a transition from Normal mode to SuperPAlmode. The implementation actually allows PALmode code to transition to SuperPALmode by using the RET instruction. It is not clear why PALcode would ever do that.

Table 17–5 RET Instruction Mode Transitions

Old Mode	New Mode		
GPR[1:0]	Normal mode	PALmode	SuperPALmode
Normal mode	RET	CALL_PAL/Trap/Interrupt	
PALmode	RET	RET	PNMI/RET
SuperPALmode	RET	RET	RET

In a RET to Native mode, the Rb field is likely to be a PALcode shadow register. In a RET to PALmode, the register may or may not be a PALcode shadow register. It is expected that Ra field of the RET will usually be R31.

Normally, the RET instruction succeeds a CALL_PAL instruction, an exception entry, or a BSR subroutine call from within PALmode. Those cases push the return PC onto the prediction stack and subsequently pop that stack to generate a predicted target address. That address is always predicted Native mode, regardless of the circumstances of the push onto the prediction stack and, therefore, all returns to PALmode incur a mispredict.

Figure 17–4 RET Instruction Fields

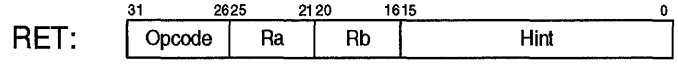


Table 17–6 RET Instruction Fields Description

Name	Extent	Description
Opcode	31:26	The instruction Opcode. 0x1A
Ra	25:21	Receives the PC of the instruction following the RET.
Rb	20:16	Holds the target PC of the RET.
Hint	15:0	Return predictor stack hints. See Section (I) 4.3.3 of the Alpha SRM for a complete description.

17.4 CMOV Execution Within PALcode

Because the shadow register replacement process in PALmode is keyed to different registers numbers for Rb and Rc, the 21464 does not correctly replace the inserted reference to Rc for native CMOVxx1 instructions in PALmode.

Legacy CMOV instructions in PALcode are special cased to disable all replacements of Rc/Fc. This allows PALcode to modify the architectural registers R24/F24 and R25/F25 without requiring a special mode to control the PALcode shadow replacement process.

The general coding rule is that shadow registers cannot be used as the destination of either a legacy or native CMOV instruction in PALmode. If PALcode needs to modify architectural registers R24/F24 or R25/F25, it must do so in PALmode by using the legacy version of CMOVxx/FCMOVxx. Other uses of either legacy or native CMOV instructions in PALmode are allowed.

See Section 2.11.2.5 for complete information about CMOV instruction execution.

17.5 PALcode Restrictions and Guidelines

Open questions:

1. Is Restriction 5 necessary? Clarification needed.
2. Is Restriction 6 necessary? Should we doubly map all elements of the DTBWINQ group? This would allow us to avoid the DTBWINQ mechanism if there is a bug in it, an IFETCHB would be required in the single miss flow.
3. For Restriction 7, clarify how back-to-back single misses work.
4. For Restriction 16, why is the IFETCHB necessary in the flow?

17.5.1 Restriction 1: PALcode Must Guarantee That IPR Writes Retire Before Returning

Use the IFETCHB instruction to guarantee IPR write data is committed before instructions that depend on the IPR value are allowed to proceed. In general, all PALcode flows that write IPRs have an IFETCHB instruction after the last IPR write before returning.

Exception:

The DTB writer block in DTBM_SINGLE is protected through the DTB Writer In Queue (DTBWINQ) interlock logic. In that case, an IFETCHB is not necessary.

17.5.2 Restriction 2: IFETCHB Required Between IPR Writes in the Same IPR Group

There can be at most one in-flight good-path IPR Write for each TPU to each IPR group.

IPR writes are speculatively issued but not committed until the HW_MTPR instruction retires. The internal storage that holds the speculative value is shared among all IPRs in a group, so take care to ensure another IPR write does not attempt to overwrite the speculative storage before the first writer retires. The 21464 interlocks the speculative register that grants access to the oldest writer, which ensures that random bad-path code does not alter the speculative value before it is written. However, if a younger good-path IPR write is issued before an older good-path IPR write in the same speculative group, the younger IPR write might get the data of the older write.

PALcode must separate writes to IPRs in the same speculative group with an IFETCHB instruction.

17.5.3 Restriction 3: Mbox IPRs Must be Written Twice to Ensure Correct Slot-

ting

Mbox IPRs must be written twice by consecutive instructions in the same fetch block.

Mbox IPR write data is communicated to the Mbox by using the primary address busses. Mbox IPR write instructions that slot to an odd position utilize bus P0 and instructions that slot to an even position utilize bus P1.

Mbox IPRs consist of two groups:

- Mbox IPRs in speculative group M1 only connect to bus P0 and can, therefore, only be written by instructions that slot to odd positions. Writing these IPRs in consecutive instructions in the same fetch block guarantees that one of the writes is slotted in an odd position.

  Exception:

  Mbox IPRs in speculative group M1 can be written with only a single write if care is taken to use the map-block alignment instruction to guarantee that the write is slotted in an odd position.

- Mbox IPRs in speculative groups M2 and M3 have two copies of each IPR, one connected to bus P0 and one connected to bus P1. Writing these IPRs in consecutive instructions in the same fetch block guarantees that one write is slotted for each bus and both copies are updated.

If the instructions were allowed to span a fetch block, the second fetch block could Icache miss, allowing the instructions to possibly map into separate blocks. Without forced alignment, the last instruction in the first block has a 50 percent chance of being even aligned and, since the first instruction of the second block is guaranteed to be even aligned, the rule could be violated.

17.5.4 Restriction 4: All Instructions in the DTB Writer Block Must be in the Same Map Block

The DTB Writer Block consists of the following instructions:

```
HW_MTPR S0 -> DTB_TAG, R#1, W#0
HW_MTPR S0 -> DTB_TAG, R#1, W#1
HW_MTPR S1 -> DTB_PTE, R#0
HW_MTPR S1 -> DTB_PTE, R#1, W#1, BB
HW_MFPR DTBMS_RET_ADDR -> S1
NOP
NOP
ALIGN_NOP
```

These instructions must be in the same map block beause the DTB Writer In Queue (DTBWINQ) interlock logic assumes that all members of the block are allocated into the queue in the same cycle.

The ALIGN_NOP instruction must be in position 7 of a fetch chunk so these eight instructions are also guaranteed to be in the same fetch block.

17.5.5 Restriction 5: All Four DTB MTPR Instructions Must Appear in the Same

Fetch Block

If any MTPR to DTB_TAG or DTB_PTE is in a fetch block, all four MTPRs must be in that fetch block.

All four MTPR instructions are necessary to write the DTB. These instructions cannot be separated:

HW_MTPR S0 -> DTB_TAG, R#1, W#0
HW_MTPR S0 -> DTB_TAG, R#1, W#1
HW_MTPR S1 -> DTB_PTE, R#0
HW_MTPR S1 -> DTB_PTE, R#1, W#1, BB

.*** What hardware case caused this restriction? Assuming Restriction 3 is obeyed, the TAG and PTE writes are correctly paired. Why must the TAG and PTE writes be in the same fetch chunk?

17.5.6 Restriction 6: Non-DTB Writer Block DTBMS_RET_ADDR MFPRs Require IFETCHB

If an MFPR From DTBMS_RET_ADDR appears in a PALcode Flow, there must be an IFETCHB before the end of that PALcode flow.

MFPRs from DTBMS_RET_ADDR are considered part of the DTB Writer In Queue (DTBWINQ) interlock mechanism. Therefore, PALcode must guarantee that a read from DTBMS_RET_ADDR retires before leaving PALmode.

Exception:

If the MFPR from DTBMS_RET_ADDR is part of the writer block in a flow that is protected by the DTBWINQ mechanism (that is, DTB Miss Single), the IFETCHB is not necessary

*** Is this restriction necessary? We have provided a second mapping for DTBMS_RET_ADDR that has no side effects. What is the impact of not including the IFETCHB? I assume it has to do with the MFPR for the DTBMS_RET_ADDR creating a writer block and somehow effecting the writer block of a subsequent DTB_MISS? We need to clarify this, and explain how normal back-to-back DTB misses do not have this problem.

17.5.7 Restriction 7: IFETCHB Required Between Non-DTB Writer Block DTB Writer Block MxPRs

If any DTB writer block instruction appears in a PALcode flow, an IFETCHB is necessary before any subsequent fetch block that contains DTB writer block instructions.

Since the DTB writer block instructions are part of the DTB Writer In Queue (DTB-WINQ) mechanism, PALcode must guarantee that only one DTB Writer Block can be in flight at a time. Therefore, PALcode must issue an IFETCHB between fetch blocks containing DTB Writer instructions.

*** How does PALcode accomplish this? How can it know that back-to-back DTB_SINGLE flows are separated by a IFETCHB?

17.5.8 Restriction 8: Padding Required Between DTB Writer Block and DTB-

Dependent Instructions

DTB-dependent instructions must not be allowed to map the cycle after a DTB writer sequence maps.

The implementation does not recognize load or store instructions that allocate into the Instruction Queue in the cycle immediately following the DTB Writer Block as DTB-dependent. Those instructions could thus issue before the DTB Writer Block has modified the speculative DTB entry or even left the IQ, causing a second DTB miss before the first has been satisfied.

Padding the DTB writer flow with enough NOPs to fill the succeeding map block guarantees that memory operations after the DTB writer flow are not allocated on the cycle following the DTB Writer Block.

Note: This issue should only apply to the DTBM_SINGLE_CONS and DTBM_SINGLE flows because they are the only flows expected to rely on the DTBWRT interlock mechanism.

17.5.9 Restriction 9: PALcode Must Not Allow Writes INVALID DTB_PTE Entries to Retire

PALcode must explicitly check the value being written to the DTB PTE and ensure bit<0> is set. If bit<0> is not set, PALcode must branch away. Since the 21464 predicts all branches in PALmode as not-taken, the MTPR speculatively issues, but is killed when the branch resolves.

The valid bit of the DTB_PTE entry is used as a completion condition. Invalid PTE entries do not complete and, therefore, preserve the DTBWRT interlock. If a invalid DTB_PTE write was not killed and attempted to retire, the machine would hang.

Note: Writing R31 to the DTB_PTE has a slightly different effect. The 21464 completes the write of R31, so the machine does not hang, but the DTB-WRT block is also lifted and subsequent loads and stores are allowed to issue.

17.5.10 Restriction 10: TAG and PTE Must be Written as Pairs with TAG Writes Before PTE Writes

The TAG (DTB_TAG/ITB_TAG) and PTE (DTB_PTE/ITB_PTE) must be written together with the tAG being written before the PTE.

The TAG and PTE IPRs represent two fields of a single PTE and, therefore, updates must be atomic — hence the pairing requirement. Also, the PTE field might contain granularity hint information that modifies the contents of the TAG field; therefore, the updates must occur in the order of TAG, and then PTE. Otherwise the TAG value might not correctly reflect the granularity hint data in PTE, potentially causing multiple matches and electrical contention in the TB.

To ensure ordering of the writes, the write the PTE must be Reader Class dependent on the write to the TAG.

17.5.11 Restriction 11: Register-Dependent MTPRs Must Not Have Read Class

Dependent MxPRs

HW_MTPR and HW_MFPR instructions must never have reader class issue dependencies on any HW_MTPR possessing a register dependency (direct or indirect) on a load unless they share the same register dependency.

The reason for this rule is poison, because poison is communicated through physical register dependencies, while Reader/Writer Class dependencies are independently transmitted through INum dependencies (which are translated into queue entry dependencies in the Instruction Queue).

Consider the following instruction sequence:

```
LD (S1) -> S0
HW_MTPR S0 -> IPRx, W#0
HW_MTPR S1 -> IPRy, R#0
```

If the load misses, the HW_MTPR from S0 is poisoned. In principal, the HW_MTPR from S1 should also be poisoned, but isn't, because it has no physical register dependency on the first HW_MTPR. It is free to issue, and may do so long before the load is satisfied and the first HW_MTPR is replayed. The two HW_MFPRs effectively issue out of order, violating the Reader/Writer class dependency.

Recall that writes to the speculative DTB entry must occur in the order of DTB_TAG, and then DTB_PTE. Now consider this hypothetical PALcode flow:

```
LD (S1) -> S0
HW_MTPR S0 -> DTB_TAG, W#0
HW_MTPR S1 -> DTB_PTE, R#0
```

If the load misses, the MBox ignores the poisoned write to DTB_TAG, allows the write to DTB_PTE, and then allows the second write to DTB_TAG, possibly causing the speculative DTB entry to be incorrect.

Note that if the value used in the HW_MTPR to DTB_TAG is sourced by a HW_MFPR, the HW_MTPR cannot be poisoned. This is the case in the DTBM_SINGLE flow.

Also note that this rule does not apply to Reader/Writer Class retire dependencies, because any poison cases have been resolved by the time the HW_MTPR makes its retire-time bubble.

If there must be a Reader Class dependency on an MTPR that has a register dependency on a load, an artificial register dependency can be created so that if the MTPR is poisoned, the Reader Class dependent MTPR or MFPR is also poisoned. For example:

```
LD (S1) -> S0
XOR S1, S0 -> S1
XOR S1, S0 -> S1
HW_MTPR S0 -> IPRx, W#0
HW_MTPR S1 -> IPRy, R#0
```

17.5.12 Restriction 12: CMOV instructions Cannot Specify PALcode Shadow Registers as Destinations

Because of the shadow register overlay rules and the way a CMOV is split into two instructions, PALcode shadow registers cannot be used as the destination (Rc) of any CMOV instruction.

Legacy CMOV instructions in PALmode are special cased to disable shadow register replacements of Rc. This allows PALcode to use a legacy CMOV to modify the architectural registers R24 and R25 without requiring a special mode to disable the PALcode shadow replacement process.

17.5.13 Restriction 13: PALmode Native CMOV Instructions Cannot Specify R24 or R25 as Destinations

The 21464 does not have a "mode" bit to enable/disable shadow register replacement. Instead, the 21464 implements the following position-dependent replacement policy.

	R22	R23	R24	R25
Operand A	—	—	S0	S1
Operand B	S0	S1	—	—
Destination	—	—	S0	S1

The SrcA and SrcB operands have different mappings for shadow registers to make it easy for PALcode to read from any register, shadow or architectural. The SrcA operand and the destination have the same mapping to ensure the correct handling of STx_C, where Ra specifies both SrcA and Destination.

For Native CMOV instructions, the 21464 replaces the original instruction:

```
CMOVxx Ra, Rb -> Rc
```

With:

```
CMOVx1 Ra, Rc -> Rc
CMOV2  Rc, Rb -> Rc
```

Because the SrcB operand is keyed to a different replacement policy than the destination, the CMOVx1 part of the instruction sequence fails to replace the SrcB operand with a the correct shadow register.

If PALcode needs to modify architectural registers R24 or R25, it must do so in PALmode by using the legacy version of CMOVxx. Other uses of legacy style CMOV instructions in PALmode are allowed.

Other uses of native the 21464 style CMOV instructions in PALmode are allowed.

17.5.14 Restriction 14: PALmode JMP Instructions Must be Followed by IFETCHB

To provide PALmode with a nonspeculative jump instruction, all JMP instructions in PALmode predict to the next instruction in the flow and always cause a jump mispredict trap.

For example, the following code sequence in PALmode behaves as follows.

```
JMP    S1
IFETCHB
```

The JMP predicts to the IFETCHB, which prevents further speculation by inhibiting fetching. When the JMP issues, it causes a jump mispredict trap, which causes a kill and redirects the machine to the true jump target.

Without the trailing IFETCHB, the 21464 speculates past the JMP, possibly leading to a trap on the bad path and corruption of implicitly instruction-written IPRs.

Note: An IFETCHB after a JMP does not satisfy the requirement for an IFETCHB prior to a return from PALcode (Restriction 2). The IFETCHB that follows a PALmode JMP never reaches its retire point and is killed by the JMP mispredict. It cannot serve as the required retire barrier since it is guaranteed to be on the bad path.

Also note that this rule applies only to JMP, not other jump instructions (i.e. JSR, JSR_COROUTINE, and RET).

17.5.15 Guideline 15: No Push or Pop Instructions in the First Fetch Block of a PALmode Flow

During the cycle when the first eight instructions of a PALmode flow are fetched, the Ibox is busy writing the trap return address to the return stack and cannot write the return address for the push (BSR, JSR, JSC, CALL_PAL) or pop (RET, JSC) instruction.

Violating this rule degrades performance because the stack order is broken and future return instructions are almost guaranteed to mispredict.

Doing a superfluous PUSH without a corresponding RET, such as a BSR to the next instruction, can repair the damage to the return stack. The first return mispredicts, but the rest of the return stack is not corrupted.

17.5.16 Restriction 16: PALmode MT_FPCR Must be Followed by IFETCHB

PALcode must guarantee the retirement of any MT_FPCR instruction in PALcode prior to the return of control to native code, and prior to the issuing of another PALmode MT_FPCR or MF_FPCR, or any PALmode instructions that implicitly read the FPCR.

The Floating-Point Control Register (FPCR) is a special form of Speculative-Committed IPR (SCIPR); the speculative value is only committed to the architectural FPCR when the MT_FPCR instruction that wrote it becomes retireable. The FPCR is explicitly read by an MF_FPCR instruction and implicitly read by all floating-point instructions.

** Peter: What makes it special? All SCIPRs commit when retirable.

MT_FPCR instructions in native mode cause a PALcode trap to the MT_FPCR entry point, which consists of the following code:

```
HW_MFPR EXC_ADDR -> S1
IFETCHB
RET S1
```

The IFETCHB causes a next-to-retire event that is younger than the MT_FPCR, causing its value to be committed.

*** Peter: Why is the IFETCHB necessary here? The User-mode MT_FPCR commits at retire time and kills any inflight FP instructions. Isn't the trap alone enough to ensure correct state?

MT_FPCR instructions in PALmode do not trap, which means that the values in EXC_ADDR and S1 do not need to be safeguarded before executing an MT_FPCR. However, as a result, any PALcode that includes an MT_FPCR must execute an IFETCHB prior to exiting the flow. If there are any MT_FPCR, MF_FPCR, or floating-point instructions in a PALcode flow after an MT_FPCR, there must be an intervening IFETCHB to ensure that the MT_FPCR value has been committed before the reading/writing instructions issue.

Issues

* The 21264 had the behavior that an implicitly written register would read as zero if read while being written. Will the 21464 have the same behavior? Should we define a valid bit in each of the implicitly written registers to explicitly flag this case? All registers except VA have bit<63> available.

18
Initialization and Configuration

19
Performance Monitoring

The 21464 provides the most performance monitoring hardware of any Alpha implementation to date. The goal of the 21464 performance monitoring hardware is to provide information about the running CPU in order to:

1. Drive profiling-directed-feedback optimizations to improve application performance.
2. Enable the development of useful performance monitoring software tools.
3. Assist in post-silicon chip and system debug.
4. Provide information to enable more intellegent OS scheduling of processes onto TPUs.
5. Provide architectural feedback for future Alpha microprocessor and system implementations.

This document consists of two main sections. The first details the implementation of an instruction-based profiling algorithm called ProfileMe. The second section describes performance monitoring hardware for memory addresses that was developed for the 21364 and is being supported by the 21464.

19.1 Instruction Based Profiling

As microprocessors get more and more complicated, the behavior of instructions flowing through the machine is harder to determine with aggregate event counter style performance monitoring hardware, such as what was implemented in the 21164. Profile-based feedback has become very important in getting the best performance out of complex CPUs. The data obtained from hardware performance monitoring can be used to drive compiler optimizations that increase a CPU's architectural performance. Instruction-based profiling can get more accurate data about performance bottlenecks in speculative out-of-order microprocessors such as the 21264, the 21364, and the 21464. The basic idea is to enable software to sample fetched instructions randomly, collecting detailed information about each sampled instruction's execution in the machine. This includes information such as the amount of time the instruction spends in each phase of its lifetime in the CPU, as well as performance impacting events, such as cache misses and branch mispredictions. In the 21464, we implement a variation of ProfileMe called paired-sampling, where two in-flight instructions can be sampled simultaneously. This provides for better analysis, because events and data that are collected for the two instructions can be correlated. For more information on ProfileMe in general, or on paired sampling specifically, refer to the Micro-30 paper:

ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors.

In Proceedings of the 30th Annual International Symposium on Microarchitecture, pages 292-302. IEEE, December 1997. (Postscript)

Jeffrey Dean, James E. Hicks, Carl Waldspurger, William E. Weihl, and George Chrysos.

19.1.1 Profiling Methodology

Instruction-based profiling is performed by sampling the dynamic instruction stream running on the 21464. Sampled instructions are chosen at map time based upon a software programmable IPR (PR_INST_CTL<63:0>) and are monitored while in-flight in the CPU. Latencies and events are recorded for two separate instructions into a set of profile record IPRs. When both instructions have finished utilizing CPU resources, a general interrupt to PALcode is triggered. The general interrupt service routine will read the INTERRUPT-SUMMARY IPR to determine that the interrupt was caused by an instruction profile event. A privileged PAL routine can then read out the associated data for each profiled instruction by reading from the profile record IPRs. In continuous sampling, software would record the data from the current sample and reinitialize the PR_INST_CTL IPR to begin the process for selecting the next pair of sampled instructions.

The hardware provides two countdown values which determine the selection of the first and second profile instructions. In the normal profile mode, the countdown values indicate the number of valid (that is, having at least one valid instruction) map blocks from the specified profiled TPUs that are counted before the profiled instruction is assigned. In a special profile mode, called "PC trigger mode", the countdown values represent the number of valid map blocks from the specified profiled TPUs after instructions corresponding to a specified PC have been mapped. Due to hardware constraints, the countdown values actually represent the number of valid map blocks plus some small constant number of valid map blocks (currently 10).

19.1.2 Initiating an Instruction Profile Sample

To setup a profile sample, software calls a PAL routine that writes to the PR_MEM_EVENT_CTL and (possibly) the PR_TRIG_PC IPRs, followed an IFETCHB and PR_INST_CTL. The PAL routine should first check that an existing profile is not already underway. It can do this by reading the PR_I_INFO IPR and examining the "outstanding" bits, PR_I_INFO<63> and PR_I_INFO<31>, which indicate whether either the first or the second profiled instruction from a prior profiling attempt is still outstanding. The outstanding bits should both be clear when a profile is complete, and it is safe to read the profile record and start a new sample.

The profiling control IPRs: PR_INST_CTL, PR_MEM_EVENT_CTL, and PR_TRIG_PC are specified in Table 19–1.

Table 19–1 Control IPRs for Instruction-Based Profiling

IPR	Field	Bits	Description
PR_INST_CTL<63:0>			
	I_EVENT_CTL	63-60	Specifies the event to be counted by the IAGG_EVENTn IPRs between the retire/kill of the first profiled instruction and the retire/kill of the section profile instruction. The events are listed in Table 19–10.
	PROFILING_EN	59	Profiling Enabled. This will create a profiled sample according to the fields specified below.
	PC_TRIG_MODE	58	PC Trigger Mode Enabled. PR0_CTR will not start countdown until instructions corresponding to the PR_TRIG_PC for one of the PR_TPUS have been mapped.
	PR1_POS	57-55	Profile Second Map-Block Position Contains the exact position of the second profiled instruction in the map-block selected by the PR1_CTR. Two bits per instruction are delivered with each map-block to the Pbox. The bits correspond to whether each instruction is the first or second profiled instruction.
	PR1_CTR	54-31	Profile Interval for the second instruction Contains an unsigned number between 0 and 16M-1 that represents the number of metered map-blocks between the profile 0 and profile 1 instructions. A counter is initially written with the PR1_CTR value when the PR_INST_CTL IPR is written. When the first interval's counter reaches 0, the second interval's counter begins decrementing its value, again based on metering map-blocks from all appropriate TPUs specified in PR_TPUS. The profile 1 instruction of the pair is chosen from the first valid map-block sent to the Pbox after the second interval counter reaches 0. If the PC_TRIG_MODE is enabled, the countdown does not begin until one of the PR_TPUs has mapped instructions that correspond to the PR_TRIG_PC.
	PR0_POS	30-28	Profile First Map-Block Position Contains the exact position of the first profiled instruction in the map-block selected by the PR0_CTR.
	PR0_CTR	27-4	Profile Interval for the first instruction Contains an unsigned number between 0 and 16M-1 that represents the number of metered map-blocks. A counter is initialized to the PR0_CTR value when the PR_INST_CTL IPR is written. Each cycle that the Ibox's collapsing buffer sends a map-block with any valid instructions for a TPU specified by PR_TPUS to the Pbox, the counter is decremented. The first profiled instruction of the pair will be chosen from the first valid map-block that is being sent to the Pbox after the first interval counter reaches 0. If the PC_TRIG_MODE is enabled, the countdown does not begin until one of the PR_TPUs has mapped instructions that correspond to the PR_TRIG_PC.

Table 19–1 Control IPRs for Instruction-Based Profiling

IPR	Field	Bits	Description
	PR_TPUS	3-0	Specifies the TPUs to be selected for profiling. Only map-blocks for specified TPUs are metered and only instructions from those TPUs are profiled. If the value is 0000, no TPU will be profiled. Different instructions in a profiled pair can come from different specified TPUs.
PR_MEM_EVENT_CTL<63:0>			
	M_EVENT_CTL	63-60	Specifies the event to be counted by the MAGG_EVENTn IPRs between the retire/kill of the first profiled instruction and the retire/kill of the section profile instruction. The events are listed in Table 19–10.
	Reserved	59-0	Reserved for future use. Not currently used.
PR_TRIG_PC<63:0>			
	Reserved	63-52	Reserved for future use. Currently, writing to these bits does not cause any defined action in the chip.
	Match PC	51-5	These bits are used to compare against bits 51:5 of the PCs of valid mapped instructions for the PR_TPUS. A match will start the MAJ and MIN countdown counters if PC_TRIG mode is set in the PR_INST_CTL IPR. If the PC_TRIG_MODE is not set in the PC_INST_CTL IPR the counters will countdown without reguard to the Match PC. Mapped instructions on a badpath (instructions that are mapped but never commit to machine state (as in a branch mispredicted path) will also trigger the match.
	Reserved	4:0	Reserved for future use. Currently, writing to these bits does not cause any defined action in the chip. This implies that bits 4:0 are don't care in the PC comparison.

Considerations:

- The instruction position indicators (PR0_POS, PR1_POS) may be assigned to a map-block instruction position that does not contain a valid instruction. This will be reflected in the profile record, and will result in some invalid samples. The samples do provide some meaningful information, however, which is that the map-block was not full.
- If the value in the second interval counter (PR1_CTR) is initially 0, the two profiled instructions will be chosen from the same map-block. Note, however, that PR0_POS and PR1_POS can point to any position in the map-block, so the second ProfileMe instruction could be before the first in program order. Also, if PR0_POS and PR1_POS indicate the same position in the map-block, the hardware collects two records of the SAME instruction. It is up to software to avoid these effects if they are unwanted.

The I_EVENT_CTL and M_EVENT_CTL fields in the controlling IPRS listed above pertain to the use of the IAGG_EVENTn and the MAGG_EVENTn IPRs which are described in Section 19.1.3.3. Certain events per TPU in the window between when the first profile instruction is retired or killed and when the second profile instruction is retired or killed can be counted. During on paired profile sample, one Ibox event and

one Mbox event can be counted together, per TPU. The events that are counted are determined by the value in the I_EVENT_CTL and M_EVENT_CTL fields. Those event designations are listed in the following table:

Table 19–2 IAGG_EVENT and MAGG_EVENT IPRs

	Value	Event Counted
I_EVENT_CTL	0000	ICache Misses
	0001	Istream Scache Misses
	0010	Line Mispredicts
	0011	Istream misses serviced by remote Rambus
	0100	Way Mispredicts
	0101	Squashes
	0110	Squash Mispredicts
	0111	ICache read bank conflicts
	1000	ICache fill bank conflicts
	1001	Total postmap exceptions
	1010	Retired Instructions
	1011	ITB Misses
	1100	Branch Mispredicts
	1101	Jump/Return Mispredicts
	1110	Mapped Instructions
	1111	Load/Store Order Traps (tentative/temporary)
M_EVENT_CTL	0000	Total Retries
	0001	Total Traps
	0010	Synonym Traps
	0011	Store/Load Order Traps
	0100	Dcache Misses
	0101	Dstream Scache Misses
	0110	DTB Misses
	0111	Bad End Inum Retries
	1000	Wrong Size Retries
	1001	Reserved
	1010	Reserved
	1011	Reserved
	1100	Dcache Bank Conflict Retries

Table 19–2 IAGG_EVENT and MAGG_EVENT IPRs

Value	Event Counted
1101	Reserved
1110	Dstream misses serviced by local Rambus
1111	Reserved

19.1.3 Instruction Profile Record IPRs

Several IPRs collect information about the selected profiled instructions. The data consists of events, addresses, and various timing information about each profiled instruction's execution in the CPU.

19.1.3.1 Data/Event IPRs

The program counter, address space number (ASN) and TPU identifier of each profiled instruction are recorded in the PR0_PC<63:0> and PR1_PC<63:0> IPRs. A valid bit associated with each IPR is set to indicate whether the profiled instruction was part of a valid position in a valid map-block. Finally, an additional bit indicates whether the instruction is PALcode. When the instruction is PALcode, the associated PC will be a physical address, and the ASN is irrelevant.

Table 19–3 Fields in the PR0_PC<63:0> and PR1_PC<63:0>

Field	Extent	Description
Reserved	63	Not yet assigned.
Valid	62	The profiled instruction was from a valid instruction in a valid map block.
ASN	61-54	The address space number for the profiled instruction, if it was not in PALmode
TPU	53-52	The encoded TPU ID of the profiled instruction
PC	51-2	The PC (normally virtual) of the profiled instruction (bits 1 and 0 are always clear and not recorded).
SuperPalMode	1	Indicates the profiled instruction was taken in a special debug mode of the 21464
PalMode	0	Indicates the profiled instruction was a PAL instruction in privileged mode. If this bit is set, it also indicates that the PC field is a physical address

Events and data associated with the profiled instructions during the fetch, map, and queue stages are recorded in the PR_I_INFO<63:0> and PR_Q_INFO<63:0>. These IPRs record events for both profiled instructions.

Table 19–4 Fields in PR_I_INFO<63:0>

Name	Extent	Description
For the Profile1 Instruction		
PR1 OUTSTANDING	63	Indicates that the profile 1 instruction is outstanding. The bit is cleared, when the profile 1 instruction is either killed or retired.
PR1 SLOT	62	Indicates which icache access (up to two per cycle) the profile 1 instruction was fetched with.
PR1 LGHIST	61-56	Indicates the latest 6 bits of the lghist of the branch predictor corresponding to its 3-slot-old index that was used to access the branch predictor the cycle that a branch in the profiled instruction's block would have been predicted. The higher numbered bits are older and the lower numbered bits are younger.
RESERVED	55-53	Not yet assigned.
PR1 CAUSED KILL	52	The profiled instruction caused a kill. This means that if the instruction was a branch, the kill was a branch mispredict, if it was a jump, it was a jump mispredict, if it was a return, it was a return stack error, or if it was a load, a memory trap occurred.
PR1 SSID	51-47	The store set id of the profile 1 instruction. Only valid if PR1 SSID[51] is set and this instruction is a load or a store.
PR1 BPRED	46	Indicates whether the profiled instruction was a PC changing instruction, ie, either a predicted taken branch or an unconditional branch or jump or callpal. For a conditional branch, this bit indicates the branch prediction.
PR1 RETIRED	45	Indicates that the profile 1 instruction was retired. If PR1 RETIRED is set, the profiled instruction retired; if clear, the profiled instruction was killed.
PR1 LP BNK CONF	44	Indicates that the fetch block for the profile 1 instruction was delayed by 1 cycle due to a conflict with a training write into the line predictor.
PR1 EXCP REST	43	Indicates that the profile 1 instruction was in the first block fetched after an exception restart in the Ibox (branch mispredict, etc).
PR1 ICF BNK CNF	42	Indicates the profile 1 instruction was delayed by 1 cycle because an attempt to fetch it from the icache failed due to a conflict with an icache fill into the same Icache bank.
PR1 IC BNK CNF	41	Indicates the profile 1 instruction was delayed by 1 cycle because an attemp to fetch it from the icache failed due to a conflict with an icache read for another icache fetch in the same cycle.
PR1 SQSH MISP	40	Indicates the profile 1 instruction had a line mispredict due to an incorrectly predicted squash.
PR1 SQSH	39	Indicates that the profile 1instruction was delayed for 1 cycle due to a squash.

Table 19–4 Fields in PR_I_INFO<63:0>

Name	Extent	Description
PR1 WAY MISP	38	Indicates the profile 1 instruction was delayed for either 4 or 5 cycles due to an Icache way mispredict.
PR1 LOCAL RAM	37	Indicates that the profile 1 instruction was an icache miss, and an scache miss, and hit in the local RAMBUS memory (not a router request).
PR1 LINE MISP	36	Indicates the profile 1 instruction was delayed for 2 or 3 cycles due to a line predictor mispredict.
PR1 SC MISS	35	Indicates that the profile 1 instruction was an icache miss and an scache miss
PR1 ITB ENA	34	Indicates that the profile 1 instruction was an icache miss, and was delayed an extra 4 or 5 cycles due to the micro TB being out of date. This bit also indicates that the main 128 entry ITB was utilized to translate the PC from a VA to a PA.
PR1_ICMISS	33	Indicates that the profile 1 instruction was an icache miss.
PR1_SLOT1	32	Indicates that the original icache fetch for the profile 1 instruction was a slot 1 fetch. This can help to determine penalties for line mispredicts and way mispredicts. It this bit is clear, an indicated line mispredict caused a 2 cycle delay, and a way mispredict caused a 4 cycle delay. If the bit is set, an indicated line mispredict caused a 3 cycle delay, and a way mispredict caused a 5 cycle delay. Note these delay assumtions may be inaccu ate for a number of reasons, but should be right in the common cases.
For the Profile0 Instruction:		
PR0 OUTSTANDING	31	Indicates that the profile 0 instruction is outstanding. The bit is cleared, when the profile 0 instruction is either killed or retired.
PR0 SLOT	30	Indicates which icache access (up to two per cycle) the profile 0 instruction was fetched with.
PR0 LGHIST	29-24	Indicates the latest 6 bits of the lghist of the branch predictor corresponding to its 3-slot-old index that was used to access the branch predictor the cycle that a branch in the profiled instruction's block would have been predicted. The higher numbered bits are older and the lower numbered bits are younger.
RESERVED	23-21	Not yet assigned.
PR0 CAUSED KILL	20	The profiled instruction caused a kill. This means that if the instruction was a branch, the kill was a branch mispredict, if it was a jump, it was a jump mispredict, if it was a return, it was a return stack error, or if it was a load, a memory trap occurred.
PR0 SSID	19-15	The store set id of the profile 0 instruction. Only valid if PR0 SSID[19] is set and this instruction is a load or a store.
PR0 BPRED	14	Indicates whether the profiled instruction was a PC changing instruction, that is, either a predicted taken branch or an unconditional branch or jump or callpal. For a conditional branch, this bit indicates the branch prediction.

Table 19–4 Fields in PR_I_INFO<63:0>

Name	Extent	Description
PR0 RETIRED	13	Indicates that the profile 0 instruction was retired. If PR0 RETIRED is set, the profiled instruction retired; if clear, the profiled instruction was killed.
PR0 LP BNK CONF	12	Indicates that the fetch block for the profile 0 instruction was delayed by 1 cycle due to a conflict with a training write into the line predictor.
PR0 EXCP REST	11	Indicates that the profile 0 instruction was in the first block fetched after an exception restart in the Ibox (branch mispredict, etc).
PR0 ICF BNK CNF	10	Indicates the profile 0 instruction was delayed by 1 cycle because an attempt to fetch it from the Icache failed due to a conflict with an Icache fill into the same Icache bank.
PR0 IC BNK CNF	9	Indicates the profile 0 instruction was delayed by 1 cycle because an attempt to fetch it from the icache failed due to a conflict with an icache read for another icache fetch in the same cycle.
PR0 SQSH MISP	8	Indicates the profile 0 instruction had a line mispredict due to an incorrectly predicted squash.
PR0 SQSH	7	Indicates that the profile 0 instruction was delayed for 1 cycle due to a squash.
PR0 WAY MISP	6	Indicates the profile 0 instruction was delayed for either 4 or 5 cycles due to an Icache way mispredict.
PR0 LOCAL RAM	5	Indicates that the profile 0 instruction was an icache miss, and an scache miss, and hit in the local RAMBUS memory (not a router request).
PR0 LINE MISP	4	Indicates the profile 0 instruction was delayed for 2 or 3 cycles due to a line predictor mispredict.
PR0 SC MISS	3	Indicates that the profile 0 instruction was an icache miss and an scache miss
PR0 ITB ENA	2	Indicates that the profile 0 instruction was an icache miss, and was delayed an extra 4 or 5 cycles due to the micro TB being out of date. This bit also indicates that the main 128 entry ITB was utilized to translate the PC from a VA to a PA.
PR0_ICMISS	1	Indicates that the profile 0 instruction was an icache miss.
PR0_SLOT1	0	Indicates that the original icache fetch for the profile 0 instruction was a slot 1 fetch. This can help to determine penalties for line mispredicts and way mispredicts. It this bit is clear, an indicated line mispredict caused a 2 cycle delay, and a way mispredict caused a 4 cycle delay. If the bit is set, an indicated line mispredict caused a 3 cycle delay, and a way mispredict caused a 5 cycle delay. Note these delay assumtions may be inaccu ate for a number of reasons, but should be right in the common cases.

Table 19–5 Fields in PR_Q_INFO<63:0>

Field	Bits	Description
For the Profile1 Instruction:		
Reserved	63–45	Not yet assigned
PR1 INSTRS MAPPED	44–41	Indicates the number of valid instructions in the profile 1 instruction's map block.
PR1 GRANT CNT	40–36	Indicates the number of times the profile 1 instruction was granted for execution. If the number is greater than 1, it indicates that the profile 1 instruction was poisoned and reexcecuted (this value - 1) times.
PR1 BRMISS OLD	35	If the profile 1 instruction was a branch that mispredicted, this bit indicates that it was the oldest mispredicting conditional branch in the cycle it mispredicted, thus qualifying it to take the "fastpath" to restart the PC on the goodpath. If it is not set for a mispredicting branch, it indicates that this branch had a several cycles of additional branch mispredict penalty (about 4 or 5).
PR1 PICKER	34–32	Indicates the number of the Qbox Picker which selected the profile 1 instruction for execution. If the picker number is not the same as the map block position of this instruction (PR1_POS for the profile 1 instruction), it indicates that this instruction followed a parent instruction to another picker.
For the Profile0 Instruction:		
Reserved	31–13	Not yet assigned
PR0 INSTRS MAPPED	12–9	Indicates the number of valid instructions in the profile 0 instruction's map block.
PR0 GRANT CNT	8–4	Indicates the number of times the profile 0 instruction was granted for execution. If the number is greater than 1, it indicates that the profile 0 instruction was poisoned and reexcecuted (this value - 1) times.
PR0 BRMISS OLD	3	If the profile 0 instruction was a branch that mispredicted, this bit indicates that it was the oldest mispredicting conditional branch in the cycle it mispredicted, thus qualifying it to take the "fastpath" to restart the PC on the goodpath. If it is not set for a mispredicting branch, it indicates that this branch had a several cycles of additional branch mispredict penalty (about 4 or 5).
PR0 PICKER	2–0	Indicates the number of the Qbox Picker which selected the profile 0 instruction for execution. If the picker number is not the same as the map block position of this instruction (PR1_POS for the profile 0 instruction), it indicates that this instruction followed a parent instruction to another picker.

Data associated with loads and stores is recorded in the PR0_MEM_INFO<63:0> and PR1_MEM_INFO<63:0> IPRs.

Table 19–6 Fields in PR0_MEM_INFO<63:0> and PR1_MEM_INFO<63:0>

Name	Extent	Description
Reserved	<63:62>	Not yet assigned..
CONTENTION	<61>	Mbox trap or replay was due to contention with the other profiled instruction.
TRP_INV	<60>	Mbox trap - invalidate speculative load or store
TRP_SYN	<59>	Mbox trap - virtual synonym dependence ignored
TRP_SLOO	<58>	Mbox trap - dependent store-load executed out of order
TRP_DTBM	<57>	Mbox trap - DTB miss.
RET_SCM	<56>	Mbox retry - scache miss
RET_BAD_EINUM	<55>	Mbox retry - bad end inum
RET_WRSZ	<54>	Mbox retry - wrong size ld/st
RET_BCNF	<53>	Mbox retry - dcache bank conflict.
RET_DCM	<52>	Mbox retry - dcache miss.
VA	<51:0>	The virtual address of the profiled instruction if it was a load or store.

The Scache, memory controller and router can be invoked for loads and stores that miss in the first level Dcache. Two IPRs, PR0_DMISS_INFO<63:0> and PR1_DMISS_INFO<63:0> collect latency and event information for the profiled instructions that generate activity in the Cbox.

Table 19–7 Fields in PR0_DMISS_INFO<63:0> and PR1_DMISS_INFO<63:0>

Name	Extent	Description
Reserved	<63:61>	Not yet assigned.
ROUTER_DEST	<60:52>	Processor id of a memory request for the profiled instruction that is serviced remotely (ie, another processor is the home node)
LAT_SNAP_FWD	<51:43>	For a local RAM access, the number of cycles from the snapshot point until the data is forwarded.
LAT_DENQ_SNAP	<42:35>	For a local RAM access, the number of cycles from DIFT enqueue until the snapshot point.
RAM_RAS	<34>	Indicates the bank for the profiled instruction's request had to perform a row access strobe, ie, this request was not a page hit in the local RAM. For local RAM requests only.
RAM_PRCHG	<33>	Indicates the bank for the profiled instruciton's request had to precharge. For local RAM requests only.
DIR_CACHE_HIT	<32>	Indicates the profiled instruction's request hit in the directory cache.
LCL_RMBS	<31>	If the profiled instruction results in an scache miss, this bit indicates whether the data was resident in the local memory.

Table 19–7 Fields in PR0_DMISS_INFO<63:0> and PR1_DMISS_INFO<63:0>

Name	Extent	Description
COHER_CNT	<30:26>	Records the number of invalidates the profiled instructions memory request must await before being allowed to obtain ownership of the requested block.
MAF_ALC_DLC	<25:16>	Records the number of cycles the profiled instruction's MAF entry was allocated to the time when it was deallocated.
PMAF_LAT_RPL	<15:12>	Records the number of cycles the profiled instruction's preMAF request waited for the Scache due to replays.
PMAF_LAT_PRB	<11:8>	Records the number of cycles the profiled instruction's preMAF request waited for the Scache due to probes.
PMAF_LAT_ICM	<7:4>	Records the number of cycles the profiled instruction's preMAF request waited for the Scache due to Icache miss requests.
PMAF_LAT_FILL	<3:0>	Records the number of cycles the profiled instruction's preMAF request waited for the Scache due to other fills.

19.1.3.2 Timeline/Latency IPRs

The 21464 keeps an internal running cycle counter whose value is recorded when certain events happen to a profiled instruction. A sequence of recorded counter values creates a timeline of a profiled instruction's execution in the CPU. For example, the counter is recorded into a timeline register field when the instruction fetch unit first tries to fetch the profiled instruction. The counter is recorded into another timeline register field when the instruction is retired. By subtracting the two recorded counter values, software can determine how long a profiled instruction was in-flight in the CPU. A complete timeline for an instruction gives insight about the performance bottlenecks in the CPU. Figure 19–1 illustrates the timeline that is captured for each profiled instruction:

Figure 19–1 Captured Timeline for Each Profiled Instruction

	Fetch (Last Valid Map)	Map	IQ Alloc	Data Ready	Bid Enable	Grant	IQ Dealloc	Chunk IQ Dealloc	Retire Able	Killed or Retired
Counter Bits Recorded	32	16	16	16	16	16	16	16	16	48

In addition to the basic timeline described above, the timeline IPRS also record the time that the last valid instruction before the profile instruction, and in the same TPU, became retireable. This is useful to determine whether the profile instruction's execution had delayed the executing program, and if so, by how many cycles. If the prior instruction's retireable time and the profiled instruction's retireable time are the same, the profiled instruction did not directly contribute to the execution time of the running program. If, on the other hand, the prior instruction's retireable time is earlier than the profiled instruction's, speeding up the processing of the profiled instruction could increase the performance of the running program.

The sizes of the "Fetch" and "Retire" timeline fields are intentionally large, to ensure that in the event of an instruction that was very slow to execute, the total time from Fetch to Retire can still be computed. The 32 bit register field sizes of fetch and the 48-bit register field of retire also serve to allow software to determine the time between the two profiled instructions. The upper 16 bits of the retire field allow software to determine the time between two pairs of samples. So, the latency between when the first profiled instruction retired and the second profiled instruction retired can be calculated simply by subtracting the two retired timeline snapshots. The same running cycle counter is recorded for both profiled instructions, which provides this feature.

Due to poisoning (see Mbox/Qbox documentation), instructions can actually go through some of the timeline points more than once. If this happens the register field(s) corresponding to the recurring events will simply be set more than once. This will actually yield the correct data, as earlier event occurances were due to a misspeculation. So, for example, an instruction may appear data ready, but is not in reality.

There are four timeline IPRs per profiled instruction. The fields of the timeline IPRs and their meanings are specified in Table 19–8. Note that the following timestamps are UNPREDICTABLE for instructions that are invalid (not mapped), decoded as a 21464 NOP, or killed:

IQ_ALLOC
DATA_READY
BID_ENABLE
GRANT
IQ_DEALLOC

Also, the RETIREABLE timestamp is UNPREDICATABLE for instructions that are killed.

Table 19–8 PRn_TIMELINE IPRs[1]

IPR	Field	Bits	Description
PRn_TIMELINE0			
	FETCH	63–32	Fetch really implies the earliest time at which the profiled instruction could have been fetched, or the time of the last valid map from one of the PR_TPUs. If there is more than 1 cycle between the last valid map, and the map time of the profiled instruction, a fetch delay of some sort was encountered. The fetch delay could have been due to an icache miss, line mispredict, way mispredict, etc, or just because the map thread chooser chose against the PR_TPUs. The use of the PR_I_INFO IPR data can help to determine the cause of the fetch delay.
	RETIRE/KILL (bits 31–0)	31–0	The time when the profiled instruction's map block inum was broadcast on the RK bus as a retire block, OR, the time that a kill that killed the profiled instruction was broadcast on the rk bus. PR_I_INFO will indicate whether the profiled instruction retired, was killed, or caused a kill itself.
PRn_TIMELINE1			
	MAP	63–48	The time when the profiled instruction's map block was driven from the Ibox to the Pbox.

Table 19–8 PRn_TIMELINE IPRs[1]

IPR	Field	Bits	Description
	IQ_ALLOC	47–32	The time when the profiled instruction's map block was allocated space in the instruction queue. This is usually a fixed delay from the "map" time, unless the instruction queue is backed up and not able to allocate more space. If this happens the profiled instruction's map block can be delayed in the post-map skid buffer.
	DATA_READY	31–16	The time when the profiled instruction is data ready, that is all of it's source operands (including store sets for loads) were available.
	BID_ENABLE	15–0	This time is normally the cycle following data_ready for most instructions. The notable exceptions are loads and stores, which could be data ready, but not issue because there are not enough entries in the load or store queues. Also, if bid_enable occured the same cycle as data_ready, it implies that the profiled instruction followed it's parent to another picker.
PRn_TIMELINE2			
	GRANT	63–48	This is more commonly known as "issue time", or the time when the profiled instruction is sent to it's functional unit for execution. If there is more cycles beween grant and bid_enable than the normal fixed delay, it implies that the profiled instruction suffered from functional unit queueing delay.
	IQ_DEALLOC	47–32	This is the time that an instruction is past it's poison point and will no longer hold up it's queue chunk's iq deallocation time. This also corresponds to the completion unit's notion of "complete".
	PRED_RETIREABLE	31–16	This is the time that the last valid predecessor instruction, before the profiled instruction, became retireable. The predecessor instruction must be in the same thread (and running on the same TPU) as the profiled instruction.
	RETIREABLE	15–0	This is the time that the profiled instruction itself became retireable, that is when it is complete, and all instructions older than it in the same TPU are also complete. It should be greater than or equal to PRED_RETIREABLE above. If this is not the same as PRED_RETIREABLE, it indicates that the profiled instruction contributed to the overall program execution time by the number of cycles difference between the two. Instructions which have much greater retireable times than PRED_RETIREABLE times point to areas in the program that contribute to significant performance loss.
PRn_TIMELINE3			

Table 19–8 PRn_TIMELINE IPRs[1]

IPR	Field	Bits	Description
	IQ_CHK_DEALLOC	63–48	This is the time that the iq chunk that the profiled instruction was a part of is deallocated. If this is the same time as the iq_dealloc time in PRn_TIMELINE2 above, it indicates that the profiled instruction was one of, or the only, instruction gating the deallocation of the queue chunk. If the iq_cnk_dealloc time is greater than the iq_dealloc time, it indicates that a different instruction in the iq chunk was gating the deallocation of the queue chunk. By subtracting iq_alloc time (PRn_TIMELINE1) from iq_cnk_dealloc time, software can obtain the total q chunk lifetime for the profiled instruction's queue chunk. Since the queue chunks are a limited resource, a high average queue chunklifetime may indicate a performance bottleneck in a running program. The compiler or a run time optimizer, may be able to associate long latency instructions together in the same queue chunks, so that other queue chunks with all shorter latency instructions are deallocated sooner, alleviating the queue chunk resource constraint.
	RETIRE (bits 47–32)	47–32	Upper 16 bits of retire timestamp
	Reserved	31–0	Not yet assigned.

[1] Where n = 0 means first profiled instruction and n = 1 means second profiled instruction.

There is an additional latency IPR associated with store processing. The latency counters are only for the first profiled instruction. The data is interesting if both profiled instructions are stores, and the latter store is not able to issue because the prior store is clogging store processing. The IPR that holds the latencies is called PR_ST_LATENCY<63:0>.

Table 19–9 Fields in PR_ST_LATENCY<63:0>

Name	Extent	Boxes	Description
Reserved	<63:24>		Not yet assigned.
ACK_2_MBFREE	<23:16>	M	Number of cycles between the first profiled instruction's merge buffer entry is acknowleged and the time that that merge buffer entry is freed.
MB_2_ACK	<15:8>	M	Number of cycles between the first profiled instruction is eligible to merge in the merge buffer and the time that its entry into the merge buffer is acknowleged.
STQ_2_MB	<7:0>	M	Number of cycles between the first profiled instruction enters the store queue and it becomes eligible to merge in the merge buffer.

19.1.3.3 Aggregate Event/Data IPRs

The IPRs listed before here in this section obtain information specifically about the profiled instructions. The 21464 also collects aggregate events in region of execution between the retrires or kills of the two profile instructions. This allows for the collecting of aggregate event or data information in a certain region, per TPU. So, over an interval delimited by the retire times of two dynamic instructions, information such as:

Retired instructions

ITB Misses

Store Load Order Traps

Scache Misses

And so forth (see the I_EVENT_CTL and M_EVENT_CTL IPR definitions above for a full list).

can be accrued per TPU. The per TPU information can be summed to give total CPU stats. The rate of these events can also be determined by subtracting the retire/kill time of the first profiled instruction from the second. This gives the total number of cycles that the aggregate event counters were monitoring the selected events. Dividing the events by cycles, yields the event rate (eg. Retired instructions per cycle). Repeated profile samples will give multiple data points over the span of a programs execution. In general, fairly infrequent samples can create very accurate data, but, if needed, the samples can be very close together, so as not to miss any statistically significant information using the following algorithm:

> In PR_INST_CTL the profile PR0_CTR to something small (say 0 or slightly larger to ignore the profiling PAL routine and other software overhead), and the PR1_CTR to something relatively large (say 1-16 Million Map Blocks). When the interrupt is triggered, collect the information, and reset the PR_INST_CTL register again to initiate the next sample.

The profiling hardware can only collect two aggregate events per sample, 1 IBOX(Instruction Unit) related event, and 1 MBOX (Memory Unit) event. A software profiler can alternate between the events to get them all. This should work well for programs whose behavior is repetative. Varying the PR1_CTR, or the order that the events are collected, will avoid missing phasic behavior (eg, If retired instructions and icache misses are alternately collected for 16M map blocks each, and the program just so happens to have different phases at the same frequency, it would be a mistake to assume that the icache miss rate and the IPC rate are constantly at their measured values. Alternating the order in which they are collected, or varying the PR1_CTR time will help avoid this). If a program does not have repeated behavior, the program can be sampled over several runs to obtain all the data.

This can be quite powerful in finding bottlenecks in a running program. A chart of IPC over time will reveal sections of the program that are performing the least. The other aggregate event information can hint at the cause of this diminished performance. Also, the PC's of profiled instructions collected in the regions of low performance, can later become trigger PCs in the PR_TRIG_PC IPR, in order to collect more instruction samples in the regions of diminished performance.

The aggregate event counter IPRs are specified in Table 19–10

Table 19–10 Aggregate Event Counter IPRs

IPR	Bits	Description
IAGG_EVENTn	31–0	The aggregate event count chosen by the value written by software into I_EVENT_CTL. The events are listed in Table 19–1. The n pertains to the TPU for which the events are collected. There are a total of 4 32 bit IPRs here, one per TPU. The events are counted between the retire/kill of the first and second profile instructions.
MAGG_EVENTn	31:0	The aggregate event count chosen by the value written by software into M_EVENT_CTL. The events are listed in Table 19–1. The n pertains to the TPU for which the events are collected. There are a total of 4 32 bit IPRs here, one per TPU. The events are counted between the retire/kill of the first and second profile instructions.

19.2 Memory Reference Performance Monitoring

The memory reference performance monitoring hardware is identical to that of the 21364. While the 21464 designers intend to support the same functionality, this specification may change to reflect architectural differences in the memory subsystem of the two processors.

Instead of IPRs, this performance monitoring hardware is controlled and collected via IO mapped CSRs. There are separate sections for the Cbox, Rbox, and Zbox.

19.2.1 Cbox Performance CSRs

19.2.1.1 Cbox Performance Control — CBOX_PRF_CTL<31:0>

Table 19–11 Fields in CBOX_PRF_CTL<31:0>

Name	Extent	Access	Description
ISTM_SAMP_ENA	<31>	RW,0	Enable istm sampling (on non-abtd bcache lookup)
PRF_SAMP_ENA	<30>	RW,0	Enable performance sampling
PAGE_MIGR_FAST	<29>	RW,0	Selects between 0 (fastest possible sample (=1)) or 16 (=0) events between migration samples
PAGE_MIGR_ENA	<28>	RW,0	Enable page migration sampling
WATCH_SEL	<27>	RW,0	Event for watch register to trigger on 0- BC lookup (non-aborted) 1- SYS sent
WATCH_ENA	<26>	RW,0	Enable watch register

19.2.1.2 Cbox Performance Address — CBOX_PRF_ADR<63:0>

Table 19–12 Fields in CBOX_PRF_ADR<63:0>

Name	Extent	Access	Description
PA<42:6>	<63:27>	RW	RW physical address
REQPID<10:0>	<25:15>	RW	Requestor PID
OPCODE<7:0>	<9:2>	RW	Network opcode -or- CMAF rdtype w/opcode<7:4> == 0

The performance sample.

A sampled watch address due to WATCH_EN locks the register until CBOX_PRF_ADR is written.

19.2.1.3 Cbox Performance Status — CBOX_PRF_STS<25:0>

Table 19–13 Fields in CBOX_PRF_STS<25:0>

Name	Extent	Access	Description
SYS_BYP_USED	<25>	RO,0	System port bypass was used
SYS_BYP	<24>	RO,0	Address granted bypass to system port
TAG_BYP	<23>	RO,0	Address bypassed from Mbox to BTAG
COUPLED	<22>	RO,0	Lookup was CMAF coupled lookup
CMAF_HIT	<21>	RO,0	c_cmf –> prq_cmaf_addr_hit_12a
SVAF_HIT	<20>	RO,0	c_cmf –> prq_svaf_addr_hit_12a
BC_DTY	<19>	RO,0	b –> c_blk_dirty_12a
BC_SHR	<18>	RO,0	b –> c_blk_shared_12a
DMR_BCV	<17>	RO,0	b –> c_dm_reqd_bcv_12a
DMR_BCVDC	<16>	RO,0	b –> c_dm_reqd_bcv_in_dc_12a
DMR_DCS	<15>	RO,0	b –> c_dm_reqd_dc_syn_12a
BC_VLC	<14>	RO,0	b –> c_local_bcv_12a
BC_VSH	<13>	RO,0	b –> c_bcv_shared_12a
DC_BCV	<12>	RO,0	b –> c_bcv_in_dcache_12a
BC_VIC	<11>	RO,0	b –> c_bc_victim_12a
DC_SYN	<10>	RO,0	b –> c_dc_synonym_12a
BC_HIT	<9>	RO,0	b –> c_bc_hit_12a
DC_VIC	<8>	RO,0	b –> c_dc_victim_12a
OPCODE<7:0>	<7:0>	RO,0	Network opcode -or- CMAF rdtype w/opcode<7:4> == 0

The status associated with CBOX_PRF_ADR.These bits are reset to zero on either cold or fast reset.

19.2.1.4 Cbox Performance Match — CBOX_PRF_MAT<25:0>

Fields are the same as CBOX_PRF_STS, except they are RW. See Section 19.2.1.3)

This register provides, in combination with CBOX_PRF_MATV, a way to filter performance events. A set bit in this register means that the CBOX_PRF_CNT event counter will only increment forperformance events which would set the corresponding bit in CBOX_PRF_STS to the value given by the corresponding bit in CBOX_PRF_MATV.

For example, if CBOX_PRF_MAT<25:24> == 3 and !CBOX_PRF_MATV<25:24> == 1, only performance events which were granted a system port bypass and did not use it would be countedby the event counter.

19.2.1.5 Cbox Performance Match Value — CBOX_PRF_MATV<25:0>

The fields are the same as CBOX_PRF_STS, except they are RW. See Section 19.2.1.3. Also see the description under CBOX_PRF_MAT, Section 19.2.1.4.

19.2.1.6 Cbox Performance Counter — CBOX_PRF_CNT<31:0>

Table 19–14 Fields in CBOX_PRF_CNT<31:0>

Name	Extent	Access	Description
EVENT_CNT<31:0>	<31:0>	RW	RW performance counter Software can write this counter to any value to provide any resolution desired. Interrupts are triggered upon carry out of the high-bit of thecounter, so writing the initial counter value with a large number willcause an earlier interrupt.

See description of CBOX_PRF_MAT (3.1.4) for explanation of how the decision to increment this counter is made.

19.2.2 Zbox Performance CSRs

19.2.2.1 Zbox Performance Counter 0 — ZBOX*n*_ZPM_CTR0<31:0>

Table 19–15 Fields in ZBOX*n*_ZPM_CTR0<31:0>

Name	Extent	Access	Description
ZBOX_PERF_CTR0_UND	<31>	RW	Indicates counter underflow.
ZBOX_PERF_CTR0	<30: 0>	RW	Zbox Performance counter 0.

Decrements when the condition specified by ZPM_CTL0 have been met. A performance counter interrupt will be signalled when the counter underflows. A 31-bit event counter and an underflow bit. ZPM_CTR0 can be programmed to count one of 32items related to the Zbox middle.The counter can be preloaded with an initial count via software.When the selected event occurs, the corresponding counter is decremented. When either counter counts below zero, the Zbox will generate a performance_monitorinterrupt.Note that only the first underflow causes a perf-monitor interrupt,so we can disable the interrupt by writing a 1 to the underflow bit.The interrupt occurs on the 0 –> –1 transition, so #events1 must be loaded into the counters.

19.2.2.2 Zbox Performance Counter 1 — ZBOX*n*_ZPM_CTR1<31:0>

Table 19–16 Fields in ZBOX*n*_ZPM_CTL1<31:0>

Name	Extent	Access	Description
ZBOX_PERF_CTR1_UND	<31>	RW	Indicates counter underflow.
ZBOX_PERF_CTR1	<30: 0>	RW	Zbox Performance counter 1.

Decrements when the condition specified by ZPM_CTL1 have been met. A performance counter interrupt will be signalled when the counter underflows.A 31-bit event counter and an underflow bit. ZPM_CTR1 can be programmed to count one of 16 items related to the Zbox front-end (DIFT).The counter can be preloaded with an initial count via software.When the selected event occurs, the corresponding counter is decremented.When either counter counts below zero, the Zbox will generate a performance_monitorinterrupt.Note that only the first underflow causes a perf-monitor interrupt,so we can disable the interrupt by writing a 1 to the underflow bit.The interrupt occurs on the 0 –> –1 transition, so #event-1 must be loaded into thecounters.

19.2.2.3 Zbox Performance Control — ZBOX*n*_ZPM_CTL<31:0>

Table 19–17 shows the fields in ZBOXn_ZPM_CTL<31:0> IPR

Table 19–17 Fields in ZBOX*n*_ZPM_CTL<31:0>

Name	Extent	Access	Description
unused<31:12>	<31:12>	RO	MBZ

Table 19–17 Fields in ZBOX*n*_ZPM_CTL<31:0> (Continued)

Name	Extent	Access	Description
unused<11:9>	<11: 9>	RW	MBZ
ZPM_CTL1<3:0>	<8:5>	RW	Control for Zbox performance counter 1:

ctl1	Item to Count (ZPM_CTR1)
0000	Incoming transaction (any)
0001	Incoming ReadSharedReq
0010	Incoming ReadModREq
0011	Incoming ReadReq
0100	Incoming FetchReq
0101	Incoming SharedToDirtyReq
0110	Incoming SharedToDirtySTCReq
0111	Incoming InvalToDirtyReq
1000	Incoming Victim
1001	Incoming VictimClean
1001	Outgoing Forward (any)
1010	Outgoing Forward=InvalSingle
1011	Outgoing Forward=InvalMask
1100	Outgoing Forward=Read(anytype)Forward
1101	Outgoing Forward=FetchForward (tRR, tPP)
1110	Outgoing Forward=ItoDForward
1111	Forward Miss received

Table 19–17 Fields in ZBOX*n*_ZPM_CTL<31:0> (Continued)

Name	Extent	Access	Description
ZPM_CTL0<4:0>	<4:0>	RW	Control for Zbox performance counter 0:

ctl0	Item to Count (ZPM_CTR0)
00000	nq_any -- regardless of reject status
00001	nq_prq -- regardless of reject status
00010	nq_rsq -- regardless of reject status
00011	nq_csq -- regardless of reject status
00100	nq_any -- qualified by & !reject
00101	nq_prq -- qualified by & !reject
00110	nq_rsq -- qualified by & !reject
00111	q_csq -- qualified by & !reject
01000	q_any -- qualified by & reject
01001	nq_prq -- qualified by & reject
01010	nq_rsq -- qualified by & reject
01011	nq_csq -- qualified by & reject
01100	nq_rej -- No Fill Buffers available
01101	nq_rej -- Shadow Reject in SHDPND , PDN Interval
01110	nq_rej -- Page-Conflict Reject (tRAS) in HLDPND, PND Interval (R/w) (tRDP) in SHDPND interval (R) (tRP,tRCD) in BLKPND, PND or NQPRPND Interval
01111	nq_rej -- WRB Reject (tRTR, tRTP)
10000	nq_rej -- Queue Full Reject
10001	nq-rej -- NQ' Waterfall prior over DFT-NQ 10010 cmd = dir_only_read
10011	cmd = dir+data_read
10100	cmd = dir_only_write
10101	cmd = dir+data_write
10110	PRER precharge
10111	PREX precharge
11000	PREC precharge
11001	COL=RD
11010	COL=WR
11011	COL=NOCOP
11100	COL=Any
11101	Starvation detections
11110	Force wr. ret
11111	Deferred write retire

19.2.3 Rbox Peformance CSRs

This section describes the Rbox performance IPRs.

19.2.3.1 Rbox Port Performance Counter — RBOX_*n*_PERF<27:0>

Table 19–18 Fields in RBOX_*n*_PERF<27:0>

Name	Extent	Access	Description
PCV<23:0>	<27:4>	RW	Counter value There is a hidden (not software visible) 8-bit register. The hidden register is cleared on every write (by software) to this register. The software-visible counter is only incremented when the hidden register overflows. The counter stops incrementing once it overflows (carry out of bit <27>). An interrupt is also asserted at that point (but may be blocked by the interrupt mask).
PCC	<1:0>	RW	Counter Control 00 - port utilization (increment for every outgoing used tick) 01 - undefined 10 - # of message bypasses 11 - # of messages

19.2.3.2 Rbox IO Port Performance Counter — RBOX_IO_PERF<27:0>

Table 19–19 Fields in RBOX_IO_PERF<27:0>

Name	Extent	Access	Description
PCV<23:0>	<27:4>	RW	Counter value There is a hidden (not software visible) 8-bit register. The hidden register is cleared on every write (by software) to this register. The software-visible counter is only incremented when the hidden register overflows. The counter stops incrementing once it overflows (carry out of bit <27>). An interrupt is also asserted at that point (but may be blocked by the interrupt mask).
PCC<1:0>	<1:0>	RW	Counter Control 00 - port utilization (increment for every outgoing used tick) 01 - # cycles the router is in drain mode 10 - # cycles the router is in starve mode 11 - # of messages

19.3 Addendum: Implemention Notes

19.3.1 From Data/Event IPRs

Implementation Note:

The information for generating the profileMe PCs is already kept in the PC Table. When the collapsing buffer drives map blocks to the Pbox, information from the pre-map PC table is read out and merged with information coming from the collapsing buffer to be stored into the Post-Map PC Table. The PC<51:2> of a profiled instruction can be determined from the following pieces of information:

- PC A<51:5> - the first fetch-block's PC in the map block
- PC B<51:5> - the second fetch-block's PC in the map block
- Total Map Block Length<3:0> - in instructions

- Length of instructions from slot A<3:0> - in instructions
- Starting PC offset position for slot A <3:0>
- Starting PC offset position for slot B <3:0>
- Position of the profiled instruction in the map block, (PR0_POS, and PR1_POS in PR_INST_CTL)
- Two bits from the selection engine that indicate either or both of the profiled instructions are being chosen this cycle.

19.3.2 Following Table 17-4

Implementation Note:

- Event-flags will be kept, per TPU, along with the current PC latches in the PC data-path. This state will indicate whether certain types of events happened while attempting to fetch from that PC. We can take advantage of the fact that all restarts: Icache Miss, Line Mispredict, Way Mispredict, micro TB (uTB) out of date, and Exception/Disruption restarts, will all restart in slot 0, resetting the current PC latch in I2B. When that PC latch is set, the event-flags for those restarts can also be set. Whenever a PC is sent onto I3, the event information is sent along with it, indicating the events that occurred regarding that PC fetch. If a pipe restart occurs while attempting to fetch a PC that already has event-flags set, the event-flags will follow the PC and be copied back into the current event-flags state. In this way, we can record multiple events for one attempted PC fetch. So, if first we line mispredict, then Icache miss, we can see both events.
- These event-flag bits will be recorded for each of the profileMe instructions when they are chosen. If the PC changes due to a PC-redirecting exception, the event-flags are cleared, because the old event-flags no longer pertain to THIS fetch. Also, an event flag indicating that this was a PC redirect is set. Whenever the PC is incremented or changed as part of normal program flow, the event-flags are cleared. The event-flags must be passed through the pre-map PC Table to stay with the appropriate fetch block. Unfortunately, the pre-map PC Table will have space for event-flags for all fetch slots, even though slot 1 fetches do not need any. This is because any or all slots in the collapsing buffer could be slot-0 fetches. When a map-block is sent with one or more profileMe instructions in it, we store the event-flags in a register (there is no need to store these in the post-map PC Table), to be read later by IPR reads. Right now, we should plan for about 8 bits for event flags.

20
Hardware Debug Features

Debugging real 21464 hardware in a timely fashion is a key element to achieving our time-to-market goals. As the complexity and speed of Alpha chips increases, the ability to observe, understand and potentially effect the operation of faulty hardware through just software or the pin interface gets more difficult and time consuming. This document will outline the capabilities we intend to include in the 21464 to aid in the hardware debug and FRS effort.

In the past, capabilities such as feature disable or bypass bits, performance monitors, and status ports have been embedded into silicon for the purpose of debug. Many of these features have proved invaluable in the quest to understand unexpected behavior. Previous experience has shown how difficult and time-consuming the process can be when there is minimal controllability and almost no visibility features. The 21464 will be significantly more complex and must have better debug hooks in the hardware.

Although this document will focus on features to help find logic bugs, other sources of problems like manufacturing defects, implementation and layout errors or even software errors can create problems which appear to be logic errors and often can be isolated through the same techniques. This document will not cover manufacturing test specific goals or features but it should be noted that observability features added for system debug can ease the manufacturing test pattern generation process.

The process of specifying the debug features in the 21464 has just begun. At this point, the goal of this document is to provide some focus and structure and to specify the common high level features that are currently being proposed. Eventually, as the detailed lists of signals and controls being integrated into each box becomes better defined, this document will become a reference. As this is still in the proposal stage, any suggestions for enhancements or other concerns are definitely welcome.

20.1 Debug Process

Typically, when something goes wrong, the first goal is to just reproduce the failure, then to prune the case down to reduce the number of cycles and eliminate interactions with other possible sources of error. The difficult task here is identifying the conditions that contribute to the failure and recreating just those conditions in an ever more simplified way.

Once the failure is easily reproducible, the task shifts to isolation of the exact cause. In a simulator we would typically augment tracefiles with signals until the problem is traced backwards from symptom to cause. In real cases where the stimulus fails to reproduce the failure in the model, hardware features that allow internal state to be observed are precious.

Quickly identifying the cause of a failure is very important to achieving time-to-market goals, but the ability to find a workaround may be equally important. Fab times for the 21464 will likely result in weeks or months to turn-around changes to silicon. Once prototypes or revenue hardware is shipped, the cost of hardware upgrades also factors into the cost and impact of bugs. Bugs will exist, our success depends on the ability to quickly resolve the issues and ship revenue hardware.

The Debug features we incorporate into the 21464 need to address the demands of all the phases of debug. When reviewing this document or defining visibility hooks, keep the debug flow in mind and try to ensure we create a complete solution.

20.2 Feature Overview

The 21464 has committed to implementing some amount of debug logic specifically targeted at observing and controlling the processors flow. The trick is to carefully balance the cost with the benefit. The overall guideline is minimal area penalty (<5%) and to avoid any speed penalty. The current plan is to support debug with the following global structures.

20.2.1 Scan

Manufacturing test is the driving force behind the Scan and BiST implementation but this infrastructure also allows for significant debug visibility to internal states. The 21364 model of multiple scan islands, each with multiple scan chains, is also the plan of record for the 21464. This methodology requires scan on only a small percentage of the latches but allows almost any latch to be included in the chain if desired. Detecting difficult stuck-at faults is the most common reason for adding latches to the scan chains, but debug visibility is an equally good reason and designers should be encouraged to make as many important states as possible visible through scan.

When considering what to add to the scan chains for debug remember that although scan is an efficient way to extract the current state of a large number of signals, it is a destructive process that reflects only one instant in time. If the failure symptom is a hang, the problem that caused the hang can hopefully be deduced from the scanable state. When the failure symptom is dynamic (data corruption, application crash, etc.), narrowing in on a point in time where incorrect behavior can be observed through scan is much more difficult.

BiST can help. The BiST engines in the 21464 will have a read-out mode that extracts the internal state into the scan chain. Structures like the register file, pc and branch history tables can be completely dumped through the scan paths. Counters, fifo's or other structures that preserve some extended state can also provide some historical information to the scan dump. CAM structures like the TBs are directly dumpable even through BiST. Additional debug hooks will need to be designed into CAMs if they are to be dumped through scan.

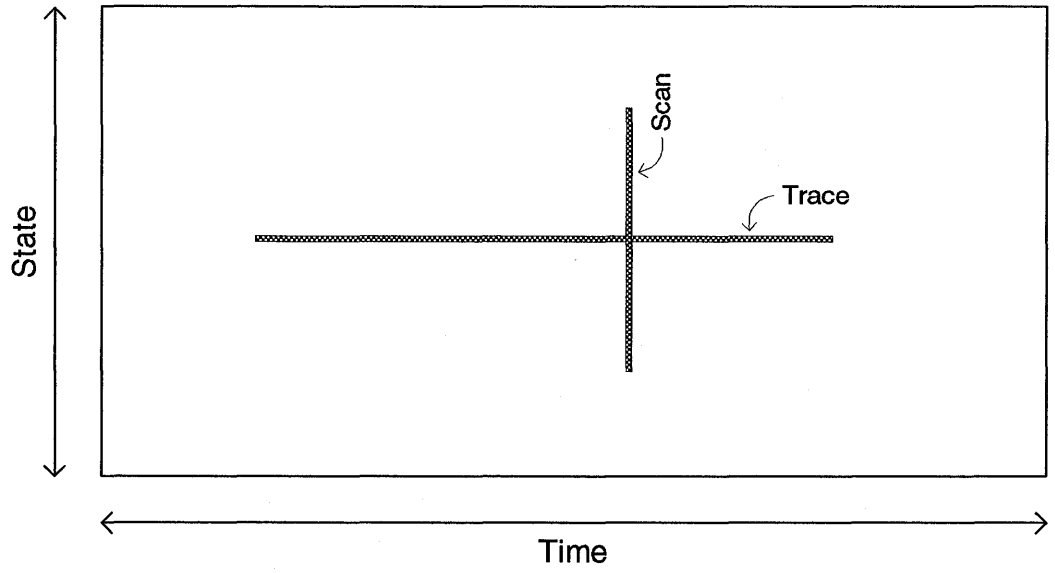
Aside from ensuring the correct information is scanable, the major issue relating to scan for debug is how to activate the scan dump.

20.2.2 Trace Bus

The traditional method of debugging hardware in the lab is to attach a logic analyzer to some number of pins, find a trigger condition near the failure and infer internal operation from the pin trace. Compared to processors of just a few years ago, the 21464 more closely resembles a system on a chip. The information needed to understand the events surrounding most failures is buried deep in the processor inaccessible to traditional oscilloscopes or logic analyzers. Our solution for the 21464 is to embed the logic analyzer functions directly into the silicon.

A logic analyzer type trace differs from a scan dump in that it provides a continuous view of signal state over a large number of cycles. The number of signals that can be traced is very small but the time window is large. The 21464 will be able to trace up to 36 signals over a window of 64 million cycles.

Figure 20–1 Trace Bus Timing Relationships



The signals captured on the trace bus will be dumped off-chip through the redundant RDRAM channel where software can later recover and process the information. Since this channel runs at a fixed 800Mhz, the full 36-bits can only be dumped when the core is running at or below 800Mhz. Between 800Mhz and 1.2Ghz the channel will be limited to 27-bits. Between 1.2Ghz and 1.6 Ghz 18-bits can be dumped and above 1.6Ghz, 9-bits per cycle is all that can be traced.

The selection of which 36 bits to trace will be somewhat programmable but degree of flexibility is still undecided. The simple approach is to allow each box to selectively drive any or all of four 9-bit chunks. Once the list of signals that can be traced is available we can better evaluate whether more sophisticated sharing is necessary.

Each box can make many more than 36 signals available for tracing, but software must select no more than 36 at once. Multiple runs can collect multiple 36-bit traces but exact cycle-to-cycle reproducibility may not be possible and will make correlation of

multiple traces difficult. When selecting signals to be traced, consider generating marker signals that could give a strong correlation among multiple traces. A signal indicating the PC matched some address within a loop would give a strong indication of how to overlay two traces.

20.2.3 Internal Processor Registers

The ability to read/write internal processor registers exits for many reasons other than debug but it also plays a major role in supporting debug. Values that are readable can provide good visibility into the current state of the machine and can allow software to watch for situations that may be causing or symptomatic of a failure. Status bits, aggregate counters and the ProfileMe registers are excellent examples of planned readable Internal Processor Registers that hold valuable information to debug. Debug is a perfectly valid reason to make additional information readable through IPRs.

IPR readability differs from scan and trace by being both non-destructive and immediate. Debug or workaround software that is trying to evaluate if the machine is in trouble could use the information to trigger a scan or trace dump. This is the primary consideration when determining if a signal should be tracable, scanable or readable.

Writable IPR bits are a major part of the low-level infrastructure. The debug process also needs writable IPR bits for two functions. First, IPRs will be used to customize the selection of signals to trace as well as the trigger conditions for the trace bus and scan chains. Writes to flow control or configuration IPRs will also be the primary method to alter chip behavior and avoid conditions that are not reliable. Even if the controls are insufficient to

20.2.4 Derived Signals

None of the methods described above provide visibility to more than a small fraction of the total internal state. Often some form of internal processing can significantly increase the information content. Encoders, programmable counters and comparitors are examples of simple structures that compress the information.

A comparison match against a PC or memory address is an example of a very valuable derived signal. It is only one bit vs. a 64-bit virtual address, makes an excellent marker in a trace file, could be used as a trigger for the scan or trace dump or even a PAL trap that could do some fix-up and prevent a bug from propagating.

20.3 Global Support

20.3.1 Scan

The debug process will leverage the existing scan for manufacturing support as much as possible. Additional capabilities that will be added include:

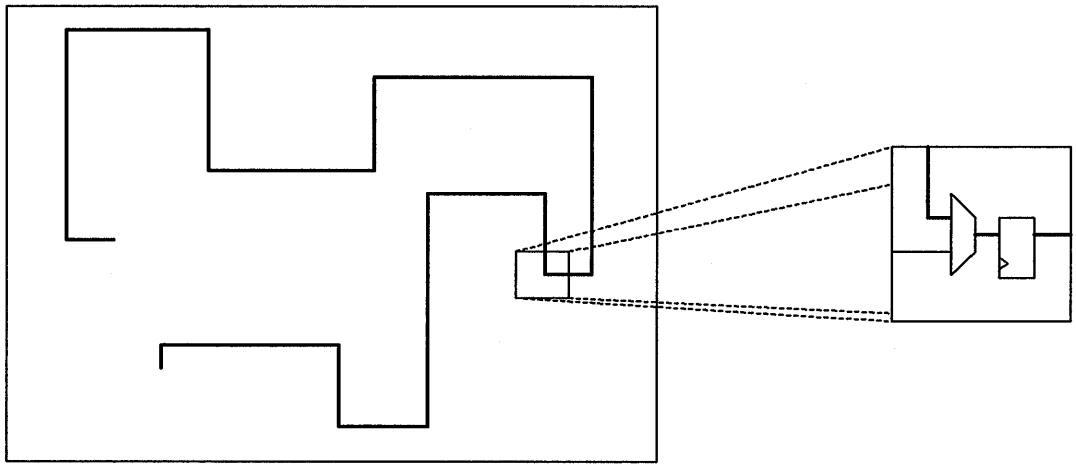
1. The ability for the BiST engines to dump the existing contents of structures onto the scan paths. The traditional BiST engine overwrites the existing contents first then reads and computes a signature. In debug mode, the overwrite phase will be skipped and the reads will be sequencing to deposit the data onto the scan chains.

2. The ability for internal trigger conditions to initiate a scan dump operation. The 21464 will have a fairly elaborate mechanism to detect internal events which may indicate a failure. The output of this detection circuit (the trigger) can be used to initiate a scan dump operation. QUESTION: How does the external logic (that captures the dump) know the scan operation is in progress?

20.3.2 Trace Bus

The Trace bus is a 36-bit bus that winds through the chip touching the section or sections of debug logic in each box. It terminates in the Cbox where the data is multiplexed onto the redundant channel in debug mode. When debug mode is enabled the Cbox will continuously dump until signaled to stop. Every 64M (or is it RDRAM depth?) cycles the addresses will wrap and start overwriting previously collected trace data. The bus will be highly pipelined with latches positioned wherever necessary to avoid timing problems. The bus will take a low priority in routing, utilizing low/slow metal and winding around congestion whenever necessary.

Figure 20–2 Trace Bus Routing



Each point will to either source or repeat the value on to the next point in the chain. An enable bit will be used to quiesce the bus when tracing is not enabled. The trace bus control signals will likely be distributed through the standard IPR mechanisms rather than from a centralized debug control section. Another suggestion was to utilize a serial channel to distribute the controls in much the same way the scan control information is distributed to the control registers in the scan islands.

There is no need to synchronize the injection of data onto the trace bus. Any skew between traced signals relative to the architectural pipeline can be adjusted for when the trace data is extracted and analyzed. This also eliminates any dependencies on the number of stages or order of connection to the trace bus. Once the final structure of the bus is known, it will be important to specify the timing of each traceable signal relative to a common point so software can reassemble and interpret the data.

*** The address of the last location written to the RDRAM will need to be readable so software can unwind the trace. If this is a problem we could burn a bit in the data stream that inverts with each pass, but that would be a waste of a precious resource.

20.3.3 Trigger Logic

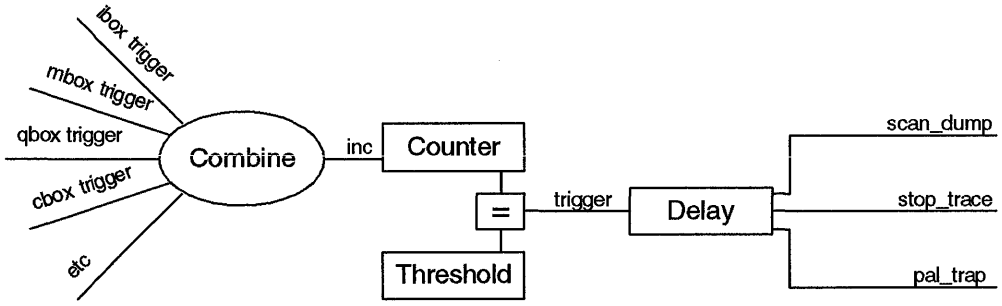
On a 1.5Ghz chip even 64M samples is only 1/20th of a second of information. Some means to stop/initiate the data collection process near the probable cause of a failure is required. The proposed trigger will accept one or two trigger signals from each box, combine them, and count the hits. When the counter reaches some programmable value, the global trigger will assert and cause one of the following events:

- Stop trace capture
- Initiate scan dump
- Trap to PAL

The trigger signals generated by each box are assumed to be a boolean combination of interesting status signals within the box. The Ibox may signal the trigger logic when it detects a specific PC, or detects an Icache miss at a specific PC or takes a cache miss at a specific PC that is a mispredicted branch.

How the global trigger combines the box triggers is still under debate. The simple approaches are to either OR the signals or select a specific signal to use. More complex approaches involve PLA type functions. A more precise definition of the types of trigger conditions that would be useful to create needs to take place before precisely defining the method of combining the signals.

Figure 20–3 Trigger Logic



Another good suggestion was to place a variable delay after the global trigger. If a reproducable trigger could be found relatively near an interesting event, the variable delay would be useful in conjunction with the scan dump to capture several positions near the failure. The variable delay might also be useful to delay stopping the trace dump until the interesting data surrounding the failure has been written into the RDRAMs.

The dynamic range of the two counters is an interesting question. The basic mechanism should degenerate into a "fire after n cycles" (increment every cycle, set threshold to n) or a "fire on first occurrence" (inc when signaled, set threshold to 1). The counter should probably be in the 16-24 bit range, the delay is probably more like a 10-bit value.

PALcode can directly force a scan dump or stop the trace collection. The operational model would be to trigger into PAL and examine the machine state. If not near the expected failure, continue otherwise dump. Writing the threshold to zero, the delay to

zero and setting the scan_dump_en bit would force a scan dump. The trace dump is started by setting an IPR bit and stopped by either clearing the bit or when the trigger fires to stop the trace.

It would also be desirable for external logic to be able to activate the trigger and for the trigger signal to be sent externally. This would allow multiple chips to trigger each other or for an external logic analyzer to trigger or be triggered in conjunction with the internal logic. An external interface, maybe the JTAG port that could trap to PAL and/or manipulate IPRs and memory would allow a simple configurable monitor to be created.

20.4 Box Support

Each box needs to:

1. Identify the states and structures that will be most useful to debug and determine which of the techniques (scan, trace, IPR) is best suited to provide visibility. The box verification teams should play an active role in the definition.
2. Define the interesting trigger conditions and any interactions with other boxes that should be considered when developing the global trigger conditions. The number of triggers sent to the global trigger logic should also be defined.
3. Define the IPRs necessary to support the debug features. These registers should include trigger condition controls, trace bus controls, counter or comparison values and general readable or writable signals for debug. To allow multiple boxes to drive pieces of the trace bus, some type of swizzling (4:1 mux) at each box would ensure that combinations of signals are not inaccessible because they share a fixed bit position.
4. Review the BiSTable structures to ensure the contents are readable for debug. Given the value to debug of most structures large enough to contain BiST, structures that will not be readable should be clearly identified and discussed.
5. Identify the point or points within the box where the trace bus should be routed. The bus needs to be identified in the global floorplan.

The following is a list of items suggested during recent discussions about debug features. It does not represent a complete list or even a committed list but rather a seed for thought as the individual boxes begin to develop official plans.

20.4.1 Ibox

The Ibox must:

- Allow the PC to be traced. Must include thread ID.
- Include a programmable PC comparitor. The output should be tracable as a marker and usable as a trigger.
- Make the TPUalive status bits IPR readable and tracable.
- Trace many of the PR_FE_INFO bits
- Trigger on the profiled instruction issuing.
- Make the PC comparison match tag an instruction for profiling.

20.4.2 Pbox/Qbox

The Pbox and Qbox must:

- Trace INUM allocation and kill sequences.
- Trace LSNUM allocation and high-water marks.
- Consider implementing some model assertion checks as debug trigger conditions.
- Bits to unwedge a hung tpu.

20.4.3 Ebox/Register File

The Ebox and Register File must:

- Encode and trace TPU for all pipes.
- Add programmable instruction trigger. Detect specific opcode/function/tpu/pipeline patterns and trigger on a match.
- BiST dump entire register file contents.

20.4.4 Mbox

The Mbox must:

- Include both virtual and physical address match comparison logic. Trace and trigger against matches.
- Trace inputs to exception funnel.
- Trace the PR_MEM_INFO bits (except VA).

20.5 Software Support

Extracting 64M samples from the RDRAMs and reassembling the data in some useful way will take some assistance from various pieces of system software. Reset modes need to be created that will ensure the information is not lost or overwritten. Boot flags will need to be defined that cause the information to be extracted and saved or left untouched during the boot process. Either PAL or OS hooks will be needed to make the trace data available to an application for processing. And the application that process the trace data needs to be written.

Console, PAL and/or OS hooks will need to be added to allow state to be monitored and trigger conditions manipulated relative to initiating a failure.

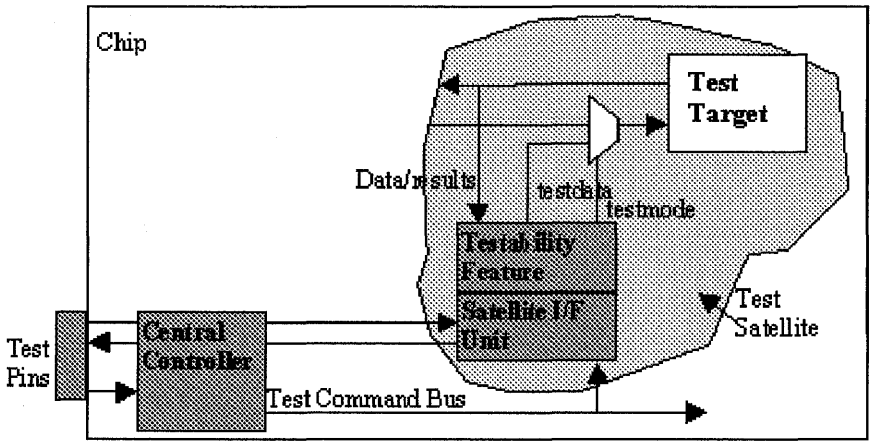
Anyone who has been close to this process in the past is encouraged to help define the software features that will best optimize the debug process.

21
Testability and Diagnostics

The Tbox provides the testability strategy and test solutions. The Tbox provides a comprehensive tester-driven access to the chip's testability features during manufacturing as well as allows a simple automatic chip-initiated access that leverages the same features during normal chip operation. The testability features themselves are scattered throughout the chip, implementing various components of the testability strategy, namely, self-test, self-repair, internal controllability and observability for debug diagnosis, manufacturing test and test pattern development. See [1] for details of the testability strategy.

Figure 21–1 shows the basic contract between the Tbox and a test target in the 21464. A Test Target is simply a functional block under test for which hardware test assist, that is, design for test feature is desired. For example, Icache, Dcache and Scache arrays, Register file, TLB are some test targets. The testability feature implements engines that exercise test algorithms or provide controllability and observability required for testing the test target. The Satellite Interface Unit provides the local control of the testability feature and communicates with the Central Controller for the transport of test commands and test data and results.

Figure 21–1 Basic Tbox Contract



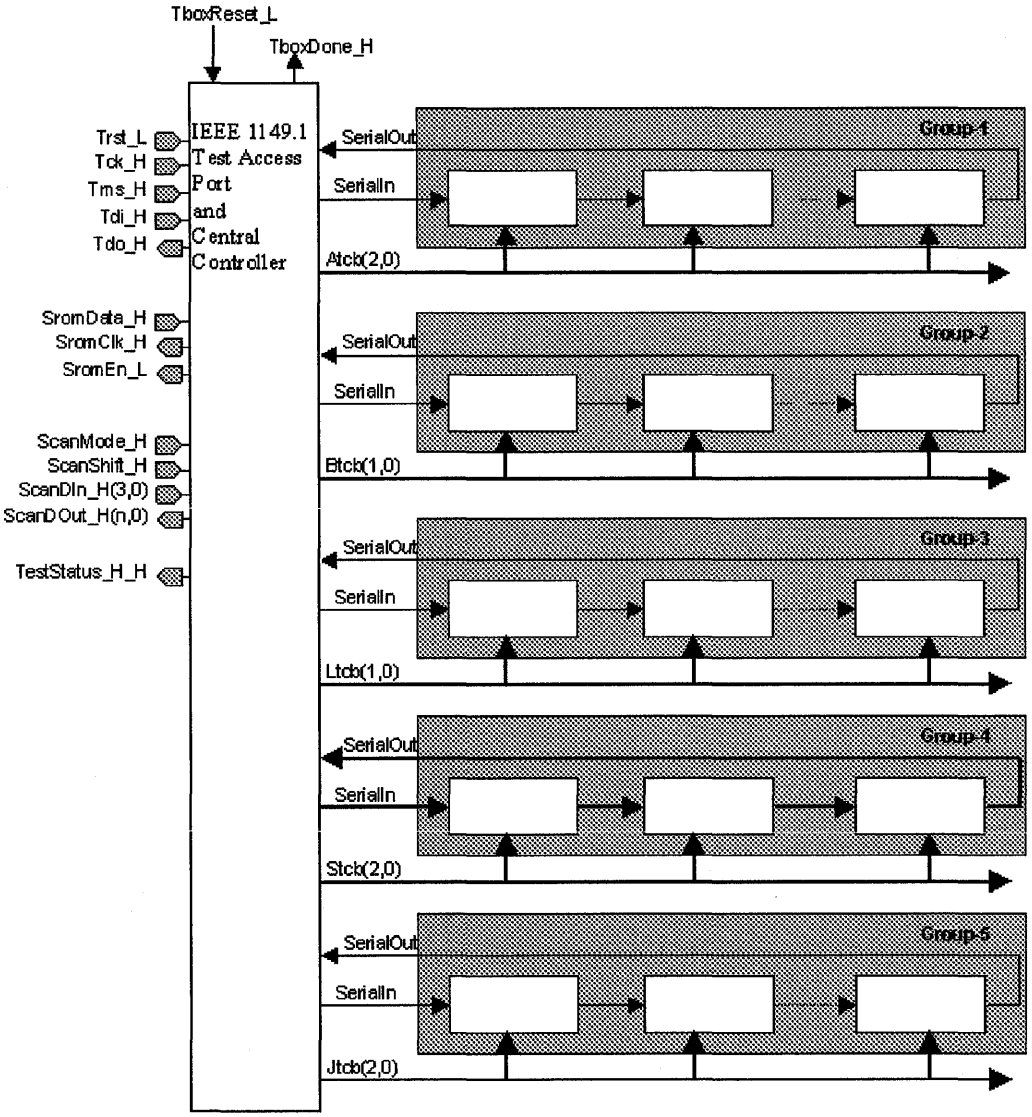
The Test Target, the Testability Feature, the Satellites Interface Unit together make up a generic test satellite. The 21464 has a number of such test satellites. The Central Controller communicates with all test satellites in the chip over dedicated test command broadcast buses and interfaces with the test pins to provide comprehensive and orderly control of and exchange of test data/results with test features.

The Central Controller, the Satellite Interface Units and the Testability features in the distributed test satellites, the tets pins and the various test command broadcast busses and the the test data lines make up the bulk of the Tbox.

21.1 Global Block Diagram

Figure 21–2 shows the global block diagram of the Tbox. The Tbox consists of three basic components: the Central Controller; the distributed test satellites that house the function under test and their testability features; and the test command broadcast buses connecting the Central Controller and the distributed test satellites. The test satellites have serial input and output data ports that are daisy-chained to form scan rings with the Central Controller.

Figure 21–2 Tbox Global Block Diagram



The Central Controller is based on the IEEE 1149.1 TAP. It receives commands from both the IEEE 1149.1 TAP and the automatic chip agents (such as chip's reset state machine), encodes them into command packets and distributes them to the test satellites over five distinct test command broadcast buses.

The test satellites are organized into five general groups. Each is serviced by the Central Controller by a dedicated test command broadcast bus and one or more pairs of serial data lines.

Sections 21.1.1 through 21.1.5 describe the five groups of Satellites.

21.1.1 Group 1 — Array BiST/BiSR Satellites

This group consists of the satellites with large array structures with self-test and self-repair features. The satellites in this group are the most comprehensive. Each has substantial Built-in Self-Test (BiST) and Built-in Self-Repair (BiSR) engines and each has sophisticated Satellite Interface Unit capable of supporting a multitude of local test feature control and exchanging test data with tester via the central controller. This group is accessed both externally from a tester and internally by the chip-initiated automatic actions.

The group is serviced by a 3-wire Array Test Command Bus ATcb_h(2,0). Table 21–1 lists the commands supported ny the bus. The satellite itself is described later. The Icache Data, Tag, and Line Predict arrays, the BHT array, the Dcache Data and Tag arrays, the Register File, the Scache Dtata and Tag arrays are expected to belong to this group.

Table 21–1 Array Test Command Broadcast Bus

Atcb(2,0)	Command	Purpose
111	ATNop	No operation.
110	ATShiftTcr	Shift Test Command Registers (Tcrs) in satellites.
101	ATShiftTdr	Shift the test data registers selected by Tcrs.
100	ATDoIt	Initiate execution.
011	ATDoBiST	Chip initiated simultaneous BiST in satellites.
010	ATDoResult	Chip initiated simultaneous result extraction from satellites.
001	ATDoQuickInit	Chip initiated simultaneous quick-init of embedded RAMs and other structures in satellites.
000	ATDoReset	Reset all satellites.

21.1.2 Group 2 — BiSt Satellites

This group also consists of array structures with only self-test support. The tets feature do not have as many test modes as in the Group 1 satellites and the only darta exchanged with the Central Controller is the Pass/Fail result and occasionally address map. The satellite's test feature consists of simple BiST engine. The satellite interface unit is also simple with limited capabilities.

This group is serviced by a 2-wire test command broadcast bus called BtCB(1,0). Table 21–2 lists the boradcast commands on this bus. A number of smaller enbedded RAM arrays, CAM arrays such as TLBs etc are expected to belong to this group.

Table 21–2 Simple BiSt Command Bus

Btcb(1,0)	Command	Purpose
11	BTNop	DoBiST
10	ATShiftTcr	Shift Test Command Registers (Tcrs) in satellites.
01	ATShiftTdr	Shift the test data registers selected by Tcrs.
00	ATDoReset	Reset all satellites

21.1.3 Group 3 — Observability Registers (LFSRs)

This group consists of the observability registers. Unlike the arrays and their self-test features with multiple modes in the first group, the observability registers are highly uniform and require a simple satellite interface unit. This unit can turn on the observation, shift out the contents and selectively bypass itself in a chain. This group is controlled by the two-wire observability register command bus RTcb_h(1:0). The commands are listed in Table 21–3. .

Table 21–3 Observability Register Command Bus

LCB(1:0)	Command	Purpose
00	RTNop	Observability registers inactive.
01	RTShiftTcr	Shifts control bits in observability register satellite interface units
10	RTShiftTdr	Shifts through observability registers to initialize it or off-load signatures.
11	RTCapture	Captures chip data for test and debug.

21.1.4 Group 4 – Scan Islands (TBD)

21.1.5 Group 5 – Boundary Scan Register

The boundary scan register cells are located at the I/O pins. There is no satellite interface unit, but the broadcast from the Central Controller directly controls each boundary scan register cell.

21.2 Test Pins

The Testability Access Architecture uses both dedicated and some shared pins. Table 21–4 lists the dedicated pins. Table 21–5 lists the shared pins.

Table 21–4 Dedicated Test Port Pins

Pin Name	Type	Function
Tms_H	Input	IEEE 1149.1 test mode select
Tdi_H	Input	IEEE 1149.1 Test data in
Trst_L	Input	IEEE 1149.1 test logic reset

Table 21–4 Dedicated Test Port Pins (Continued)

Pin Name	Type	Function
Tck_H	Input	IEEE 1149.1 test clock
Tdo_H	Output	IEEE 1149.1 test data output
SromData_H	Input	SROM data/Diagnostic terminal data input.
SromClk_H	Output	SROM clock/Diagnostic terminal data output
SromEn_L	Output	SROM enable/Diagnostic terminal enable
ScanMode_H	Input	Scan Mode Control (place holder)
ScanShift_H	Input	Scan shift operation control

Table 21–5 Shared Test Pins

Pin Name	Type	Test Function/Normal Function
TestStat_H	Output	BiST status/timeout output
DumpData0_H(63,0)	Output	Bitmap-LFSR Dump port-0/Tbd
DumpValid0_H	Output	Bitmap-LFSR Sample valid for port-0/Tbd
DumpData1_H(63,0)	Output	Bitmap-LFSR Dump port-1/Tbd
DumpValid1_H	Output	Bitmap-LFSR Sample valid for port-1/Tbd
ScanDataIn_H(3,0)	Input	Scan Data Inputs/Tbd
ScanDOut_H(3,0)	Output	Scan Data Outputs/Tbd

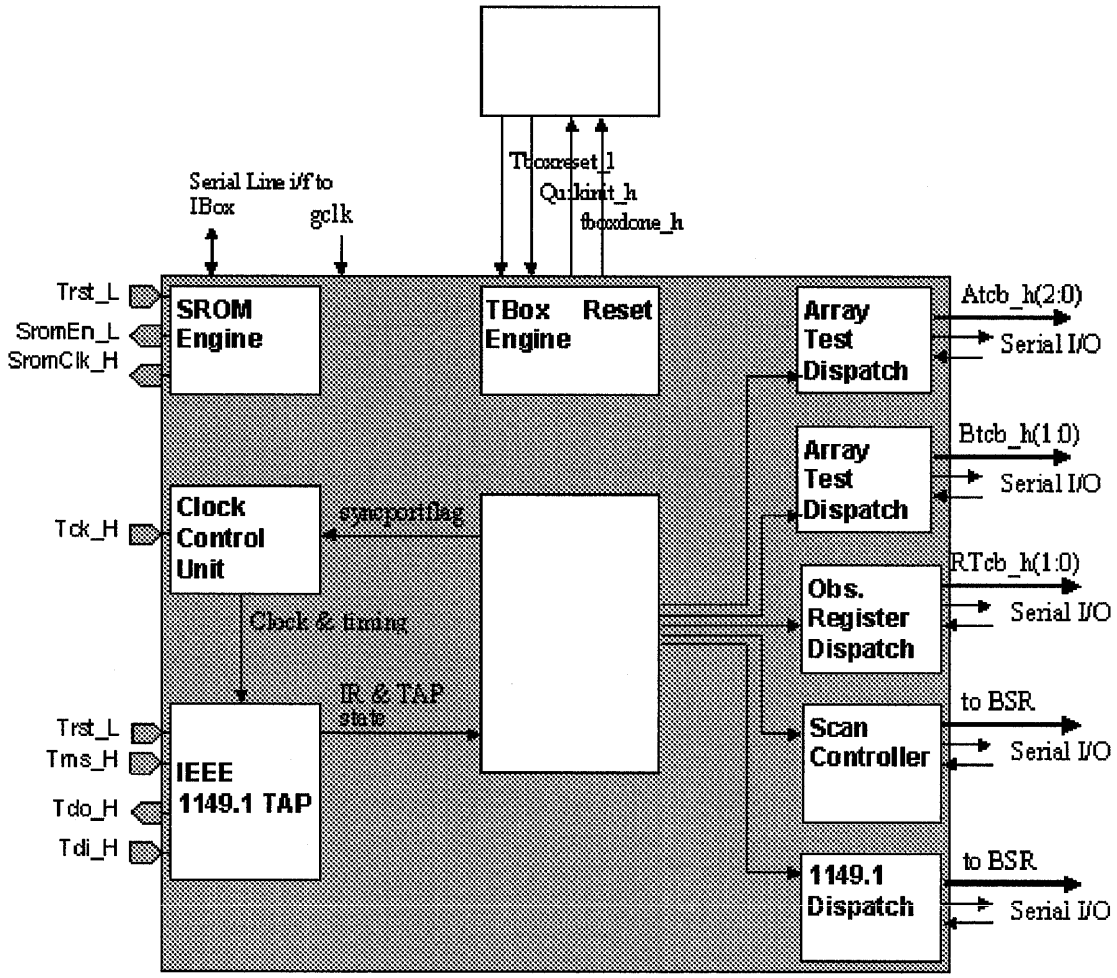
21.3 Central Port Controller

The Central Port Controller links the external world with the testability features. It broadcasts test commands to control test operation exchange test data between the test controller (tester) and the testability features. It is an IEEE 1149.1 based controller that accesses both the standard compliant features and the chip manufacturing test features. The later are accessed synchronously with CPU clock.

Figure 21–3 shows the block diagram of the Central Port Controller, which consists of the:

- IEEE 1149.1 TAP Controller
- Timing Control Unit
- Configuration Flags and Fire Wall
- SROM Engine, Reset Engine
- Output Mux
- Dispatch Units for the IEEE 1149.1
- Cache
- LFSR featues

Figure 21–3 Central Port Controller



21.3.1 IEEE1149.1 Test Access Port Controller

This is the IEEE 1149.1 compliant Test Access Port Controller. The port's pin interface consists of Tdi_H, Tdo_H, Tms_H, Tck_H, and Trst_L pins.

The port supports access to the IEEE 1149.1 mandated public test features as well as several chip manufacturing test features. The scope of 1149.1 complaint features on the 21464 is expected to be limited to the board level assembly verification test. The systems that do not intend to drive this port MUST terminate the port pins as follows: pull-ups on Tdi_H and Tms_H, pull-downs on Tck_H and Trst_L.

The controller is clocked by the Clock Control Unit. It is clocked externally by the test clock Tck_H during normal operation, and internally by the cpu clock during synchronous manufacturing operation.

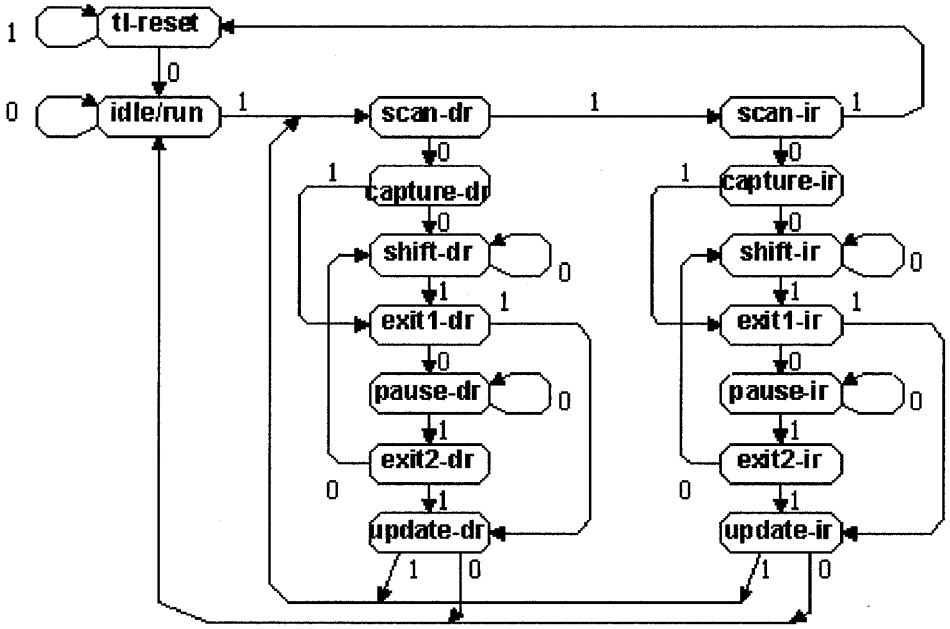
The Port Controller consists of the TAP Controller State Machine, the Instruction Register, the Bypass Register, and the TDO Mux. The Bypass Register provides a short shift path through the chip's IEEE 1149.1 logic. It is generally useful at the board level testing. It consists of a 1-bit shift register.

The Instruction Register holds test instructions. It is 8-bit wide. Section 4.9 lists (Table 8) and describes the instructions supported on the 21464.

Figure 21–4 shows the TAP Controller State Machine state diagram. Tms_H controls the state transitions. The transitions occur with the rising edge of clock.. The TAP state machine states are decoded and used for initiating various actions for testing.

The Output Mux steers the output from the various testability shift registers in the chip to the Tdo_H pin.

Figure 21–4 TAP Controller State Machine



21.3.2 Port Configuration and FireWall Logic

21.3.3 Clock Control Unit

21.3.4 Tbox Reset Engine

The Reset Engine controls the flow of automatic BiST/BISR, self-init, and Icache initialization operations.

The Reset engine consists of the Reset State Machine and the IRESET flag, and the Master BiST Counter. Figure 21–5 shows the flow diagram of the Reset Engine. The engine is triggered upon detection of the chip reset deassertion edge. MRESET and DONTBIST flags determine the path through the flow diagram. Do-Array-Test state either performs the simultaneous BiST or self-init. Do-Results state extracts the result

of BiST from the test satellites. IRESET flag holds reset to the internal chip logic. The flag is set by the chip reset and deasserted by the Reset State machine returning to the Idle state. Figure 21–6 shows the reset engine state machine.

Figure 21–5 Tbox Reset Engine

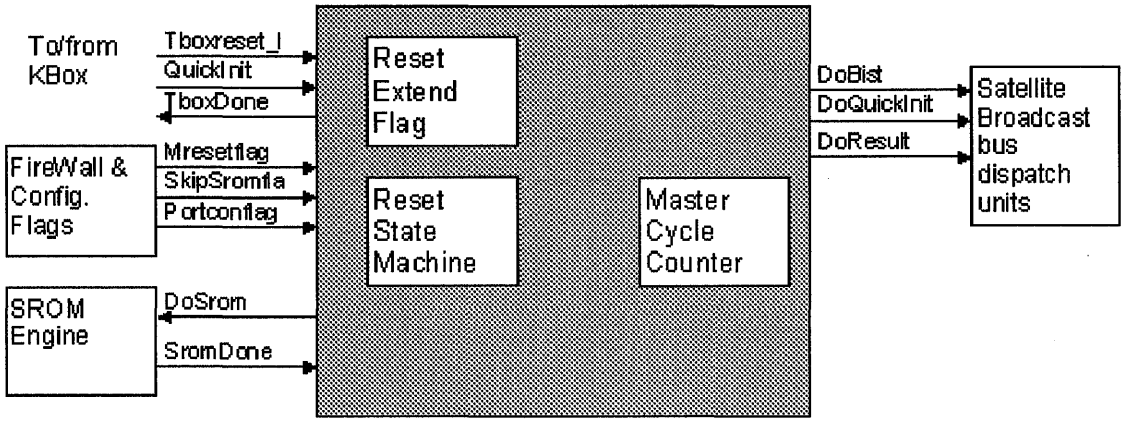
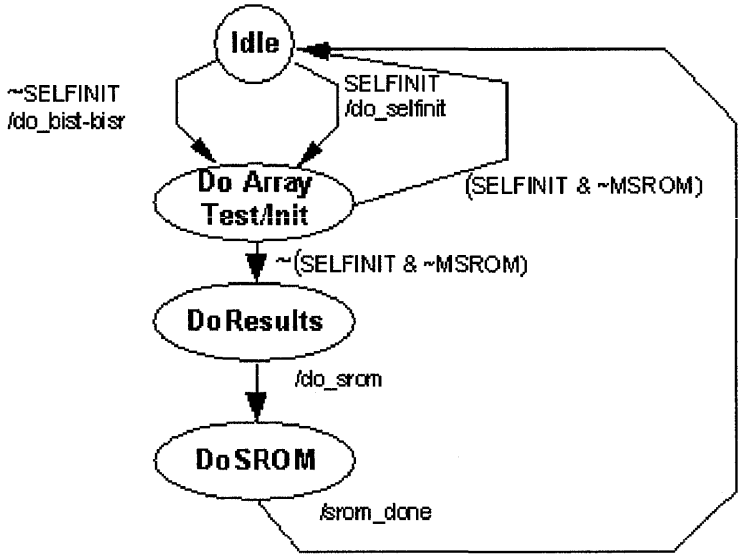


Figure 21–6 Tbox Reset Engine State Diagram



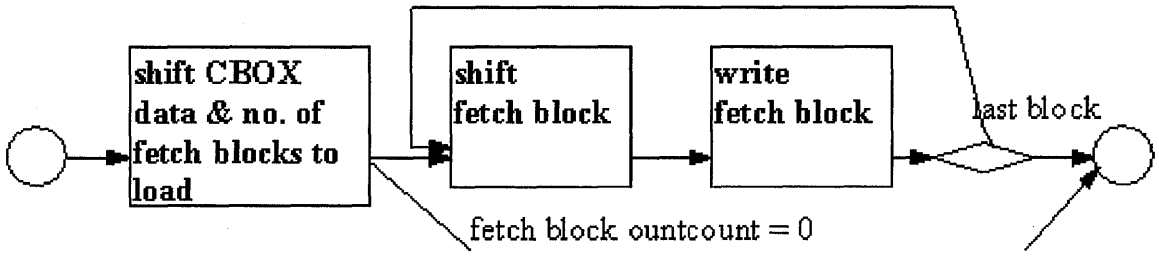
21.3.5 SROM Engine

The SROM Engine controls the serial initialization of the Cbox configuration bits and the instruction cache array. The SROM Engine drives the SROM pin interface as well as controls the shift and write operation in the Icache test satellites. Normal loading occurs at the rate of 1 bit per 256 CPU cycles. If FASTSROM is set, the loading occurs at the rate of one bit per 16 CPU cycles. The maximum rate at which the port may be operated is limited to one-bit per 160 nanoseconds. Thus, the fast SROM loading is usable only if the CPU clock is slowed down, for example, at wafer probe.

Figure 21–7 shows the state diagram of the SROM Engine. The engine is triggered by reset engine upon entry into the DOSROM state. It first loads the Cbox configuration registers followed by the Icache address counter. The

value shifted in the Icache address counter determines the number of fetch blocks to be filled. Unlike the previous generations of Alpha microprocessors, EV6 allows variable amount of Icache to be filled. In the next step, engine loads each fetch-block in reverse order. See Section 9.2 for the details of the SROM map and SROM operation.

Figure 21–7 SROM Engine State Diagram



This port supports two functions. During power-on, It supports automatic initialization of the Cbox configuration registers and the instruction cache (Icache) from the system serial ROMs. After power-on, it supports a serial diagnostic terminal terminal.

During SROM load:

- The Srom_OE_L pin supplies the output enable as well as the reset to the serial ROM. (Refer to the serial ROM specifications for details.) The 21264 asserts this signal low for the duration of the Icache load from the serial ROM. Once the load is complete, the signal remains deasserted.
- The Srom_Data_H pin reads data from SROMs.
- The Srom_Clk_H pin supplies the clock to the SROMs that causes it to advance to the next bit. Simultaneously, it causes the existing data on Srom_Data_H pin to be shifted into an internal shift register. The cycle time of this clock is 256 times the CPU clock rate. (If FASTROM flag is set, the rate is 16 times the CPU clock rate.) The hold time on Srom_Data_H is 2* CPU Cycle time with resepect to the Srom_Clk_H.

Once the Icache load is complete, the port reconfigures into a simple software-timed serial line interface, similar to RS422, that may be used for system debug and diagnosis. In a system the serial line interface is automatically enabled if the Srom_OE_L pin is wired to the active high enable of an RS422 (or 26LS32) driver driving to Srom_Data_H and to the active high enable of an RS422 (or 26LS31) receiver driven from the Srom_Clk_H pin.

After reset, the Srom_Clk_H pin is driven from the sl_xmit bit I_CTL (13) in the Ibox IPR. This IPR is cleared during reset, so it will start driving as a 0, but it can be written and modified by any program. The data becomes available at the pin after retire of the HW_MTPR instruction that write the sl_xmit bit. (Remember that the output only changes after retirement of the HW_MTPR which can take a variable number of cycles depending on machine state.)

On the receive side, while in native mode, any transition on the sl_rcv bit (I_CTL(14) driven from the Srom_Data_H pin result in a trap to the pal interrupt handler (assuming that the serial line interrupt enable bit is set in the SIRR). Once in pal mode, all interrupts are blocked. The interrupt routine can then begin sampling the sl_rcv bit in the I_CTL ipr under a software timing loop to input as much data as needed using whatever

serial line timing protocol chosen. The delay between transition on the pin and interrupt trap is TBD, but probably around 5 cycles or so. For complete description of IPRs associated with this interface refer to IPR Chapter.

21.4 Dot1 Test Decode and Dispatch Logic

This logic controls the operation of the boundary scan register and the Die-ID register. It basically decodes the instructions held in the Port Controller and combines the same with suitable TAP Controller state machine decodes to generate control signals.

When BSRDLY instruction is loaded, bsr_drv_pins_h, bsr_highz_h, bsr_capture_h, bsr_update_h, signals are con ected back-to-back to form a long inverter delay path consisting of 780x4 = 3120 inverters with a nominal delay of approximately 624ns. This delay path may be used for predicting the speed performance of a die.

22
Error Detection and Error Handling

22.1 Disruptions

We need a word for "Interrupts and Exceptions". We will use the word disruption to describe an event that could be either an interrupt or an exception.

A disruption will be delivered - that is, it will start the events in motion that cause the change in program flow - from one of three points in the pipeline.

- Retire-time delivery - Most disruptions will be delivered when the instruction that caused them retires.
- Execution-time delivery - Some disruptions, whose rapid handling is important to performance, will be delivered when their triggering instruction executes.
- Pre-map time delivery - A few disruptions, including all interrupts, will be delivered before the triggering instruction has reached the INum map stage.

Besides delivery time alone, disruptions are divided into several classes. The first division is into Pre-map and Post-map disruptions; the former have no INums associated with them, while the latter do. Post-map disruptions with PALcode handlers are further divided into those delivered at execution time, i.e. DTB misses only, and those delivered at retire time, which includes everything else. Micro-traps, which have no associated PALcode flows, can have either execution-time or retire-time delivery depending on the cause. Pre-map disruptions are subdivided into Internal IBox disruptions, such as ITB misses, and Interrupts proper - both from hardware and software sources. Finally, there is a special class of disruptions, Machine Checks, which signify fatal hardware errors.

The PBox Bid/grant Exception Logic (BEL) prioritizes all post-map disruptions. It puts execution-time disruptions from all sources (not just PALcode-assisted ones) through a structure known as the Exception Funnel (or Efunnel) and picks the oldest. The BEL also monitors whether the Completion Unit (CU) in the QBox is posting a retire-time exception (RTE) for this TPU. RTE's are reported to directly to Completion Unit, or to the QBox Inflight Table (from whence they flow into the CU), depending on at what point in the Araña pipeline they are detected. A valid retire-time exception will always take priority in the BEL, since a retiring instruction is by definition the oldest in the CPU. The IBox uses an algorithm to arbitrate between pre-map and post-map disruptions. Briefly, every cycle the IBox services disruptions with the following priority:

1. Post-map
2. IBox internal

3. Interrupts

Interrupts are postponed in favor of IBox internal disruptions, which are deferred in favor of post-map disruptions. If the IBox has decided to take a post-map disruption, it stores the INum of the disrupting instruction and tells the BEL to broadcast the kill. The IBox will accept no younger post-map disruptions for a fixed period of time - enough for the kill to take effect across the chip, but not so long as to ignore valid disruptions on the new good path. However, the IBox will restart the disruption flow if an older disruption is signaled. For taken post-map disruptions, the BEL vectors the IBox into the appropriate PALcode flow, while on pre-map disruptions the IBox redirects itself. For all taken PALcode-assisted disruptions, IBox starts fetching down the PALcode path after saving the correct return PC. Note that there is a special class of interrupt, RESET/WAKEUP, which has a PALcode entry point that the IBox only vectors into when a TPU is restarted or woken from sleep mode.

It should be noted that there is an important consequence to delaying the taking of certain disruptions until retire time. This means that all of the data needed to identify and rectify the disruption must be stored somewhere in the time between the error event and the retirement of the instruction. Although the details are still being worked out, our current plan is to store most of the retire-time disruption type information in the Completion unit in a compressed form. We have also defined, coded up, and reviewed the Virtual Register Table (VRT) in the PBox, which supplies the virtual source or destination of the faulting instruction (depending on the exception type) for a particular class of disruptions that require it.

Keep in mind that there are many error cases that can occur in different parts of the chip which do not rise to the level of disruptions. For instance, ICache misses and line mispredictions are not only not visible architecturally, but they are handled entirely within the IBox without intervention or assistance from any other part of the CPU.

22.1.1 High-Level Features

Text goes here.....

Table 22–1 Key to Table 22–2, "Summary of Disruption High-Level Features'

Heading	Meaning
Name:	Name of exception, interrupt, trap, and so forth, such as Integer Overflow.
Posted Time	Point in time when disruption is delivered.

Values	Meaning
Interrupt	Asynchronous with respect to instruction stream, lower priority than other disruptions. Reported to Ibox PCC.
N-M Interrupt	At interrupt time but Non-Maskable - even by PALcode. Reported to Ibox PCC.
Reset	At interrupt time but taken unconditionally — that is, with highest priority of all disruptions. Reported to Ibox PCC.
Pre-map	Prior to mapping INum assignment of disrupting instruction. Reported to Ibox PCC
Execution	After mapping of disrupting instruction. Reported to Pbox BEL.
Retire	When disrupting instruction is next eligible to retire. Reported to Qbox CMP.

Restart PC: Virtual address of post-disruption good path after handler (if any) relative to PC of disrupting instruction or interrupt victim.

Values	Meaning
PC	Disrupting instruction or interrupt victim.
PC + 4	Instruction after disrupting instruction.
CBR/FCBR Target	True branch target of mispredicted (integer/floating) conditional branch.
Jump Target	True jump target of mispredicted jump.
n/a	Code does not return from handler.

Kill Point: Location of kill relative to disrupting INum.

Values	Meaning
At	Kill disrupting instruction and all younger in TPU
After	Kill instruction after disrupting instruction and all younger in TPU
n/a	Not relevant — for example, is for pre-map disruptions)

PALcode Entry Point:	Name of PALcode disruption entry if any (for example, DTBM_SINGLE).
Description:	Textual explanation of disruption meaning, purpose, function, etc.

Table 22–2 Summary of Disruption High-Level Features

Name	Posted Time	Restart PC	PALcode Entry Point[1]	Kill Point	Description
Ibox Disruptions					
Bad Jump Istream VA	Retire	PC[2]	BAD_JUMP_IVA	At	Jump target is outside of current virtual address space
Ibox Debug Trap	Retire	PC	n/a	At	Placeholder for chip debug exception from Ibox

Table 22–2 Summary of Disruption High-Level Features (Continued)

Name	Posted Time	Restart PC	PALcode Entry Point[1]	Kill Point	Description
Istream Access Violation	Pre-map	PC[2]	IACV	n/a	Istream access violation (prvilege mismatch or walk/branch out of IVA space)
ITB Miss Single	Pre-map	PC	ITB_MISS	n/a	Single level ITB miss (not in console mode)
ITB Miss Single Console	Pre-map	PC	ITB_MISS_CONS	n/a	Single level ITB miss while in console mode
Jump Mispredict	Execution	Jump Target	n/a	After	Predicted Jump Target did not match true Jump Target
Uncorrectable Istream ECC Error	Pre-map	PC[2]	IMCHK	n/a	Instruction fetch experienced an uncorrectable ECC error
Ebox Disruptions					
Add Overflow	Retire	PC + 4[2]	ARITH	After	Add/Subtract operation overflowed/ underflowed
CBR Mispredict	Execution	CBR Target	n/a	After	An integer conditional branch instruction was incorrectly predicted
Ebox Debug Trap	Retire	PC	n/a	At	Placeholder for chip debug exception from Ebox
Floating-Point Disabled Fault	Retire	PC[2]	FEN	At	A legal FP instruction issued while the Floating-Point Enable (FPE) bit was deasserted
IFETCHB Issued	Retire	PC + 4	n/a	After	A IFETCHB instruction was executed
Illegal Instruction	Retire	PC[2]	OPCDEC	At	Thread not allowed to execute this instruction, or invalid opcode/function
Mul Overflow	Retire	PC + 4[2]	ARITH	After	Integer multiply operation overflowed/underflowed
Native Mode MT_FPCR Issued	Retire	PC + 4	MT_FPCR	After	An MT_FPCR instruction has issued in user mode
Mbox Disruptions					
Bad VA Alignment	Execution	PC[2]	UNALIGN	At	Computed virtual address LSBs (VA<2:0>) not legal for datatype
Bad VA Sign	Execution	PC[2]	DFAULT	At	Computed Dstream virtual address sign extension (VA<63:52>) not correct
Dstream Access Violation	Execution	PC[2]	DFAULT	At	Process has insufficient privileges to load to/store from this page
DTB Miss Double	Execution	PC	DTBM_DOUBLE	At	DTB miss on LD_VPTE with default page table configuration
DTB Miss Double Alternate	Execution	PC	DTBM_DOUBLE_ALT	At	DTB miss on LD_VPTE with alternate page table configuration
DTB Miss Single	Execution	PC	DTBM_SINGLE	At	Single level DTB miss (not in console mode)
DTB Miss Single Console	Execution	PC	DTBM_SINGLE_CONS	At	Single level DTB miss while in console mode
Fault On (Read/Write)	Execution	PC[2]	DFAULT	At	Fault on Read or Fault on Write bits set in PTE for this VA

Table 22–2 Summary of Disruption High-Level Features (Continued)

Name	Posted Time	Restart PC	PALcode Entry Point[1]	Kill Point	Description
Load Data Parity Error	Execution	PC	n/a	At	Data returned from the Dcache had bad parity
Load Double-Bit ECC Error	Execution	PC[2]	MCHK	At	Load tag or data experienced an uncorrectable ECC error
Load ErrResp from Memory	Execution	PC[2]	MCHK	At	Memory system returned an Error Response on a load
Load Invalidate	Execution	PC	n/a	At	TPU received an invalidate probe
Load NXMResp from Memory	Execution	PC[2]	MCHK	At	Load attempted from Non-eXistent Memory
Load Rambus Uncorrectable Error	Execution	PC[2]	MCHK	At	Rambus interface detected an uncorrectable error on a load
Load Single-Bit ECC Error	Execution	PC	n/a	At	Load tag or data experienced a correctable ECC error
Load Tag Parity Error	Execution	PC	n/a	At	Tag matching load VA experienced a parity error
Load/Store Order Violation	Execution	PC	n/a	At	Store executed out of order with respect to a load
Load/Store Synonym Detection	Execution	PC	n/a	At	Mbox has detected a virtual-to-physical alias
Mbox Debug Trap	Execution	PC	n/a	At	Placeholder for chip debug exception from Mbox
QUIESCE	Execution[3]	PC + 4	n/a	After	A QUIESCE instruction is about to retire
Fbox Disruptions					
FCBR Mispredict	Execution	FCBR Target	n/a	After	A floating conditional branch instruction was incorrectly predicted
FP Trap (SW = 0)	Retire	PC + 4[2]	ARITH	After	Floating-Point trap without software completion
FP Trap (SW = 1)	Retire	PC + 4[2]	ARITH	After	Floating-Point trap with software completion
FPCR Update	Retire	PC + 4	ARITH	After	Fbox requests an update of the FPCR
Pbox/Qbox Disruptions					
Pbox/Qbox Debug Trap	Retire	PC	n/a	At	Placeholder for chip debug exception from Pbox/Qbox
Cbox Reset Interrupts					
Cold Reset	Reset	n/a	MILD_RESET[4]	n/a	Cold start (power-on or platform/ remote reset) - initialize all state, run SROM and BIST
Fast Reset	Reset	n/a	FAST_RESET	n/a	Reset after loss of lockstep - initialize core/caches, no SROM, no BIST - e.g. Tandem re-sync
Mild Reset	Reset	n/a	MILD_RESET	n/a	Reset after core HW error - initialize core, no SROM, no BIST

Table 22–2 Summary of Disruption High-Level Features (Continued)

Name	Posted Time	Restart PC	PALcode Entry Point[1]	Kill Point	Description
Tepid Reset	Reset	n/a	MILD_RESET[4]	n/a	Reset after system HW error - initialize core, system/memory interfaces, run SROM, no BIST
TPU Restart	N-M Interrupt	n/a	TPU_RESTART	n/a	Another TPU has requested a restart
Wakeup	Reset	n/a	WAKEUP	n/a	Wakeup from sleep mode - initialize core/caches, no SROM, no BIST
Cbox Service/Error Interrupts					
ALERT Interrupt	Interrupt	PC[2]	INTERRUPT	n/a	A remote CPU has signaled an ALERT
External Interrupt	Interrupt	PC[2]	INTERRUPT	n/a	External hardware interrupt
IP Bus Correctable Error	Interrupt	PC[2]	INTERRUPT	n/a	Switchport experienced a correctable (single-bit) ECC error
IP Bus Uncorrectable Error	Interrupt	PC[2]	INTERRUPT	n/a	Switchport experienced an uncorrectable (double-bit) ECC error
ProfileMe Service	Interrupt	PC[2]	INTERRUPT	n/a	Data collection for a ProfileMe instruction pair is complete
Rambus Correctable Error	Interrupt	PC[2]	INTERRUPT	n/a	Rambus experienced an correctable (single-bit/RAID-correctable multi-bit) ECC error
Rambus Uncorrectable Error	Interrupt	PC[2]	INTERRUPT	n/a	Rambus experienced an uncorrectable (double-bit/RAID-uncorrectable) ECC error
Scache Data Correctable ECC Error	Interrupt	PC[2]	INTERRUPT	n/a	Second-level cache data experienced a correctable (single-bit) ECC error
Scache Tag Correctable ECC Error	Interrupt	PC[2]	INTERRUPT	n/a	Second-level cache tag experienced a correctable (single-bit) ECC error
Scache Uncorrectable ECC Error	Interrupt	PC[2]	INTERRUPT	n/a	Second-level cache tag or data experienced an uncorrectable (double-bit) ECC error
Software Interrupt	Interrupt	PC[2]	INTERRUPT	n/a	Software interrupt
TPU PALmode Timeout	N-M Interrupt	PC[2]	INTERRUPT	n/a	A TPU has been in PALmode too long
Cbox Logging Interrupts					
Dcache Parity Error	Interrupt	PC[2]	INTERRUPT	n/a	Data cache tag or data experienced a parity error
Icache Parity Error	Interrupt	PC[2]	INTERRUPT	n/a	Instruction cache tag or data experienced a parity error
Load IP Bus Parity Error	Interrupt	PC[2]	INTERRUPT	n/a	Load switchport experienced a parity error
Oustanding DIFT Entry Timeout	Interrupt	PC[2]	INTERRUPT	n/a	A forwarded DIFT entry has been outstanding too long

Table 22–2 Summary of Disruption High-Level Features (Continued)

Name	Posted Time	Restart PC	PALcode Entry Point[1]	Kill Point	Description
Outstanding MAF Entry Timeout	Interrupt	PC[2]	INTERRUPT	n/a	A MAF entry has been outstanding too long
Store IP Bus Parity Error	Interrupt	PC[2]	INTERRUPT	n/a	Store switchport experienced a parity error
TPU Inst. Retirement Timeout	Interrupt	PC[2]	INTERRUPT	n/a	A TPU has not retired any instructions for too long

1 See Table 22–3

2 For these PALcode traps, the Restart PC is nominal; the PALcode handler may elect to not return to the trapping code flow (e.g. in the case of uncorrectable errors). However, this value still needs to be saved in the appropriate IPR.

3 The disruption is reported via the execution-time interface, but only when the disrupting instruction is reported as next-to-retire on the Retire/Kill bus.

4 Since both Cold Reset and Tepid Reset execute from SROM code, which essentially has its own address space, the 21464 overlays their entry points on top of the one for Mild Reset.

Table 22–3 Disruption PALcode Entry Points

Disruption PALcode Entry Points	PC	IPRs Implicitly Written
Reserved[1]	PB + x000	n/a
Available	PB + x080	n/a
DTBM_DOUBLE	PB + x100	EXC_ADDR
DTBM_DOUBLE_ALT	PB + x180	EXC_ADDR
FEN	PB + x200	EXC_ADDR
UNALIGN	PB + x280	EXC_ADDR, EXC_SUM, VA, VA_FORM, M_STAT
DTBM_SINGLE	PB + x300	DTBMS_RET_ADDR, EXC_SUM, VA, VA_FORM, M_STAT
DFAULT	PB + x380	EXC_ADDR, EXC_SUM, VA, VA_FORM, M_STAT
OPCDEC	PB + x400	EXC_ADDR
IACV	PB + x480	EXC_ADDR
MCHK	PB + x500	EXC_ADDR, M_STAT
ITB_MISS	PB + x580	EXC_ADDR, IVA_FORM
ARITH	PB + x600	EXC_ADDR, EXC_SUM
INTERRUPT	PB + x680	EXC_ADDR
MT_FPCR	PB + x700	EXC_ADDR
IMCHK	PB + x780	EXC_ADDR
DTBM_SINGLE_CONS	PB + x800	DTBMS_RET_ADDR, EXC_SUM, VA, VA_FORM, M_STAT
ITB_MISS_CONS	PB + x880	EXC_ADDR, IVA_FORM
BAD_JUMP_IVA	PB + x900	EXC_ADDR, EXC_SUM
FAST_RESET	PB + x980	n/a
WAKEUP	PB + xA00	n/a
TPU_RESTART	PB + xA80	n/a

Table 22–3 Disruption PALcode Entry Points (Continued)

Disruption PALcode Entry Points	PC	IPRs Implicitly Written
MILD_RESET	PB + xB00	n/a
DST_NXM	PB + xB80	EXC_ADDR
Available	PB + xC00	n/a
Available	PB + xC80	n/a
Available	PB + xD00	n/a
Available	PB + xD80	n/a
Available	PB + xE00	n/a
Available	PB + xE80	n/a
Available	PB + xF00	n/a
Available	PB + xF80	n/a

[1] PB + x000 is reserved as entry point FROM the Swap PALcode (CALL_PAL SWPPAL) routine or the SROM boot codeinto the RESET code sequence.

22.1.2 Low-Level Features

Text here.....

Table 22–4 Key to Table 22–5, "Summary of Disruption Low-Level Features'

Heading	Meaning
Name:	Name of exception, interrupt, trap, and so forth, such as Integer Overflow.
Detected By:	box responsible for detecting the disruption.
EType Code:	Encoding of exception type communicated to the Ibox to determine its restart address (symbolic name defined in `global/arana_traps.mnh`)
Completion Prevention:	Method of preventing retirement of disrupting instruction or interrupt victim.

Values	Meaning
Map	Never mapped.
Inflight	Invalidated in inflight table.
Zap	Zapped/retire stalled in completion unit.

Any other text here....

Table 22–5 Summary of Disruption Low-Level Features

Name	Kill Point	Detected By	EType Code	Completion Prevention
Ibox Disruptions				
Bad Jump Istream VA	At	Ibox	AT_RTE_BAD_JUMP_IVA	Inflight
Ibox Debug Trap	At	Ibox	AT_RTE_I_DBG	Inflight

Table 22–5 Summary of Disruption Low-Level Features (Continued)

Name	Kill Point	Detected By	EType Code	Completion Prevention
Istream Access Violation	n/a	Ibox	n/a	Map
ITB Miss Single	n/a	Ibox	n/a	Map
ITB Miss Single Console	n/a	Ibox	n/a	Map
Jump Mispredict	After	Ibox	AT_ETE_JMP_MISPRED	Inflight
Uncorrectable Istream ECC Error	n/a	Ibox	n/a	Map
Ebox Disruptions				
Add Overflow	After	Ebox	AT_RTE_IOVF	Inflight
CBR Mispredict	After	Ebox	AT_ETE_CBR_MISPRED	Inflight
Ebox Debug Trap	At	Ebox	AT_RTE_E_DBG	Inflight
Floating-Point Disabled Fault	At	Ebox	AT_RTE_FPDIS	Inflight
IFETCHB Issued	After	Ebox	AT_RTE_IFETCHB	Inflight
Illegal Instruction	At	Ebox	AT_RTE_OPCDEC	Inflight
Mul Overflow	After	Ebox	AT_RTE_IOVF	Inflight
Native Mode MT_FPCR Issued	After	Ebox	AT_RTE_MT_FPCR	Inflight
Mbox Disruptions				
Bad VA Alignment	At	Mbox	AT_ETE_BADVA	Zap
Bad VA Sign	At	Mbox	AT_ETE_DST	Zap
Dstream Access Violation	At	Mbox	AT_ETE_DST	Zap
DTB Miss Double	At	Mbox	AT_ETE_DTB_DBL	Inflight
DTB Miss Double Alternate	At	Mbox	AT_ETE_DTB_DBL_ALT	Inflight
DTB Miss Single	At	Mbox	AT_ETE_DTB_SING	Inflight
DTB Miss Single Console	At	Mbox	AT_ETE_DTB_SING_CONS	Inflight
Fault On (Read/Write)	At	Mbox	AT_ETE_DST	Zap
Load Data Parity Error	At	Mbox	AT_ETE_DST_RPLAY	Zap
Load Double-Bit ECC Error	At	Mbox	AT_ETE_DST_MCHK	Zap
Load ErrResp from Memory	At	Mbox	AT_ETE_DST_MCHK	Zap
Load Invalidate	At	Mbox	AT_ETE_DST_RPLAY	Zap
Load NXMResp from Memory	At	Mbox	AT_ETE_DST_MCHK	Zap
Load Rambus Uncorrectable Error	At	Mbox	AT_ETE_DST_MCHK	Zap
Load Single-Bit ECC Error	At	Mbox	AT_ETE_DST_RPLAY	Zap
Load Tag Parity Error	At	Mbox	AT_ETE_DST_RPLAY	Zap

Table 22–5 Summary of Disruption Low-Level Features (Continued)

Name	Kill Point	Detected By	EType Code	Completion Prevention
Load/Store Order Violation	At	Mbox	AT_ETE_LDST_ORDER	Zap
Load/Store Synonym Detection	At	Mbox	AT_ETE_DST_RPLAY	Zap
Mbox Debug Trap	At	Mbox	AT_RTE_M_DBG	Zap
QUIESCE	After	Mbox	AT_ETE_QUIESCE	Zap
Fbox Disruptions				
FCBR Mispredict	After	Ebox	AT_ETE_CBR_MISPRED	Inflight
FP Trap (SW = 0)	After	Fbox	AT_RTE_SW0 [110xxxx]	Inflight
FP Trap (SW = 1)	After	Fbox	AT_RTE_SW1 [111xxxx]	Inflight
FPCR Update	After	Fbox	AT_RTE_FPCR [101xxxx]	Inflight
Pbox/Qbox Disruptions				
Pbox/Qbox Debug Trap	At	Pbox/Qbox	AT_RTE_PQ_DBG	Inflight
Cbox Reset Interrupts				
Cold Reset	n/a	Cbox	n/a	Map
Fast Reset	n/a	Cbox	n/a	Map
Mild Reset	n/a	Cbox	n/a	Map
Tepid Reset	n/a	Cbox	n/a	Map
TPU Restart	n/a	Cbox	n/a	Map
Wakeup	n/a	Cbox	n/a	Map
Cbox Service/Error Interrupts				
ALERT Interrupt	n/a	Cbox	n/a	Map
External Interrupt	n/a	Cbox	n/a	Map
IP Bus Correctable Error	n/a	Cbox	n/a	Map
IP Bus Uncorrectable Error	n/a	Cbox	n/a	Map
ProfileMe Service	n/a	Ibox	n/a	Map
Rambus Correctable Error	n/a	Cbox	n/a	Map
Rambus Uncorrectable Error	n/a	Cbox	n/a	Map
Scache Data Correctable ECC Error	n/a	Cbox	n/a	Map
Scache Tag Correctable ECC Error	n/a	Cbox	n/a	Map
Scache Uncorrectable ECC Error	n/a	Cbox	n/a	Map

Table 22–5 Summary of Disruption Low-Level Features (Continued)

Name	Kill Point	Detected By	EType Code	Completion Prevention
Software Interrupt	n/a	Cbox	n/a	Map
TPU PALmode Timeout	n/a	Qbox	n/a	Map
Cbox Logging Interrupts				
Dcache Parity Error	n/a	Mbox	n/a	Map
Icache Parity Error	n/a	Ibox	n/a	Map
Load IP Bus Parity Error	n/a	Mbox	n/a	Map
Oustanding DIFT Entry Timeout	n/a	Cbox	n/a	Map
Outstanding MAF Entry Timeout	n/a	Cbox	n/a	Map
Store IP Bus Parity Error	n/a	Mbox	n/a	Map
TPU Inst. Retirement Timeout	n/a	Qbox	n/a	Map

23 Hardware Interface

23.1 Signal Pad Requirements

Table 23–1 lists the signal pad requirements for the 21464.

Table 23–1 Signal Pad Requirements

Signal	I/O/B	Pins	Type	Description
RamDataA_L(8,0)	B	9	RSL	RAM Data
RamDataB_L(8,0)	B	9	RSL	RAM Data
RamRow_L(2,0)	O	3	RSL	RAM Row Control
RamCol_L(4,0)	O	5	RSL	RAM Column Control
RamClkToMaster_H	I	1	RSL	RAM Receive Clock
RamClkToMaster_L	I	1	RSL	RAM Receive Clock
RamClkFromMaster_H	I	1	RSL	RAM Transmit Clock
RamClkFromMaster_L	I	1	RSL	RAM Transmit Clock
RamCMD	O	1	CMOS	RAM Control register command
RamSCK	O	1	CMOS	RAM Control register clock
RamSIO(1,0)	B	2	CMOS	RAM Serial rd/wr data for register (daisy chained)
RamVRef	I	1	Analog	RAM Reference Voltage for above signals
RamVTerm	I	1	Analog	RAM Termination Voltage for above signals
RamSCL	O	1	CMOS	RAM Presence Detect Clock
RamSDA	B	1	CMOS OC	RAM Presence Detect Data
RamClkOut_L	O	1	?	RAM 400 Mhz clock for distribution to RClk/TClk
Subtotal Per-Rambus	—	**39**	—	**Subtotal Rambus Signals**
PortData_L(55,0)	I or O	56	?	Port Data
PortClock_H(2,0)	I or O	6	?	Port Clock
PortVRef	IAnalog	1	Analog	Port Reference Voltage for Data & Clock
Subtotal Per-Port	—	**63**	—	**Subtotal Port Signals**
Srom_Data_H	I	1	?	Serial ROM data/receive data
Srom_Clk_H	O	1	?	Serial ROM clock/transmit data
Srom_OE_L	O	1	?	Serial ROM output enable

Table 23–1 Signal Pad Requirements

Signal	I/O/B	Pins	Type	Description
Tdi_H	I	1	?	JTAG test data in
Tdo_H	O	1	?	JTAG test data out
Trst_L	I	1	?	JTAG test reset
Tck_H	I	1	?	JTAG test clock
Tms_H	I	1	?	JTAG test mode select
TestStat_H	O	1	?	Test???
ClkIn_H	I	1	?	Clock input, differential
ClkIn_L	I	1	?	Clock input, differential
reset_L	I	1	?	Processor reset
DcOK_H	I	1	?	System DC power OK
PllBypass_H	I	1	?	Bypass internal PLL
PllVdd	IAnalog	1	?	PLL Supply voltage
VddSel	IAnalog	1	?	Supply selection
Subtotal Common	—	**16**	—	**Subtotal Common Signals**
Subtotal Rambus Signals	—	**39*10**	—	**Subtotal Rambus Signals**
Subtotal Port Signals	—	**63*10**	—	**Subtotal Port Signals**
Subtotal Common Signals	—	**16**	—	**Common Signals**
Total Signals	—	**1036**	—	**Total Signals**

24
New Instructions

//This is a place holder for this chapter.//

25
System Configurations

//This is a placeholder for a new chapter.//

26

Physical Addressing and Input/Output

27

Requirements to Support "Tandem"

A
Instruction Decoding

This appendix defines the exact behavior of instruction decoding in the 21464. This is not a rewrite of the Alpha System Reference Manual (the SRM). Rather, it is a clarification of some of the exact implementation details. The target audience is the design and verification teams, but the information might also be useful to compiler developers or anyone who generates assembly code by hand.

The instruction set is organized in ascending order, according to opcode value, or by instruction type for Load and Store, Jump and Branch, and PALcode instructions.

Instruction decoding is a distributed event. The Ibox, Pbox, Ebox, Fbox, and Mbox all decode portions of the Istream. To ease verification, we want to ensure that all boxes that decode instructions make the same assumptions. For the instructions defined by the SRM, this is straightforward, but there are many unused function codes and combinations of instruction bits that are only defined by the SRM as producing UNPREDICTABLE behavior. Verifying that several boxes that separately decode UNPREDICTABLE instructions do not cause hangs or otherwise violate the requirements of the SRM would be a tedious task at best.

This appendix specifies an instruction decoding that uniquely maps all unused function codes to a known behavior. Assuming that all boxes in the 21464 use this decoding scheme, behavior is easily predictable and we will avoid cross-box bugs where the instruction stream was interpreted differently.

Because of the large number of instructions and function codes, the decoding descriptions are broken into opcode groups.

Table A–1 Opcode Groups

Opcode	Type	Format	In Section
00	Call_PALL	PALcode	A.6.1
01-07	Reserved	PALcode	A.6.2
08-0F	Load and store	Memory Displacement	A.6.12
10	Integer add/sub/compare	Integer Operate	A.6.3
11	Integer logical	Integer Operate	A.6.4
12	Integer shift	Integer Operate	A.6.5
13	Integer multiply	Integer Operate	A.6.6
14	ITOFx and FSQRT	Floating Operate	A.6.7
15	VAX floating-point	Floating Operate	A.6.8

Table A–1 Opcode Groups (Continued)

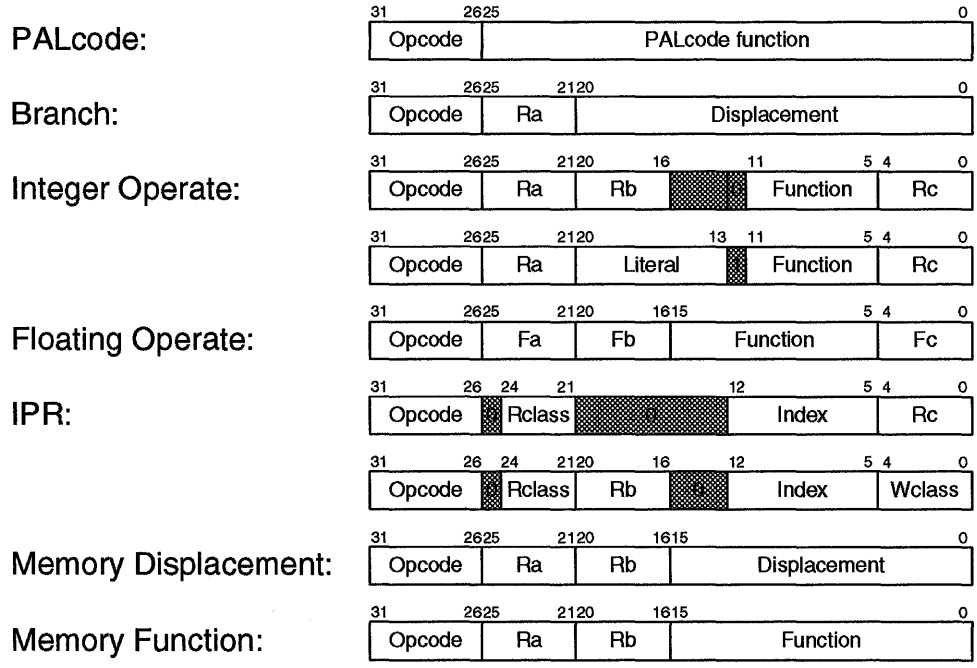
Opcode	Type	Format	In Section
16	IEEE floating-point	Floating Operate	A.6.9
17	Miscellaneous floating-point	Floating Operate	A.6.10
18	Miscellaneous	Memory Function	A.6.11
1A	Jump	Memory Function	A.6.14
1C	Multimedia	Integer Operate	A.6.13
19,1D	HW_MxPR	IPR	17.2
1B,1F	HW_LD/ST	Memory Displacement	17.1
1E	IFETCHB	PALcode	—
20-2F	Load and store	Memory Displacement	A.6.12
30 – 3F	Branch	Branch	A.6.14

A.1 Instruction Format

The Alpha SRM defines several instruction formats and specifies the format for each instruction. The 21464 generally uses the SRM-defined formats with the following exceptions:

- The FTOIx instructions are decoded as Integer Operate rather than the Floating Operate specified in the SRM.
- Instructions listed as Memory format in the SRM are explicitly categorized as either Memory Displacement format or Memory Function format in the 21464.
- Instructions listed as Misc format by the SRM are decoded as Memory Function format by the 21464.

Figure A–1 Instruction Formats



The instruction format defines the location of the operand specifiers and any function code bits. The function code further subdivides the opcode into many separate instructions. The decode tables in this document define a decoding of function code bits that form a non-overlapping map of every possible bit combination. The behavior of every instruction bit pattern is known and consistently decoded throughout the 21464.

The tables use the Alpha SRM mnemonics (in upper-case) to identify instructions. Mnemonics listed in lower-case do not exactly map to a specific Alpha instruction.

A.2 Predecodes

The Ibox does a quick partial decode of instructions defining several buckets useful to the early stages of the pipeline. The predecode logic identifies an instruction as belonging to one of the following 23 groups.

Table A–2 Predecode Logic Groups

Instruction Type	Format	PreDec Type[1]	SrcA	SrcB	Dest	PreDec Bits[2]
CALL_PAL instruction	PALcode	XXP	—	—	P	00010
Floating conditional branch	Branch	FXX	Ra	—	—	01100
Floating-point load operation	Memory	SIF	S	Rb	Fa	11000
Floating-point operation	Floating	FFF	Fa	Fb	Fc	11100
Floating-point store instruction	Memory	FIS	Fa	Rb	S	11101
FTOI instruction	Integer	FXI	Fa	—	Rc	01110
HW_MFPR instruction	IPR	RXI	R	—	Rc	00110
HW_MTPR instruction	IPR	RIW	R	Rb	W	10110
Integer conditional branch	Branch	IXX	Ra	—	—	01000
Integer load operation	Memory	SII	S	Rb	Ra	10000
Integer operation	Integer	III	Ra	Rb	Rc	00100
Integer operation with Rb a literal	Integer	IXI	Ra	—	Rc	00101
Integer store instruction (Not STx_C)	Memory	IIS	Ra	Rb	S	11111
ITOF instruction	Floating	IXF	Ra	—	Fc	00111
LDQ_U instruction	Memory	SUI	S	Rb	Ra	11001
Misc with no A operand	Memory	XII	—	Rb	Ra	11011
Misc with no operands	Memory	XXX	—	—	—	01001
Misc with no result	Memory	IIX	Ra	Rb	—	01111
MT_FPCR instruction	Floating	FFC	Fa	Fb	C	11110
RPCC instruction	Memory	XIY	—	Rb	Ra	01011
Rs / Rc VAX compatibility	Memory	XXN	—	—	Ra	00011
Store conditional	Memory	IIL	Ra	Rb	L	00000
Unconditional branch	Branch	XXI	—	—	Ra	00001

[1] The three-character type identifier defines the type of the A operand, B operand, and result, as follows:

Character	Meaning
C	Floating-Point Control register
F	Floating-point register or result
I	Integer register or result
L	Lock flag value
N	Interrupt flag value
P	PALmode shadow register S1 (CALL_PAL only)
R	IPR reader class specifier
S	Store Set identifier
U	Unaligned address operand
W	IPR writer class specifier
X	No operand or No result
Y	Cycle counter IPR

[2] The Ibox IFU predecode bits EDCBA. See Section 3.8.2.3.1.

For Opcodes 10, 11, 12, 13 and 1C, bit<12> of the instruction defines whether a literal or a register is used for Rb. In the tables, the predecode is listed as I?I for these instructions. They predecode to III if bit<12> is clear (register Rb operand) or IXI if bit<12> is set (literal Rb operand).

Not every instruction is defined exactly as the predecodes suggest. Many instructions identified as III or FFF do not require two input operands (ex. SEXT, SQRT). In most of these cases the SRM requires the unused register to be R31/F31 which results in the exact same treatment as if the extra predecodes had existed. The few exceptions are listed in the format discussion below.

A.3 Instruction Latency

Defines the parent-to-child issue latency. Also identifies any cross-pipeline delay associated with broadcasting the parents results to other pipelines. Instructions that are not pipelined are also identified as "bubbling" for completion. For Example:

n	*N* cycle latency to a child in any pipeline
m+*n*	*M* cycle latency plus extra *n* cycle to other pipelines.
n+B	*N* cycle latency non-pipelined, requires bubble (B) to signal completion.

A.4 Execution Pipelines

Identifies which of the eight pipelines the instruction can execute in. The actual slotting algorithm is a function of the types and positions of the instructions in each map block. Details about instruction slotting can be found at <???>. Just because an instruction is slotted to a particular pipeline does not mean it must execute there, follow-me capabili-

ties in the Qbox allow instructions whose operands are data-ready in another allowed pipeline in the same half of the Queue to issue from that pipeline. Pipelines 0, 2, 5 and 7 are in one half of the Queue, pipes 1, 3, 4, 6 are in the other half.

Format	Meaning
0 – 7	Can execute in any pipe
0 – 3	Can execute in pipes 0, 1, 2, or 3.
0 , 3	Can execute in only pipes 0 or 3
0 ~ 1	Can execute in only pipes 0 or 1 and not both in the same cycle.
Alt 0 – 3	Can execute in pipes 0, 1, 2, or 3, but does not issue to the same pipe in consecutive cycles

A.5 Instruction Info (INST_INFO<15:0>)

To optimize the efficiency of internal queues, the instruction longword is not passed throughout the chip but compressed into two separate fields. The opcode field contains the original 6-bit instruction opcode but the rest of the instruction longword is compressed into a 16-bit inst_info field based on instruction format.

The general rule is:

Instruction Format	Contents of INST_INFO<15:0>
PALcode	OR(inst<25:15>), inst<14:0>
Memory/IPR	inst<15:0>
Otherwise	inst<20:5>

The only exceptions to the general rule follow:

Instruction	INST_INFO<15:0>
RPCC	inst<15:13>, index:0b10111000, inst<4:0>
RS/RC	inst<15:1>, flag

A.6 Specific Opcode and Instruction Type Decoding

A.6.1 Opcode 00, CALL_PAL

The CALL_PAL instruction is only executed in combination with a valid PALcode instruction. For example, a valid combination for OpenVMS is CALL_PAL BPT, with an opcode/function code of 00.0080. Valid PALcode instructions and their function codes are specified in the Alpha SRM according to operating system. The CALL_PAL instruction issues on pipelines 0~1 with a latency of 5.

A.6.2 Opcodes 01 through 07, Reserved

Opcodes 01 through 07 are reserved for the 21464. They predecode to XXX and if executed, return an OPCDEC (or opDec) fault.

A.6.3 Opcode 10, Integer Add/Subtract/Compare

Integer Add/Subtract/Compare instructions.

Table A–3 Opcode 10 Instruction Decoding

Function Code	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
00	x00 0x0x	ADDL	I?I	0 – 7	1 + 1
02	x00 0x1x	S4ADDL	I?I	0 – 7	1 + 1
09	x00 100x	SUBL	I?I	0 – 7	1 + 1
0B	x00 101x	S4SUBL	I?I	0 – 7	1 + 1
0F	xxx 111x	CMPBGE	I?I	0 – 7	1 + 1
12	X01 0xxx	S8ADDL	I?I	0 – 7	1 + 1
1B	X01 10xx	S8SUBL	I?I	0 – 7	1 + 1
1D	001 110x	CMPULT	I?I	0 – 7	1 + 1
20	x10 0x0x	ADDQ	I?I	0 – 7	1 + 1
22	x10 0x1x	S4ADDQ	I?I	0 – 7	1 + 1
29	x10 100x	SUBQ	I?I	0 – 7	1 + 1
2B	x10 101x	S4SUBQ	I?I	0 – 7	1 + 1
2D	0x0 110x	CMPEQ	I?I	0 – 7	1 + 1
32	x11 0xxx	S8ADDQ	I?I	0 – 7	1 + 1
3B	x11 10xx	S8SUBQ	I?I	0 – 7	1 + 1
3D	011 110x	CMPULE	I?I	0 – 7	1 + 1
4D	10x 110x	CMPLT	I?I	0 – 7	1 + 1
6D	11x 110x	CMPLE	I?I	0 – 7	1 + 1

The specific logic functions within the Integer adder are selected as:

Table A–4 Opcode 10 Specific Logic Functions Within the Integer Adder

21464 Decode	Mnemoic	Description
xxx 0xxx	ADD	Add operations
xxx 10xx	SUB	Subtract operations
xxx 11xx	CMP	Compare operations
xx0 xx0x	S0	Ra used unshifted
xx0 xx1x	S4	Ra shifted left two bits before use
xx1 xxxx	S8	Ra shifted left three bits before use
1x0 0X0X	ADDx/V	Enable overflow/under flow exception trapping
1x0 100X	SUBx/V	Enable overflow/under flow exception trapping
x0x xxxx	Long	32-bit inputs/outputs sign extended into 64-bits
x1x xxxx	Quad	64-bit inputs/outputs

A.6.4 Opcode 11, Integer Logical

Integer Logical instructions.

Table A–5 Opcode 11 Instruction Decoding

Function Code	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
00	00x 00xx	AND	I?I	0–7	1+1
08	00x 1xxx	BIC	I?I	0–7	1+1
14	00x 010x	CMOVLBS	I?I	0–7	1+1
16	00x 011x	CMOVLBC	I?I	0–7	1+1
20	01x 00xx	BIS	I?I	0–7	1+1
24	01x 010x	CMOVEQ	I?I	0–7	1+1
26	01x 011x	CMOVNE	I?I	0–7	1+1
28	01x 1xxx	ORNOT	I?I	0–7	1+1
40	10x 00xx	XOR	I?I	0–7	1+1
44	10x 010x	CMOVLT	I?I	0–7	1+1
46	10x 011x	CMOVGE	I?I	0–7	1+1
48	10x 1xxx	EQV	I?I	0–7	1+1
61	11x 00xx	AMASK	I?I	0–7	1+1
64	11x 010x	CMOVLE	I?I	0–7	1+1
66	11x 011x	CMOVGT	I?I	0–7	1+1
68	11x 10xx	CMOV2	I?I	0–7	1+1
6C	11x 11xx	IMPLVER	I?I	0–7	1+1

A.6.5 Opcode 12, Integer Shift

The mskbh, insbh and extbh decodes are not formally defined by the Alpha SRM because all combinations of inputs produce a zero result. The generalized decoding in the 21464 Integer Shifter does not special case these code points and will produce a zero result.

Table A–6 Opcode 12 Instruction Decoding

Function Code	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
02	000 001x	MSKBL	I?I	0–3	1+1
06	x00 0110	EXTBL	I?I	0–3	1+1
0B	x00 1011	INSBL	I?I	0–3	1+1
12	001 001x	MSKWL	I?I	0–3	1+1
16	x01 0110	EXTWL	I?I	0–3	1+1
1B	x01 1011	INSWL	I?I	0–3	1+1

Table A–6 Opcode 12 Instruction Decoding (Continued)

Function Code	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
22	010 001x	MSKLL	I?I	0 – 3	1 + 1
26	x10 0110	EXTLL	I?I	0 – 3	1 + 1
2B	x10 1011	INSLL	I?I	0 – 3	1 + 1
30	xxx 0000	ZAP	I?I	0 – 3	1 + 1
31	xxx 0001	ZAPNOT	I?I	0 – 3	1 + 1
32	011 001x	MSKQL	I?I	0 – 3	1 + 1
34	xxx 010x	SRL	I?I	0 – 3	1 + 1
36	x11 0110	EXTQL	I?I	0 – 3	1 + 1
39	xxx 100x	SLL	I?I	0 – 3	1 + 1
3B	x10 1011	INSQL	I?I	0 – 3	1 + 1
3C	xxx 11xx	SRA	I?I	0 – 3	1 + 1
42	100 001x	Mskbh	I?I	0 – 3	1 + 1
47	x00 0111	Insbh	I?I	0 – 3	1 + 1
4A	x00 1010	Extbh	I?I	0 – 3	1 + 1
52	101 001x	MSKWH	I?I	0 – 3	1 + 1
57	x01 0111	INSWH	I?I	0 – 3	1 + 1
5A	x01 1010	EXTWH	I?I	0 – 3	1 + 1
62	110 001x	MSKLH	I?I	0 – 3	1 + 1
67	x10 0111	INSLH	I?I	0 – 3	1 + 1
6A	x10 1010	EXTLH	I?I	0 – 3	1 + 1
72	111 001x	MSKQH	I?I	0 – 3	1 + 1
77	x11 0111	INSQH	I?I	0 – 3	1 + 1
7A	x11 1010	EXTQH	I?I	0 – 3	1 + 1

A.6.6 Opcode 13, Integer Multiply

Integer Multiply Instructions.

Table A–7 Opcode 13 Instruction Decoding

Function Code	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
00	x0x xxxx	MULL	I?I	4 , 5	5
20	x10 xxxx	MULQ	I?I	4 , 5	5
30	x11 xxxx	UMULH	I?I	4 , 5	5

The specific logic functions within the Integer adder are selected as:

Table A–8 Opcode 13 Specific Logic Functions Within the Integer Adder

21464 Decode	Qualifier	Description
1x0 xxxx	MULL/V	Enable overflow/under flow exception trapping for
110 xxxx	MULQ/V	MULL and MULQ instructions

A.6.7 Opcode 14, ITOFx and Floating-Point Square Root

Integer to Floating register transfer and Floating square root instructions.

For ITOFx instructions, the Ebox format converts Ra and multiplexes the result into the Fbox load datapath.

SQRT instructions only issue on even cycles and are not pipelined.

Table A–9 Opcode 14 Instruction Decoding

Function Code	21464 Decode[1]	Mnemonic	Predecode	Pipelines	Latency
004	xxx xx00 0xxx	ITOFS	IXF	6 , 7	5
014	xxx xx01 0xxx	ITOFF	IXF	6 , 7	5
024	xxx xx1x 0xxx	ITOFT	IXF	6 , 7	5
x0A	ttt rr0x 1xx0	SQRTF	FFF	Alt 0 – 3	18 + B + 1
x0B	ttt rr0x 1xx1	SQRTS	FFF	Alt 0 – 3	18 + B + 1
x2A	ttt rr1x 1xx0	SQRTG	FFF	Alt 0 – 3	33 + B + 1
x2B	ttt rr1x 1xx1	SQRTT	FFF	Alt 0 – 3	33 + B + 1

[1] For SQRT instructions, the ttt and rr fields define the trapping and rounding modes, and all modes are defined for each function code (see below). The 21464 generates an OPCDEC (illegal instruction) trap for any opcode 14 function code that is not defined.

rr	0x	/C	Chopped
	1x	None	Normal (default)
ttt	0x0	None	Imprecise (default)
	0x1	/U	Underflow Enable
	1x0	/S	Exception completion enabled
	1x1	/SU	Underflow & Exception enabled

The FBOX decodes these modes for IEEE instructions (SQRTS, SQTRT) as follows:

rr	00	/C	Chopped
	01	/M	Minus Infinity
	10	None	Normal (default)
	11	/D	Dynamic
ttt	0x0	None	Imprecise (default)
	0x1	/U	Underflow Enable
	10x	/SU	Software completion w/underflow
	11x	/SUI	Software completion w/inexact

A.6.8 Opcode 15, VAX Floating-Point

VAX floating-point instructions.

Table A–10 Opcode 15 Instruction Decoding

Function Code	21464 Decode[1]	Mnemonic	Predecode	Pipelines	Latency
x00	ttt rr0x x000	ADDF	FFF	0 – 3	3 + 1
x01	ttt rr0x x001	SUBF	FFF	0 – 3	3 + 1
x02	ttt rr0x x010	MULF	FFF	0 – 3	3 + 1
x03	ttt rr0x x011	DIVF	FFF	Alt 0 – 3	9 + B + 1
x1E	ttt rr0x 11xx	CVTDG	FFF	0 – 3	3 + 1
x20	ttt rr1x x000	ADDG	FFF	0 – 3	3 + 1
x21	ttt rr1x x001	SUBG	FFF	0 – 3	3 + 1
x22	ttt rr1x x010	MULG	FFF	0 – 3	3 + 1
x23	ttt rr1x x011	DIVG	FFF	Alt 0 – 3	13 + B + 1
x25	ttt xxxx 010x	CMPGEQ	FFF	0 – 3	3 + 1
x26	ttt xxxx 0110	CMPGLT	FFF	0 – 3	3 + 1
x27	ttt xxxx 0111	CMPGLE	FFF	0 – 3	3 + 1
x2C	ttt rr10 1100	CVTGF	FFF	0 – 3	3 + 1
x2D	ttt rr10 1101	CVTGD	FFF	0 – 3	3 + 1
x2F	ttt rr10 111x	CVTGQ	FFF	0 – 3	3 + 1
x3C	xxx rr11 110x	CVTQF	FFF	0 – 3	3 + 1
x3E	xxx rr11 111x	CVTQG	FFF	0 – 3	3 + 1

[1] The ttt and rr fields define the trapping and rounding modes. The Fbox decodes these modes for VAX instructions as shown below.

rr	0x	/C	Chopped
	1x	None	Normal (default)
ttt	0x0	None	Imprecise (default)
	0x1	/U	Underflow Enable
	1x0	/S	Exception completion enabled
	1x1	/SU	Underflow & Exception enabled

The CMPxxx instructions only define one trap option. The txx mode is decoded as follows:

txx	0xx	None	Imprecise (default)
	1xx	/S	For CMPxxx

A.6.9 Opcode 16, IEEE Floating-Point

IEEE floating point instructions.

Table A–11 Opcode 16 Instruction Decoding

Function Code	21464 Decode[1]	Mnemonic	Predecode	Pipelines	Latency
x00	ttt rr0x x000	ADDS	FFF	0 – 3	3 + 1
x01	ttt rr0x x001	SUBS	FFF	0 – 3	3 + 1
x02	ttt rr0x x010	MULS	FFF	0 – 3	3 + 1
x03	ttt rr0x x011	DIVS	FFF	Alt 0 – 3	9 + B + 1
x20	ttt rr1x x000	ADDT	FFF	0 – 3	3 + 1
x21	ttt rr1x x001	SUBT	FFF	0 – 3	3 + 1
x22	ttt rr1x x010	MULT	FFF	0 – 3	3 + 1
x23	ttt rr1x x011	DIVT	FFF	Alt 0 – 3	13 + B + 1
x24	txx xxxx 0100	CMPTUN	FFF	0 – 3	3 + 1
x25	txx xxxx 0101	CMPTEQ	FFF	0 – 3	3 + 1
x26	txx xxxx 0110	CMPTLT	FFF	0 – 3	3 + 1
x27	txx xxxx 0111	CMPTLE	FFF	0 – 3	3 + 1
x2C	ttt rrx0 110x	CVTTS	FFF	0 – 3	3 + 1
2AC	t10 xxx0 110x	CVTST	FFF	0 – 3	3 + 1
x2F	ttt rrx0 111x	CVTTQ	FFF	0 – 3	3 + 1
x3C	txx rrx1 110x	CVTQS	FFF	0 – 3	3 + 1
X3E	txx rrx1 111x	CVTQT	FFF	0 – 3	3 + 1

[1] The ttt and rr fields define the trapping and rounding modes. The Fbox decodes these modes for VAX instructions as shown below:

rr	00	/C	Chopped
	01	/M	Minus Infinity
	10	None	Normal (default)
	11	/D	Dynamic
ttt	0x0	None	Imprecise (default)
	0x1	/U	Underflow Enable
	10x	/SU	Software completion w/underflow
	11x	/SUI	Software completion w/inexact

The CMPxxx and CVTQx instructions only define one trap option. The txx mode is decoded as follows:

txx	0xx	None	Imprecise (default)
	1xx	/SU	For CMPxxx, /SUI For CVTQx

Unlike any other floating-point instruction, decoding of the trap mode bits differentiates CVTST and CVTTS instructions. The special decoding is:

ttt	000	CVTTS	None	Imprecise (default)
	0x1	CVTTS	/U	Underflow enable
	10x	CVTTS	/SU	Software w/Underflow
	111	CVTTS	/SUI	Software w/Inexact
t10	010	CVTST	None	Imprecise (default)
	110	CVTST	/S	Software denormal fixup

A.6.10 Opcode 17, Miscellaneous Floating-Point

For FCMOVxx instructions, the Ibox will scan for a FCPYS->R31 instruction immediately following the FCMOVxx instruction and if found replace it with a FCMOV2 instruction. If the instruction following a FCMOVxx is not a FCPYS->R31, the Ibox tags the FCMOVxx instruction as legacy. Legacy FCMOVxx instructions terminate a map-block and are repeated in the next map-block. The Pbox then converts the repeated FCMOVxx instruction (which is always in position 0 of the new map-block) to the FCMOV2.

The Ebox detects user-mode MT_FPCR instructions and traps to PALmode to fix-up.

Table A–12 Opcode 17 Instruction Decoding

Function Code	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
x10	xxx xx01 xxxx	CVTLQ	FFF	0 – 3	3 + 1
x20	xxx xxx0 0000	CPYS	FFF	0 – 3	1 + 1
x21	xxx xxx0 0001	CPYSN	FFF	0 – 3	1 + 1
x22	xxx xxx0 001x	CPYSE	FFF	0 – 3	1 + 1
x24	xxx xxx0 01x0	MT_FPCR	FFC	0,3	-
x25	xxx xxx0 01x1	MF_FPCR	FFF	0,3	3 + 1
x68	xxx xxx0 100x	FCMOV2	FFF	0 – 3	1 + 1
x2A	xxx xxx0 1010	FCMOVEQ	FFF	0 – 3	1 + 1
x2B	xxx xxx0 1011	FCMOVNE	FFF	0 – 3	1 + 1
x2C	xxx xxx0 1100	FCMOVLT	FFF	0 – 3	1 + 1
x2D	xxx xxx0 1101	FCMOVGE	FFF	0 – 3	1 + 1
x2E	xxx xxx0 1110	FCMOVLE	FFF	0 – 3	1 + 1
X2F	xxx xxx0 1111	FCMOVGT	FFF	0 – 3	1 + 1
X30	ttt[1] xx11 xxxx	CVTQL	FFF	0 – 3	3

[1] Only CVTQL has a defined trap mode, as shown below:

ttt	0x0	None	Imprecise (default)
	0x1	/U	Underflow Enable
	1xx	/SU	Software completion w/underflow

A.6.11 Opcode 18, Miscellaneous

TRAPB, EXCB and FETCHx instructions never actually issue from the Qbox but are completed immediately and therefore act as NOPs. MBs also never formally issue from the Qbox but are instead sent to the Mbox as soon as they enter the Qbox. MB instructions do not complete until the Mbox notifies the Qbox that the necessary conditions have been met.

The WH64EN instruction is currently proposed as ECO#127 to the Alpha SRM.

Table A–13 Opcode 18 Instruction Decoding

Function Code	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
0000	00xx x0xx xxxx xxxx	TRAPB	XXX	-	-
0400	00xx x1xx xxxx xxxx	EXCB	XXX	-	-
4000	01xx 00xx xxxx xxxx	MB	XXX	-	-
4400	01xx 01xx xxxx xxxx	WMB	IIX	4 , 5	-
4800	01xx 1xxx xxxx xxxx	(MB)	XXX	-	-
8000	100x xxxx xxxx xxxx	FETCH	XXX	-	-
A000	1010 xxxx xxxx xxxx	FETCH_M	XXX	-	-
B000	1011 00xx xxxx xxxx	LDL_ARM	SII	6 , 7	3
B400	1011 01xx xxxx xxxx	LDQ_ARM	SII	6 , 7	3
B800	1011 1xxx xxxx xxxx	QUIESCE	IIX	4 , 5	-
C000	110x xxxx xxxx xxxx	RPCC	XIY	0~1	5
E000	1110 0xxx xxxx xxxx	RC	XXN	4, 5	1 + 1
E800	1110 10xx xxxx xxxx	ECB	IIX	4 , 5	-
EC00	1110 11xx xxxx xxxx	CCB	IIX	4 , 5	-
F000	1111 0xxx xxxx xxxx	RS	XXN	4, 5	1 + 1
F800	1111 10xx xxxx xxxx	WH64	IIX	4, 5	-
FC00	1111 11xx xxxx xxxx	WH64EN	IIX	4, 5	-

A.6.12 Load and Store Instructions

Load and store instructions.

Table A–14 Load and Store Instruction Decoding

Opcode	Mnemonic	Predecode	Pipelines	Latency
08	LDA	XII	0 – 7	1 + 1
09	LDAH	XII	0 – 7	1 + 1
0A	LDBU	SII	6 , 7	3
0B	LDQ_U	SUI	6 , 7	3
0C	LDWU	SII	6 , 7	3

Table A–14 Load and Store Instruction Decoding (Continued)

Opcode	Mnemonic	Predecode	Pipelines	Latency
0D	STW	IIS	4, 5	3[2]
0E	STB	IIS	4, 5	3[2]
0F	STQ_U	IIS	4, 5	3[2]
20	LDF	SIF	6 , 7	5
21	LDG	SIF	6 , 7	5
22	LDS	SIF	6 , 7	5
23	LDT	SIF	6 , 7	5
24	STF	FIS	4, 5	3[2]
25	STG	FIS	4, 5	3[2]
26	STS	FIS	4 , 5	3[2]
27	STT	FIS	4 , 5	3[2]
28	LDL	SII	6 , 7	3
29	LDQ	SII	6 , 7	3
2A	LDL_L	SII	6 , 7	3
2B	LDQ_L	SII	6 , 7	3
2C	STL	IIS	4 , 5	3[2]
2D	STQ	IIS	4 , 5	3[2]
2E	STL_C	IIL	4 , 5[1]	3
2F	STQ_C	IIL	4 , 5[1]	3

[1] Store Conditional instructions issue as stores to pipelines 4 and 5 but bubble back completion to the QBOX, and the final completion of the STx_C instruction appears on the load pipes 6 and 7.

[2] Although store instructions do not produce a register result and therefore do not have normal dependents, the IBOX store-set logic can create dependency groups of loads and stores. A load that is store-set dependent on a store instruction will have an effective issue latency of three cycles from the issue of the store.

A.6.13 Opcode 1C, Integer Multimedia

Integer multimedia instructions.

Table A–15 Opcode 1C Instruction Decoding

Function Code	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
00	000 00x0	SEXTB	I?I	0 – 7	1 + 1
01	000 00x1	SEXTW	I?I	0 – 7	1 + 1
04	000 01x0	CMPWGE	I?I	2 , 3	5
05	000 01x1	CMPLGE	I?I	2 , 3	5
08	000 1xx0	PERMB8	I?I	0 , 1	5

Table A–15 Opcode 1C Instruction Decoding (Continued)

Function Code	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
09	000 1xx1	GPKBLB4	I?I	0,1	5
10	001 00zz	VADDzzz	I?I	2,3	5
14	001 010x	VADDUL2	I?I	2,3	5
16	001 011x	VADDSL2	I?I	2,3	5
18	001 10zz	VSUBzzz	I?I	2,3	5
1C	001 110x	VSUBUL2	I?I	2,3	5
1E	001 111x	VSUBSL2	I?I	2,3	5
20	010 00zz	VMINMAXzzz	I?I	2,3	5
24	010 010x	VMINMAXUL2	I?I	2,3	5
26	010 011x	VMINMAXSL2	I?I	2,3	5
28	010 1000	PKUWB8	I?I	0,1	5
29	010 1001	PKULW4	I?I	0,1	5
2A	010 1010	PKSWB8	I?I	0,1	5
2B	010 1011	PKSLW4	I?I	0,1	5
2C	010 1100	UPKUBW4	I?I	0,1	5
2D	010 1101	UPKUWL2	I?I	0,1	5
2E	010 1110	UPKSBW4	I?I	0,1	5
2F	010 1111	UPKSWL2	I?I	0,1	5
30	011 0000	CTPOP	I?I	2,3	5
31	011 0001	PERR	I?I	2,3	5
32	011 0010	CTLZ	I?I	2,3	5
33	011 0011	CTTZ	I?I	2,3	5
34	011 0100	UNPKBW	I?I	0,1	5
35	011 0101	UNPKBL	I?I	0,1	5
36	011 0110	PKWB	I?I	0,1	5
37	011 0111	PKLB	I?I	0,1	5
38	011 1000	MINSB8	I?I	2,3	5
39	011 1001	MINSW4	I?I	2,3	5
3A	011 1010	MINUB8	I?I	2,3	5
3B	011 1011	MINUW4	I?I	2,3	5
3C	011 11zz	MAXzzz	I?I	2,3	5
40	100 00zz	TADDzzz	I?I	2,3	5
44	100 01zz	TSUBzzz	I?I	2,3	5
48	100 10zz	TABSERRzzz	I?I	2,3	5

Table A–15 Opcode 1C Instruction Decoding (Continued)

Function Code	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
4C	100 11zz	TSQERRzzz	I?I	2,3	5
50	101 00zz	TMULzzz	I?I	2,3	5
54	101 01x0	TMULUSB8	I?I	2,3	5
55	101 01x1	TMULUSW4	I?I	2,3	5
59	101 1x0x	VMULLUW4	I?I	2,3	5
5b	101 1x1x	VMULHUW4	I?I	2,3	5
60	110 0000	VSRB8	I?I	0,1	5
61	110 0001	VSRW4	I?I	0,1	5
62	110 0010	VSRAB8	I?I	0,1	5
63	110 0011	VSRAW4	I?I	0,1	5
64	110 010x	VSRL2	I?I	0,1	5
66	110 011x	VSRAL2	I?I	0,1	5
68	110 10x0	VSLB8	I?I	0,1	5
69	110 10x1	VSLW4	I?I	0,1	5
6C	110 11xx	VSLL2	I?I	0,1	5
70	111 0xxx	FTOIT	FXI	4,5	3
78	111 1xxx	FTOIS	FXI	4,5	3

A.6.14 Branch and Jump Instructions

Branch and Jump instructions.

Table A–16 Branch and Jump Instruction Decoding

Opcode	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
1A.0	00xx xxxx xx...	JMP	XII	0~1	5
1A.1	01xx xxxx xx...	JSR	XII	0~1	5
1A.2	10xx xxxx xx...	RET	XII	0~1	5
1A.3	11xx xxxx xx...	JSR_CO	XII	0~1	5
30		BR	XXI	0~1	5
31		FBEQ	FXX	0 – 1	-
32		FBLT	FXX	0 – 1	-
33		FBLE	FXX	0 – 1	-
34		BSR	XXI	0~1	5
35		FBNE	FXX	0 – 1	-
36		FBGE	FXX	0 – 1	-

Table A–16 Branch and Jump Instruction Decoding (Continued)

Opcode	21464 Decode	Mnemonic	Predecode	Pipelines	Latency
37		FBGT	FXX	0 – 1	-
38		BLBC	IXX	0 – 7	-
39		BEQ	IXX	0 – 7	-
3A		BLT	IXX	0 – 7	-
3B		BLE	IXX	0 – 7	-
3C		BLBS	IXX	0 – 7	-
3D		BNE	IXX	0 – 7	-
3E		BGE	IXX	0 – 7	-
3F		BGT	IXX	0 – 7	-

A.6.15 PALcode Instructions

The MSB of the index field of the HW_MTPR instruction indicates the destination:

0 = Mbox

1 = Ibox

Table A–17 PALcode Instruction Decoding

Opcode	21464 Decode[1]	Mnemonic	Predecode	Pipelines	Latency
0x19	xxx0 iiii iiix xxxx	HW_MFPR	RXI	4~5	5
	xxx1 iiii iiix xxxx	HW_MFPR	RX1	0~1	5
0x1B	ttts sxxx xxxx xxxx	HW_LD	SII	6 , 7	3
0x1D	xxx0 iiii iiiw wwww	HW_MTPR	RIW	6 , 7	1 / 3[2]
	xxx1 iiii iiiw wwww	HW_MTPR	RIW	0~1	1[2]
0x1E	xxxx xxxx xxxx xxxx	IFETCHB	XXX	4 , 5	—
0x1F	ttts sxxx xxxx xxxx	HW_ST	IIX	4 , 5	—

[1] The decode bit symbols i, r, s, and t are defined as follows:

For HW_LD and HW_ST:

ttt Type of memory reference to perform. See the *Type* field in Table 17–1.

ss Size of the data transaction. See the *Length* field in Table 17–1.

For HW_MFPR and HW_MTPR:

iiii iiii Identifier of the IPR to read. See the Index field in Tables 17–2 and 17–3 and Table 16–1.

rrrr Reader class of the instruction. See the *Rclass* field in Tables 17–2 and 17–3.

w wwww Writer class of the instruction. See the *Wclass* field in Table 17–3.

[2] HW_MTPR instructions can specify a writer class to create an issue dependency to future HW_MxPR instructions. HW_MxPR instructions that identify a reader class dependency are scheduled to issue no earlier than 1 cycle after the HW_MTPR instruction that wrote the class dependency. HW_MTPR instructions can also specify writer class dependencies that are satisfied on completion rather than issue. HW_MxPR instructions that identify a reader class dependency against this type of writer class are scheduled to issue no earlier than 3 cycles after the issue of the completion bubble signal for the writer. The 21464 only allows specifying completion dependencies for Mbox HW_MTPR instructions; the completion bit is ignored for Ibox destinations.

B

LDx_ARM/QUIESCE Instruction Characteristics

The 21464 supports simultaneous multithreading (SMT), where up to 4 threads (or processes) share the resources of the CPU. On an SMT CPU, a spin-lock loop wastes CPU resources that could be used by other processes or threads that are executing. We are proposing two new instructions for the Alpha architecture, LDx_ARM and QUIESCE, which will permit a thread to wait on a memory location without actively spinning.

B.1 Relationship Between SMT and LDx_ARM/QUIESCE

The 21464 is implementing Simultaneous Multithreading because of the boost in throughput it provides when running independent programs, and because of the expected performance improvement for decomposed application. For independent programs executing simultaneously, performance studies show roughly 100% increase in throughput, compared with running the programs only one at a time. June 1998 results showed:

Programs	1-threaded IPC[1] harmonic mean	4-threaded IPC	4T/1T IPC increase
Compress, Gcc, M88ksim, Go (int)	2.38	5.15	2.16x
Tomcat, Applu, Swim, Povray (float)	3.50	6.12	1.75x
SQL traces (database)	1.33	3.40	2.55x

[1] IPC=Instructions Per Cycle

We anticipate that SMT will also be very useful for decomposed applications, where one program is broken into multiple threads and locking protocols are used by each thread to control access to shared data. Preliminary results show speedups from 1.1x to 2.5x (Cilk results from CRL). These results were produced with code that used QUIESCE; other runs done without QUIESCE showed no speedup at all, or even degradation. The reason for the poor performance without QUIESCE is due to the way threads wait for access to a lock, as explained in the following paragraphs.

An integral part of many locking protocols is a busy wait loop, often referred to as a spin lock. In a spin lock, a process loops looking at a particular memory location and waiting for it to change to a specific value before proceeding. Once the value has changed, the process is then free to attempt an atomic update of the location, thus obtaining the lock.

In a conventional multiprocessor the CPU resources and memory bandwidth consumed by a task in a spin lock are not simultaneously shared with any other tasks. Thus, while the task is spinning there is no resource contention within the CPU, and no reason not to let the task spin as much as it wants. Studies have shown that approximately 15% of processor time is spent in spin loops.

In a simultaneous multithreaded CPU, however, the resources consumed by the spinning task are being denied to the other threads that are doing useful work. Thus, it is desirable to prevent the task in the spin lock from consuming resources when there is no chance that it will find the value it is looking for. We refer to the action of pausing execution of a thread (until the condition it is waiting for might be satisfied) as quiescing the thread. In a simultaneous multithreaded machine, the act of quiescing would mean that no instructions are executed from the quiesced "thread processing unit" or TPU. The other TPUs continue normally.

We propose to add two new instructions to the Alpha architecture, LDx_ARM (long or quad) and QUIESCE. Software uses these in sequence, LDx_ARM and then QUIESCE. These instructions allow a processor to declare that it has no work to do until some other processor writes a specified location in memory space. Two internal processor registers are also involved, watch_physical_address and watch_flag. In addition, the processor has a counter to signal the timeout of the quiesce period at intervals.

For backwards compatibility, the LDx_ARM/QUIESCE sequence must be conditionalized with the AMASK instruction, so that non-SMT processors do not execute these new instructions. We also propose defining a new AMASK bit to identify a processor with SMT capability.

Table B–1 SMT AMASK Instruction Bit

Bit	Meaning
10	Support for Simultaneous Multithreading. This processor is part of an implementation where multiple threads, or processes, are executed simultanously within a single CPU.

B.2 Goals for the LDx_ARM and QUIESCE Instruction Definition

We would like to achieve the following goals in our definition of these instructions:

- Define the instructions such that in-order execution of LDx_ARM and QUIESCE is ensured. This is accomplished through the defined dependency on watch_flag - LDx_ARM sets it and QUIESCE uses it as a condition on its operation.
- Eliminate possibility of a race between the lock just becoming available, and quiescing the machine. This is accomplished by having the LDx_ARM load the lock value so that code can test the lock before executing the QUIESCE.
- May be used either in PALmode or in normal mode.
- Have code using these instructions still be functional if executed in older machines.

B.2.1 Specific LDx_ARM Instruction Characteristics

The following sections contain the specific characteristics and requirements that define the LDL_ARM and LDQ_ARM instructions.

B.2.1.1 Instruction Description

The mnemonics/description for the LDx_ARM instructions are:

Mnemonic	Description
LDL_ARM	Load sign-extended longword from memory to register and arm
LDQ_ARM	Load quadword from memory to register and arm

The LDx_ARM instructions are described in a manner that is as similar as possible to the LDx_L instructions, except the LDx_L instructions affect lock_flag and locked_physical_address, while the LDx_ARM instructions affect watch_flag and watch_physical_address. Note however, that LDx_ARM, because they use the Memory/function code instruction format, have no displacement.

Instruction Format:

```
LDx_ARM         Ra, (Rb.ab)             ! Mfc format
```

Operation:

```
va <- Rbv
CASE
  big_endian_data: va' <- va XOR 000(base 2)     !LDQ_ARM
  big_endian_data: va' <- va XOR 100(base 2)     !LDL_ARM
  little_endian_data: va' <- va                  !LDL_ARM
 ENDCASE
 watch_flag <- 1
 watch_physical_address <- PHYSICAL_ADDRESS(va)
 Ra <- SEXT((va')<31:0>)                         !LDL_ARM
 Ra <- (va')<63:0>                               !LDQ_ARM
```

Exceptions:

Access Violation

Alignment

Fault on Read

Translation Not Valid

Qualifiers:

None

LDx_ARM is used in conjunction with QUIESCE to idle a process while waiting for a shared resource (rather than looping and continually testing the lock bit).

The virtual address is in register Rb. For a big-endian longword access, va<2> (bit 2 of the virtual address) is inverted, and any memory management fault is reported for va (not va'). The source operand is fetched from memory, sign-extended for LDL_L, and written to register Ra. If the LDx_ARM instruction encounters an exception, it is treated just as for a LDQ instruction.

When a LDx_ARM instruction is executed without faulting, the processor records the target physical address in a per-processor watch_physical_address register and sets the per-processor watch_flag.

If the per-processor watch_flag is (still) set when a QUIESCE instruction is executed, the processor quiesces, as described for the QUIESCE instruction.

Processor A causes the clearing of a set watch_flag in Processor B by doing any of the following in B's watched range of physical addresses: a successful store, a successful store_conditional, or executing a WH64 instruction that modifies data on processor B. A processor's watched range is the aligned block of 2**N bytes that includes the watch_physical_address. The 2**N value is implementation dependent, and must match the lock range implemented for LDx_L and STx_C. It is at least 16 (minimum lock range is an aligned 16-byte block) and is at most the page size for that implementation (maximum lock range is one physical page).

A processor's watch_flag is cleared if that processor's implementation-specific quiesce timeout counter expires, as described for the QUIESCE instruction. A processor's watch_flag is also cleared if that processor encounters a CALL_PAL REI, CALL_PAL rti, or CALL_PAL rfe instruction. A processor's watch_flag is cleared if that processor encounters a CALL_PAL retsys (return from syscall) or CALL_PAL urti (return from user mode trap). It is UNPREDICTABLE whether or not a processor watch_flag is cleared on any other CALL_PAL instruction. It is UNPREDICTABLE whether a processor's watch_flag is cleared by that processor executing a WH64 or ECB instruction. The watch_flag may also be cleared for implementation-specific reasons.

It is UNPREDICTABLE whether the watch_flag is cleared by an interrupt to the processor, whether or not the processor is in PALmode.

The following sequence:

```
LDQ_ARM
<branch to GetLock if lock available>
QUIESCE
GetLock:
```

when executed on a given processor, will quiesce the processor if the branch to GetLock is not taken, and will continue execution if the branch is taken.

Notes

- The conditions which clear watch_flag are intended to cover the cases when a change of control, occurring between a LDx_ARM and a QUIESCE, may have executed a LDx_ARM and changed the watch_physical_address value. We don't want to quiesce on the wrong value of watch_physical_address.
- Executing a LDx_ARM on one processor does not affect any architecturally visible state on another processor, and in particular cannot clear watch_flag on another processor, causing the other processor to come out of a quiescent state. Note: Without this restriction, two processors executing LDQ_ARM/QUIESCE sequences could be continually re-arming each other.

LDx_ARM and QUIESCE instructions need not be paired. In particular, a LDx_ARM may be followed by a conditional branch: on the fall-through path a QUIESCE is executed, whereas on the taken path no matching QUIESCE is executed.

If two LDx_ARM instructions execute with no intervening QUIESCE, the second one overwrites the state of the first one. If two QUIESCE instructions execute with no intervening LDx_ARM, the second one never quiesces the processor because watch_flag was clear after execution of the first, whether it quiesced the processor or not.

- Software will not emulate any LDx_ARM instruction.
- If the physical address of the LDx_ARM and the physical address intended to be watched are not within the same naturally aligned 16-byte sections of physical memory, the processor may continue to be quiesced despite another processor's store to the watched range; hence, care should be taken to specify the addresses with correct alignment.
- If any other memory access (ECB, LDx, LDDQ_U, STx, STQ_U, WH64) is executed on the given processor between the LDx_ARM and the QUIESCE, the QUIESCE may always fail on some implementations.

Note: Otherwise, a direct-mapped TB could thrash. Or, the memory reference could change the contents of the cache which the implementation might depend upon. It should be possible to always code very few instructions between the LDx_ARM and the QUIESCE.

- If a branch is taken between the LDx_ARM and the QUIESCE, the sequence above may always fail (processor will not quiesce) on some implementations. (CMOVxx may be used to avoid branching.)
- If a subsetted instruction (for example, floating-point) is executed between the LDx_ARM and the QUIESCE, the QUIESCE may always fail on some implementations because of the Illegal Instruction Trap.
- If an instruction with an unused function code is executed between the LDx_ARM and the QUIESCE, the QUIESCE may always fail on some implementations because an instruction with an unused function code is UNPREDICTABLE.
- If a large number of instructions are executed between the LDx_ARM and the QUIESCE, the QUIESCE may always fail on some implementations because of a timer interrupt always clearing the watch_flag before the sequence completes.
- Execution of a WH64 instruction on processor A to a region within the watched range of processor B, where the execution of the WH64 changes the contents of memory, causes the watch_flag on processor B to be cleared. If the WH64 does not change the contents of memory of processor B, it need not clear the watch_flag.

Implementation Notes

- When not in PALmode, the signalling of an interrupt should clear the watch flag, so that QUIESCE cannot be used as a way to delay interrupt processing.

- The watch_flag and watch_physical_address register must be loaded simultaneously with reading the value of the lock. Hardware must ensure that even if the lock value becomes unlocked immediately after reading it, and before the QUIESCE is executed, watch_flag will be cleared and will prevent the processor from quiescing (the QUIESCE will fail, as should happen). Note: in some sense, this is a performance issue, not a functional issue: if the watch_flag is not cleared due to the change in the lock, QUIESCE_TIMER will eventually time out and end the quiesce period.
- Since watch_flag and watch_physical_address are implicitly written by LDx_ARM and implicitly read by QUIESCE, implementations must ensure that any speculative execution of those instructions preserves the read-order and write-order of watch_flag and watch_physical_address, as intended in the original program.

  For example, in the code below, if the first branch is incorrectly predicted taken, the second LDx_ARM must not be allowed to affect the behavior of the first QUIESCE by changing watch_physical_address.

```
    LDQ_ARM R1, (R5)
    BEQ R1, test
    QUIESCE
test:
    LDQ_ARM R1, (R5)
    BEQ R1, xxx
    QUIESCE
```

B.2.2 Specific QUIESCE Instruction Characteristics

The following sections contain the specific characteristics and requirements that define the QUIESCE instruction.

The mnemonic/description for the QUIESCE instruction are as follows:

Mnemonic	Description
QUIESCE	Quiesce Conditional

Format

```
QUIESCE                 ! Mfc format
```

Operation:

```
IF (watch_flag != 0)  THEN
    start QUIESCE_TIMER
    suspend program execution
    resume execution when watch_flag==0
```

Exceptions:

NONE

Qualifiers:

None

QUIESCE checks the watch_flag to see if it is set; if it is, the processor starts the implementation-specific QUIESCE_TIMER and pauses execution of this instruction stream. When the watch_flag is cleared, execution begins again. It is implementation-specific exactly how/if the machine pauses execution and when exactly it resumes. If the watch_flag is set, the QUIESCE instruction is considered complete at the beginning of the quiescent period.

The implementation-specific QUIESCE_TIMER starts counting when the QUIESCE determines that the processor is going to quiesce. After some implementation-specific finite period of time, QUIESCE_TIMER expires and clears the watch_flag. If the quiesce period ends before the QUIESCE_TIMER expires, the QUIESCE_TIMER must be stopped, to prevent it clearing watch_flag after a future LDx_ARM.

After the quiescent period, execution resumes at the instruction following the QUIESCE, or, if the QUIESCE was terminated because watch_flag was cleared by an interrupt, execution may resume at an interrupt servicing routine.

By definition, the watch_flag is clear at the end of the quiescent period, since the quiescent period cannot end until watch_flag is clear.

Implementation notes

- If an interrupt causes a processor to end a quiescent period and immediately start executing the interrupt servicing routine, that ISR may return to the QUIESCE instruction only if watch_flag is guaranteed to be clear. If it is not, the ISR must return to the instruction after the QUIESCE, since the value of watch_physical_address may have been changed by a LDx_ARM executed while servicing the interrupt.
- If an interrupt occurs during a quiescent period, an implementation does not have to start the ISR immediately after the QUIESCE; it may choose to delay execution of the ISR until some later point in the instruction stream.
- An implementation may allow software to specify the QUIESCE timeout period through an IPR. A timeout value of 0 would effectively disable the QUIESCE instruction.
- The implementation-specific maximum timeout value should not exceed *n* microseconds (where *n* is TBD).

Software/Hardware Note

The quiesce timeout counter is useful/necessary for the following reasons:

- The timeout enables the implementation of a backoff algorithm, where a process can deschedule itself after some period of time if it hasn't gotten the lock.
- The quiesce timeout counter prevents a processor from deadlocking if there is a coding error.
- Suppose the code updating the memory location takes an access violation and never gets to unlock the lock. The quiesce timeout allows the waiting processor to wake up and discover the problem with checking code.

Software Note

If a longer quiesce period is desired than that provided by a given implementation, software can accomplish that by looping and quiescing repeatedly.

B.2.2.1 Data Sharing Using LDx_ARM/Quiesce

Efficient Data Sharing in a Simultaneous Multithreaded Processor

In a simultaneous multithreaded (SMT) CPU, multiple threads, or processes, are executed simultaneously while sharing the resources of a single CPU. On an SMT CPU, a spin-lock loop wastes CPU resources that could be used by other processes or threads that are executing. The LDx_ARM and QUIESCE instructions are used in a simultaneous multithreaded CPU to keep a thread from consuming resources while it waits for a lock.

An example code sequence using the quiesce operation follows. In this program, R5 contains the address of a lock. The program is spinlocking on the lock until it is 0. R0 is loaded with the value of the lock.

```
GetLock:
        LDQ_L   R0,(R5)             ;load the lock value
        BNEQ    R0, HandleBusyLock  ;if not available, quiesce
        <modify R0>
        STQ_C   R0,(R5)             ;store new lock value if lock_flag still set
        BEQ     R0, GetLock         ;if store conditional failed, try again
        <critical section>    ;we have the lock, now do the real work
        <clear lock>          ;done
        RET


HandleBusyLock:
        LDA R2, 0x400(R31)      ;set bit 10, SMT bit in AMASK
        AMASK R2, R2            ;test whether SMT processor
        BEQ R2, CheckLock       ;if no SMT, skip quiesce
        LDQ_ARM R0,(R5)         ;load the lock value at address R5 into R0
                                ;put lock address into watch_physical_address
                                ;set watch_flag
        BEQ     R0, GetLock     ;if lock available, try to get it
        QUIESCE                 ;if watch_flag set, go quiet
CheckLock:
        LDQ     R0,(R5)         ;load lock value again
        BEQ     R0, GetLock     ;if available, try for it again
        <check for spinning on lock too long>
        BR HandleBusyLock       ;loop again
```

In that code sequence, testing the lock just after the LDQ_ARM is crucial to performance in the case where the lock is available - otherwise the code would quiesce for no reason. Having the QUIESCE fall through into the CheckLock section allows us to check the lock again, in case the QUIESCE ended for some other reason than a change in the lock value. Note however that for a lock which is highly contended, the lines "BEQ R0, GetLock" will mispredict when the lock is finally given up, assuming that we issued QUIESCE multiple times before getting a chance at the lock. This mispredict will slow down the attempt to get the lock.

Note also that if we execute the LDQ_ARM and we don't QUIESCE, because we branch away to get the lock, the watch_flag will still be set. It will continue to be set until it is cleared by one of the conditions given for clearing watch_flag. This should have no actual effect on machine hardware since it won't be quiesced at the time. The fact that watch_flag is set when a QUIESCE is not actually watching for anything is harmless - the next LDx_ARM which executes will load a new watch_physical_address and set watch_flag whether or not it is already set.

B.3 Proposed Opcode Assignments

We propose the following opcode assignments

Table B–2 Proposed LDx_ARM/QUIESCE Opcode Assignments

Mnemonic	Instruction Type	Opcode
LDL_ARM	Mfc	18.B000
LDQ_ARM	Mfc	18.B400
QUIESCE	Mfc	18.B800

Ideally, we would choose opcodes with the following characteristics:

- They are memory format instructions, for ease of implementation.
- They are NOPs to all pre-EV8 Alpha processors.
- LDx_ARM has a displacement field.

If we found opcodes meeting these criteria, QUIESCE code could be written without using AMASK to condition the code based on the processor type. (If code depended on the register value loaded by the LDx_ARM, an ordinary load would be needed before the LDx_ARM, to accomplish the load operation in older machines.) We do not believe it is possible to find opcodes that look like NOPs to all older implementations, and also fit our other criteria. For example, we could use opcodes in the 11.xx (integer logical) category, such as were used for AMASK; however, these are operate format, not memory format.

We conclude that QUIESCE code sequences will have to be conditioned with AMASK. Since this is the case, we'd like to choose opcodes with the following characteristics:

- They are memory format instructions, for ease of implementation.
- LDx_ARM instructions are illegal operations to all pre-EV8 Alpha processors.
- LDx_ARM has a displacement field.

The Miscellaneous category of opcodes (18.xxxx) provides memory format instructions. But, this category has no displacement field, since that field is used as the function field. This gives LDx_ARM a dissimilarity from LDx_L: LDx_ARM cannot have a displacement when specifying the load address.

Note: Matt Reilly tried all the 18.xxxx opcodes and found that most of the unused ones trap (illegal instruction trap) on the 21164, but are NOPs on the 21064 and the 21264. So, we don't think we can meet the goal of having the LDx_ARM instructions trap on all previous implementations.

B.4 Implementation

The design intent is that the LDx_ARM/Quiesce mechanism have the following properties:

- For the most part, it makes use of hardware or architectural features or components that already exist to support single threaded uniprocessor operation.
- The quiesce instructions are implemented such that speculative or spurious execution of these instructions in any form or sequence will not result in an UNDEFINED operation. This is accomplished by deferring most state changes related to LDx_ARM and QUIESCE until retire time.
- The quiesce operation eliminates the possibility of quiescing the TPU just as the lock becomes available (this is necessary for conformance to the instruction definitions).
- Restarting after a quiesce is low-overhead.

The LDx_ARM instruction looks very much like a load-lock. The load-lock returns load data, sets lock_flag and loads lock_physical_address. LDx_ARM returns load data, sets watch_flag and loads watch_physical_address. The load data is returned to the LDx_ARM at the time the cache access is done, before retire, as is done also for load-lock. The watch_flag and watch_physical_address are updated when the LDx_ARM retires (just as lock_flag and lock_physical_address are changed when the load-lock retires).

When the LDx_ARM is executed, it is put into the Load Queue in the Mbox. Among other things, the entry contains the physical address specified by the LDx_ARM. If, at any time before the LDx_ARM retires, a memory write of any type occurs to that physical address, the LDx_ARM is aborted and scheduled to be reexecuted. At the time when the LDx_ARM retires successfully, watch_flag is set and watch_physical_address is loaded with the address from the LDx_ARM.

The LDx_ARM instruction, since it has the characteristics of LDx, may incur a DTB miss. In this case the DTB miss is handled before watch_flag and watch_physical_address are affected (since they don't change until retire time).

Similarly, the QUIESCE effects occur at the point of the QUIESCE retiring. This ensures that the instructions are executed in-order, and the watch_flag/ watch_physical_address values loaded by the LDx_ARM are what are used by the QUIESCE.

If the watch_flag is set when the QUIESCE instruction retires, this TPU enters sleep mode. In this case, all instructions subsequent to the QUIESCE are flushed, the QUIESCE_TIMER is started, and the QUIESCE retires. Certain hardware resources on the chip are deallocated from the quiescent TPU (described below). Once watch_flag is cleared, this TPU becomes active again, the map thread chooser resumes mapping at the instruction following the QUIESCE, and the hardware resources are reallocated back to the reactivated TPU.

We implement the QUIESCE trap by treating a QUIESCE as a WMB. When it is issued to the MBOX, the MBOX makes an entry in the store queue for it and waits for the QBOX to signal that the QUIESCE instruction is the next to retire. The MBox then checks the watch_flag: if it is set, the MBOX signals a trap to flush the subsequent instructions. The QUIESCE is allowed to retire.

The quiesce trap is analogous to a branch-mispredict, in that it kills the wrongly speculated instructions and restarts at the next correct instruction after the branch. The QUIESCE retires, all instructions after the QUIESCE are killed, and instruction fetch restarts at the next instruction after the QUIESCE.

Alternate method: The trap clears out all subsequent instructions and instruction fetch restarts at the QUIESCE. In this case, the second QUIESCE would never successfully quiesce the machine, since watch_flag would by definition be clear since no LDx_ARM instruction has set it again. Or, if an interrupt was serviced in this TPU just after finishing the trap - the watch_flag would have been cleared on the REI.

After the TPU's instructions are flushed, instruction fetch resumes for the quiesced TPU. Only at the map thread chooser is this TPU idled (not chosen). This means that when the thread restarts, it can start from the map point, which is much faster than starting at instruction fetch. This brings the idle TPU's instructions as far forward as possible into the pipe, without using Inum space or registers for those waiting instructions.

B.4.1 Interaction of Interrupts and QUIESCE

When a TPU is quiescing, it kills all following instructions and starts refetching from the instruction following the quiesce. These instructions enter the pipe up to the map stage, where they are not chosen for mapping until the quiesce is over.

If the watch_flag is cleared due to an interrupt, the pipeline is already full of the instructions following the QUIESCE. The Ibox starts fetching the ISR but does not disturb the instructions already in the pipeline. Thus, the ISR will be executed at some point downstream from the QUIESCE instruction.

If a branch mispredict on the previously fetched code kills the ISR code, the TPU needs to remember to service the interrupt. This works because the interrupt signal is level-sensitive, and is only cleared once the interrupt servicing routine code is successfully executing.

If an interrupt is directed to a quiesced TPU, the watch_flag is cleared so that the quiesce period will end immediately, in the interest of getting to the interrupt as soon as possible.

If an interrupt is pending to a quiesced TPU, any attempted setting of the watch_flag by a LDx_ARM fails, so that the TPU will not quiesce. Again, this is in the interest of getting to the ISR as soon as possible. This situation could come up if a QUIESCE is already in the pipe with an ISR coming along behind it.

B.4.2 Quiesce-Related Hardware

- QUIESCE_TIMEOUT_VALUE[3:0]<19:0> IPR. One per TPU, each 20 bits wide. This is an implementation-specific IPR, which specifies a limit to the number of CPU cycles that may elapse between the QUIESCE instruction and watch_flag being cleared. This IPR is writable by PALcode; it does not have to be readable, as it is never modified by hardware. The value in this register is used to load the QUIESCE_TIMER (internal 21464 hardware). The default value loaded by hardware at power-up is 10K cycles, which proved to be an effective timeout period in simulation.

  QUIESCE_TIMEOUT_VALUE[3:0]<19:0> is loaded by startup (boot) code. The timeout value can be specified up to 2^20, or 1048576 (1M) cycles.

Note: For ease of implementation, it may be useful to have the bottom two bits be free running, so that the incrementer only has to cycle every fourth cycle.

If software wanted to use a different QUIESCE_TIMEOUT_VALUE for each process that is scheduled on a TPU, then QUIESCE_TIMEOUT_VALUE would have to become part of the process context. We are assuming this is not the case, instead, QUIESCE_TIMEOUT_VALUE is loaded by powerup code for each TPU.

Note: Allowing a different value for each TPU is necessary to provide the capability of running virtual machines, i.e., the ability for different TPUs to run different operating systems simultaneously. If we rule out this design alternative, one single QUIESCE_TIMEOUT_VALUE IPR, used by all TPU's and loaded at powerup is sufficient.

- QUIESCE_TIMER[3:0]<19:0>. This is hardware internal and not accessible by software. There is one QUIESCE_TIMER per TPU. The QUIESCE_TIMER is loaded with the value in the QUIESCE_TIMEOUT_VALUE IPR, when the QUIESCE instruction retires. It then decrements once per CPU cycle. When it reaches zero, watch_flag is cleared, and the timer remains at zero until restarted by the next QUIESCE retiring. Since QUIESCE_TIMER[n] is only started when a QUIESCE retires on TPU[n], it is guaranteed to count down to zero eventually; it can't be restarted speculatively by another QUIESCE. Also, this implementation should give reproducible results, as desired for verification and also for Tandem.
- QUIESCE_TIMEOUT[3:0]. Each TPU has its own QUIESCE_TIMEOUT signal. This signal is asserted for one cycle when QUIESCE_TIMER reaches zero. This assertion has the effect of clearing watch_flag.
- watch_flag[3:0]. As specified by the LDx_ARM and QUIESCE instructions; one per TPU.

- watch_physical_address[3:0]<43:4>. One per TPU. Note that this register does not have to be the full width of the physical address, it could be less wide. In this case watch_flag would be cleared more frequently than with the full-width address.

B.4.3 Reallocation Hardware Resources During Quiesce

For as long as a TPU[j] is quiesced, one of four bits M%QUIESCE_TPU_Q19A_H[3:0] is asserted which has the following effects:

- Thread map chooser no longer chooses instructions from TPU[j]
- Inum allocator allocates no more Inums to TPU[j], and as TPU[j] frees Inums, allows them to be allocated to other, active, TPUs.
- Load queue entries are repartitioned among the remaining active TPUs. (need more description here).
- Store queue entries are repartitioned among the remaining active TPUs. (need more description here).
- As a result of Inum reallocation, non-architectural physical registers gradually migrate from TPU[j] to other TPUs, as they are freed. more description needed here
- Other effects worth mentioning?

When watch_flag is cleared, M%QUIESCE_TPU_Q19A_H[3:0] is deasserted and machine resources are gradually given back to the previously quiesced TPU, so that it can resume execution. At this point, instructions are waiting in the collapsing buffer, ready to be mapped, once chosen by the map chooser. The TPU is out of Inums, and it may take some time for the Inums to become available for the TPU, which is probably the critical resource as far as restarting this TPU.

We estimate it will cost about 100 cycles for a TPU to quiesce and wake up. It will take about 50 cycles, on average, from the time a TPU comes out of quiesce to the time it executes its first instruction. The delay is because while quiesced, the TPU gave up resources (Inums, registers, load queue and store queue entries). It can only gradually get those resources back as other TPUs retire instructions.

It is not desirable to reserve an Inum block for a quiesced TPU, because we would not want to do that for a TPU that is not being used at all, and we want to use the same mechanisms no matter why the TPU is inactive.

B.4.4 Issues to Consider While Finalizing the Hardware Design

- How much performance is lost because LDx_ARM, QUIESCE, LDx_L and STx_C all wait until retire time to have an effect?
- If 4 TPUs are using a lock heavily, is the hardware fair in passing the lock from TPU to TPU? Consider 2 EV8 CPUs, which each have 4 TPUs. A contended lock will have the same kinds of concerns between CPUs as exist today with load-lock/ store conditional - the local TPUs will have an advantage over the remote ones. We need to avoid the situation where the winner keeps winning on repeated uses of the lock.
- Does a TPU that quiesces and times out repeatedly eat up too much of the CPU from the other TPUs?

- For real time applications, should we have a mode so that a quiesced TPU would not give up resources at all (except being scheduled for execution slots).
- How can we make branch predictions go the right way - so that non-contended lock works fast? Can we build a branch predictor hint in somehow?

B.5 Alternative Proposals to the LDx_ARM/QUIESCE Current Design

As the current proposal for LDx_ARM/QUIESCE was being developed, a number of alternatives were considered but not chosen. The alternatives are presented here as background material.

B.5.1 Timer-Based

A purely timer-based approach was studied at CRL, using the 21464 model, but found not to work. A QUIESCE instruction that watched the memory location until it changed was needed to obtain speedups.

B.5.2 Unified QUIESCE Instruction

QUIESCE Ra, (Rb)

This QUIESCE is a load. If a QUIESCE is executed when the watch_flag is clear, it loads watch_physical_address, sets watch_flag and does not quiesce the processor. Thus, it acts as an ARM. If a QUIESCE is executed when the watch_flag is set, it does quiesce the processor. It then stays quiesced until watch_flag is cleared by a store to watch_physical_address.

For the "first" QUIESCE, the load data can be tested by subsequent instructions to find out if the lock is held. For a "second" quiesce, it is unclear what that load means or when it is loaded. It would be nice to load it at the end of the QUIESCE period, to see what it has changed to, but this is very difficult to implement.

Pros:

- Just one instruction.

Cons:

- More difficult to understand and implement.
- Two flavors of the instruction, "first" and "second", are hard to think about.
- Returning meaningful load data to the second QUIESCE would be difficult.
- Specifying what can or can't happen "between" QUIESCEs seems unmanagable.

B.5.3 Use architectural Registers to Enforce LDx_ARM/QUIESCE Dependency

Here, LDQ_ARM is a load and QUIESCE is a store, of sorts.

```
LDQ_ARM R0, (R5)          ; this is a load
BEQ getlock
QUIESCE R0, (R31)         ; this is a "store"
```

Since the QUIESCE reads the value in R0, the already-existing hardware in an out-of-order implementation will naturally keep the QUIESCE in-order with the LDQ_ARM, which it is dependent upon. The watch_physical_address and watch_flag registers are used as in the originally proposed instructions.

However, having these registers explicitly part of the instruction still does not solve the problem of keeping writes/reads to watch_flag and watch_physical_address in order. Hardware still must solve this (the 21464 does it by not accessing them till retire time). So this suggestion does not really solve any problem.

Pros:

- Hardware implementations don't have to take special care to order the instructions.

Cons:

- QUIESCE "looks" like a store but it really isn't; non-intuitive.
- watch_physical_address and watch_flag access order still must be managed by hardware other than the usual register-ordering hardware.

B.5.4 Add LDx_ARM Functionality to LDx_L

The LDx_ARM functionality is overloaded on the LDx_L instruction. Whenever a LDx_L is executed, the watch_physical_address and the watch_flag are set, in addition to the lock_flag and the lock_physical_address. Or, instead of having the watch_flag and watch_physical_address registers at all, the lock_flag and the lock_physical_address could be used both for LDx_L/STx_C functionality and for ARM/QUIESCE functionality. In this case, QUIESCE would watch for the clearing of the lock_flag. The same LDx_L would not be used both as the partner of a QUIESCE and the partner of a STx_C. If the watch* registers are used, LDx_ARM functionality could be specified using the low address bit of the LDx_L to specify ARM. If only the lock* registers are used, no differentiation in the LDx_L instruction is needed.

Pros:

- Have to define only one new instruction (QUIESCE).
- Backwards-compatible code, if QUIESCE is a NOP.
- LDx_L and LDx_ARM already share a lot of functionality.

Cons:

- Overloading the LDx_L instruction (even more difficult to understand and verify)
- Restricts implementations by requiring two functionalities; for example, LDx_L would not be able to request write privileges for a block, since it might be used in conjunction with a QUIESCE rather than a STx_C.
- Using low-order address bit to differentiate functionality seems kludgy.

B.5.5 Define QUIESCE to be a load and test

The idea here is to have the QUIESCE load a value, and quiesce based on that value. QUIESCE Ra, (Rb) would load Ra from the memory address in Rb. Then, the thread would quiesce if the value in Ra was non-zero, and would effectively be a NOP if the value in Ra was a zero. The QUIESCE instruction would also load the watch_flag and the watch_physical_address.

It is too restrictive to have just one flavor of test, so we would have to define different types of QUIESCE, just as there are many types of branches.

Pros:

- LDx_ARM is not needed.
- Coding restrictions not needed.
- Only one instruction accomplishes the functionality.

Cons:

- Too many new instructions to define (multiple flavors)
- Different type of instruction - hardware has to operate on load data (data from memory).

B.5.6 Define QUIESCE to be a read of memory and compare with a register

(Ernie Petrides)

This version of QUIESCE is used as follows:

```
LDQ R0, (R5)
BEQ R0, getlock
QUIESCE R0, (R5)
```

The QUIESCE translates the VA in R5 and reads the lock value from that physical address. It then compares that lock value with the contents of R0, which was previously loaded by a vanilla load preceding the QUIESCE. If the two values are equal, the QUIESCE succeeds and the thread goes to sleep. If they are not equal, the QUIESCE is like a NOP and does not put the thread to sleep.

While the processor is asleep, the hardware watches the PA as calculated when the QUIESCE executed. This is analogous to the watch_physical_address register as defined in other instructions, but is entirely private to the hardware (not software visible at all). The quiesce period ends if some write access happens to that PA, etc.

Pros:

- LDx_ARM is not needed.
- Coding restrictions not needed.
- Only one instruction accomplishes the functionality.
- watch_flag and watch_physical_register do not need to be defined as IPRs or mentioned in the SRM at all.
- Very appealing from a software point of view.

Cons:

- Complicated instruction, unlike any other - loads from memory, reads from a register, does a compare all in the same instruction.
- Difficult to implement - introduces datapath completely unlike anything we have already.

B.6 Open Issues

- Is the lack of a displacement for LDx_ARM a problem? We believe it is not an issue for Unix or VMS.
- Should we add anything to Section B.2.2.1?
- Sections B.2.1.1 and B.2.2.1 use "processor" to mean "TPU" and "CPU" to mean a thing that can contain multiple processors. It seems confusing to use "processor" to mean both "TPU" and "CPU". Does the terminology need to be changed?

Open Issues

C

Proposed Memory Management IPR Design

This appendix proposes a design for the 21464 memory management IPRs from Jeff Wiedemeier, Judy Hall, and Eileen Samberg. Upon approval, it will be incorporated into the body of the Specification.

There are references in this appendix to ECO 129, which is available at:

`http://amt233.lkg.dec.com/alphaarchitect/approved-ecos/eco129/eco129_prelim_mm.doc`

C.1 Motivation for This Design

The 21464 memory management IPR design eliminates bit overloading that can limit the options that are available to an operating system and the PALcode.

Limitation: The same bit (VA_48) that controls sign-extension checking also dictates the format of VA_FORM. When VA_48 is set, VA_FORM is based on a 4-level page table. Software that uses 48-bit superpage and 43-bit (or smaller) mapped addresses has no VA_FORM that works. If software uses 3-level page tables, it must use VA_48=0. This prevents it from using 48-bit superpage and thus from being able to directly address (without mapping) the entire physical address space of the CPU outside of PALmode.

Correction: Separate the control of sign-extension checking from the format control of VA_FORM.

Limitation: Sign extension checking is applied to superpage and mapped addresses equally. If software wants to use 48-bit superpage, sign extension checking must be set up for 48-bit checking. This means that if software uses 48-bit superpage and 43-bit virtual addresses, VA<46:42> are not checked by hardware for proper sign extension and must either be checked by PALcode or ignored and assumed to be correct. Besides being slow, checking by PALcode can only be done in memory-management related traps and therefore cannot catch all cases.

Correction: Modify the sign extension checking algorithm to accommodate the large superpage in all virtual addressing modes. This is how the 21464 will support the mixed mode described in ECO 129.

Limitation: Arbitrarily basing the decision of which double miss flow to use (DTB_MISS_DOUBLE_3 or DTB_MISS_DOUBLE_4) on VA_48 prevents other uses for multiple double miss flows.

Correction: Base the decision of which double miss flow to use on an independent IPR bit rather than VA_48.

C.2 Page Table Assumptions

The following assumptions are made concerning the page tables.

1. The SRM allows only 3-level page tables.

   ECO 129, Section 1, removes 4-level page tables from the architecture.

2. The SRM allows the level 1 page table to be partially utilized.

   ECO 129, Section 3, states:

   > The level one of the page table is partially utilized, similar to the previous 4-level proposal.

   If the level 1 page table is required to be fully-utilized, then 64 KB pages require 55-bit virtual addresses. Since the 21464 implements a 52-bit virtual address, the level 1 page table in 64 KB page mode will not be fully-utilized.

3. At least 2 bits of level 1 page table index must be implemented

   Page (II-A) 3-3 of the Version 7 SRM states:

   > An implementation that supports the fourth level-number field may further subset the supported address space to include only a subset of low-order bits within that field. That subset must be at least two bits[1], and may be as large as n bits, where n is the full bit count of any given level-number field. The most significant bit in the chosen subset is sign-extended to VA<63> for any valid virtual address.
   >
   > [1]OpenVMS requires at least three PTEs in the highest-level page table. The lowest-order PTE must map process space, the highest-order PTE must map system space, and the penultimate PTE maps the page table structure.

   If 2 bits of level 1 page table index must be implemented, then 64 KB pages require at least a 44-bit virtual address [2/13/13/16] but can be used with up to a 55-bit virtual address [13/13/13/16].

C.3 I-Stream (I_CTL) and D-Stream (M_CTL) Control Registers

The following sections define the fields for I_CTL and M_CTL.

C.3.1 I_CTL

The following fields are defined for I_CTL.

Table C–1 I_CTL Field Definitions

Name	Type	Description
PAGE_SIZE	RW	Controls the I-Stream page size: **Value** **Meaning** 1 I-Stream page size is 64 KB 0 I-Stream page size is 8 KB PAGE_SIZE influences the format of IVA_FORM (see Section C.4).
VA_SIZE	RW	Controls the I-Stream virtual address size: **Value** **Meaning** 1 I-Stream virtual address size is 52 bits 0 I-Stream virtual address size is 43 bits VA_SIZE influences the format of IVA_FORM (see Section C.4) and influences sign extension checking (see Section C.5).

Table C–1 I_CTL Field Definitions (Continued)

Name	Type	Description
REDUCED_PAGE_TABLE	RW	Controls reduced page table mode: **Value** **Meaning** 1 Quadrant 1 of the virtual address space (VA<n:n-1> = 01) is the reduced page table region 0 No special handling of quadrant 1 REDUCED_PAGE_TABLE influences the format of IVA_FORM (see Section C.4). See ECO 129 for information on reduced page table mode.
SPE<2:0>	RW	I-Stream Superpage mode enable. **Bit** **Mnemonic** **Meaning When Set** SPE<2> SPE52 Enables superpage mapping when VA<63:50> = 0x3FFE. In this mode VA<47:0> are mapped directly to PA<47:0>. Because the physical address is only 48 bits, VA<49:48> are ignored. SPE<1> SPE43 Enables superpage mapping when VA<63:41> = 0x7FFFFE. In this mode VA<40:0> are mapped directly to PA<40:0> and PA<47:41> are copies of PA<40> (sign extension). SPE<0> SPE32 Enables superpage mapping when VA<63:30> = 0x3FFFFFFFE. In this mode, VA<29:0> are mapped directly to PA<29:0> and PA<47:30> are cleared. o Any non-kernel mode access to an enabled superpage region must result in an access violation. o Any combination of these bits is allowed. o These bits influence sign extension checking (see Section C.5).
DOUBLE_MISS_CONTROL	RW	Controls the vectoring for all double TB misses, both I-Stream and D-Stream, and determines which double miss flow is vectored to when a hw_ld/vpte misses in the TB. **Value** **Meaning** 1 A TB miss on a hw_ld/vpte will vector to DTB_MISS_DOUBLE_ALT. 0 A TB miss on a hw_ld/vpte will vector to DTB_MISS_DOUBLE. DTB_MISS_DOUBLE and DTB_MISS_DOUBLE_ALT are in used in place of the 21264's DTB_MISS_DOUBLE_3 and DTB_MISS_DOUBLE_4, the distinction from the 21264 being that PALcode decides which to use.

C.3.2 M_CTL

The following fields are defined for M_CTL.

Table C–2 M_CTL Field Definitions

Name	Type	Description
PAGE_SIZE	RW	Controls the D-Stream page size: **Value** **Meaning** 1 D-Stream page size is 64 KB. 0 D-Stream page size is 8 KB. PAGE_SIZE influences the format of VA_FORM (see Section C.4).
VA_SIZE	RW	Controls the D-Stream virtual address size: **Value** **Meaning** 1 D-Stream virtual address size is 52 bits. 0 D-Stream virtual address size is 43 bits. VA_SIZE influences the format of VA_FORM (see Section C.4) and influences extension checking (see Section C.5).
REDUCED_PAGE_TABLE	RW	Controls reduced page table mode: **Value** **Meaning** 1 Quadrant 1 of the virtual address space (VA<n:n-1> = 01) is the reduced page table region. 0 No special handling of quadrant 1. REDUCED_PAGE_TABLE influences the format of VA_FORM (see Section C.4). See ECO 129 for information on reduced page table mode.
SPE<2:0>	RW	D-Stream Superpage mode enable. **Bit** **Mnemonic** **Meaning When Set** SPE<2> SPE52 Enables superpage mapping when VA<63:50> = 0x3FFE. In this mode VA<47:0> are mapped directly to PA<47:0>. Because the physical address is only 48-bits, VA<49:48> are ignored. SPE<1> SPE43 Enables superpage mapping when VA<63:41> = 0x7FFFFE. In this mode VA<40:0> are mapped directly to PA<40:0> and PA<47:41> are copies of PA<40> (sign extension). SPE<0> SPE32 Enables superpage mapping when VA<63:30> = 0x3FFFFFFFE. In this mode, VA<29:0> are mapped directly to PA<29:0> and PA<47:30> are cleared. o Any non-kernel mode access to an enabled superpage region must result in an access violation. o Any combination of these bits is allowed. o These bits influence sign extension checking (see Section C.5).

C.3.3 PAGE_SIZE, VA_SIZE, and REDUCED_PAGE_TABLE Field Combinations

Combinations of the PAGE_SIZE, VA_SIZE, and REDUCED_PAGE_TABLE fields are valid or invalid as shown in Table C–3.

Although every valid combination of these bits has PAGE_SIZE and VA_SIZE set the same way, it is recommended that the bits remain separate since the two controls serve distinct functions. One of the primary situations which led to this document was the overloading of bits in previous Alpha implementations.

Table C–3 Valid and Invalid PAGE_SIZE, VA_SIZE, and REDUCED_PAGE_TABLE Combinations

PAGE_SIZE	VA_SIZE	REDUCED_PAGE_TABLE	Description
0	0	0	43-bit VA with 8 KB pages. This is the addressing mode used by Tru64 UNIX and OpenVMS today.
0	0	1	43-bit VA with 8 KB pages and reduced page tables. **This mode is invalid**[1]**.**
0	1	0	52-bit VA with 8 KB pages. This mode is invalid[2].
0	1	1	52-bit VA with 8 KB pages and reduced page tables. **This mode is invalid**[1,2]**.**
1	0	0	43-bit VA with 64 KB pages. **This mode is invalid**[3]**.**
1	0	1	43-bit VA with 64 KB pages and reduced page tables. **This mode is invalid**[3]**.**
1	1	0	52-bit VA with 64 KB pages.
1	1	1	52-bit VA with 64 KB pages and reduced page tables.

[1] Reduced Page Table mode requires 64 KB pages

[2] 3-level page tables with 8 KB pages only allow a 43-bit virtual address

[3] 64 KB pages require at least a 44-bit virtual address

C.4 VA_FORM and IVA_FORM

This is a generalized discussion of the impact of the PAGE_SIZE, VA_SIZE, and REDUCED_PAGE_TABLE IPR bits on the format of VA_FORM and IVA_FORM.

Note: These bits in I_CTL control the format of IVA_FORM; in M_CTL, they control the format of VA_FORM. Because the behavior of VA_FORM and IVA_FORM is the same, VA_FORM represents both VA_FORM and IVA_FORM throughout this discussion.

The effect of these bits on VA_FORM is:

- PAGE_SIZE controls where VA is positioned for inclusion in VA_FORM. If PAGE_SIZE is set, VA<16> is positioned at VA_FORM<3>. If PAGE_SIZE is clear, VA<13> is positioned at VA_FORM<3>. VA_FORM<2:0> are always 0 for alignment of the 64-bit PTE address.

- VA_SIZE controls how many bits are transferred from VA to VA_FORM. If set, VA<51:n> are transferred. If clear, VA<42:n> are transferred. n is either 16 if PAGE_SIZE is set or 13 if PAGE_SIZE is not set.
- REDUCED_PAGE_TABLE controls how VA_FORM is formed in quadrant 1 (VA<n:n-1> == 01) and is only valid in 52-bit VA/64 KB page mode. If set, VA_FORM is formed as discussed in the transformations that follow.

The transformations in Section C.4.1 show how VA_FORM is formed in each of the valid modes.

C.4.1 The Transformation From VA to VA_FORM

The diagrams below show how the transformation from VA to VA_FORM is made for each of the valid combinations of VA_SIZE, PAGE_SIZE, and REDUCED_PAGE_TABLE. The addresses in the diagrams are broken down into their component fields:

L1 - Level 1 PFN

L2 - Level 2 PFN

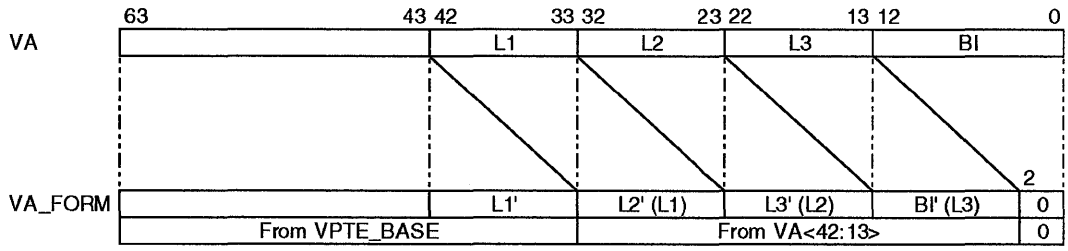
L3 - Level 3 PFN

BI - Byte Index (Offset within page)

The data in VA_FORM is different from the data in VA. Therefore, the component fields in VA_FORM are referred to as L1', L2', L3', and BI'. Additional information indicating where in VA the data came from is listed parenthetically for VA_FORM. So, L2' (L1) indicates that this is the Level 2 PFN in VA_FORM and the data came from the Level 1 PFN of VA.

C.4.2 43-bit VA / 8 KB Page

VA_SIZE = 0, PAGE_SIZE = 0, REDUCED_PAGE_TABLE = 0)

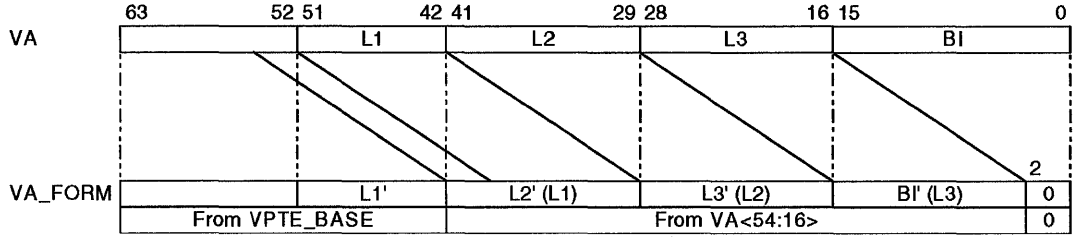


So:

VA_FORM<63:33> comes from VPTE_BASE
VA_FORM<32: 3> comes from VA<42:13>
VA_FORM< 2: 0> is 0

C.4.3 52-bit VA / 64 KB page

VA_SIZE = 1, PAGE_SIZE = 1, REDUCED_PAGE_TABLE = 0



So:

VA_FORM<63:42> comes from VPTE_BASE
VA_FORM<41:39> comes from VA<54:52> or SEXT(VA<51>)
VA_FORM<38: 3> comes from VA<51:16>
VA_FORM< 2: 0> is 0

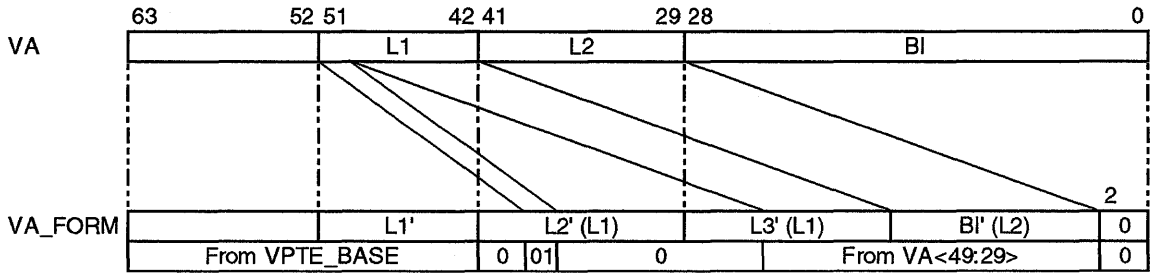
Note:

There are only 52 bits of VA (VA<51:0>), but VA<54:52>, the sign extension of VA<51:0>, is used in VA_FORM. This is required because with 64 KB pages, a 52-bit address does not fully utilize a 3-level page table. With 64 KB pages, 55 bits of virtual address are required to fully utilize a 3-level page table. See Assumptions 2 and 3 at the beginning of this document for the full discussion of partially-utilized page tables.

C.4.4 52-bit VA / 64 KB Page / Reduced Page Tables

VA_SIZE = 1, PAGE_SIZE = 1, REDUCED_PAGE_TABLE = 1

Quadrant 1 (VA<51:50> = 01_2) behaves as described here. VA_FORM is set for all other virtual addresses as described above under 52-bit VA / 64 KB page.



So:

VA_FORM<63:42> comes from VPTE_BASE
VA_FORM<41: 39> is 0
VA_FORM<38:37> comes from VA<51:50> (01_2)
VA_FORM<36:24> is 0
VA_FORM<23: 3> comes from VA<49:29>
VA_FORM< 2: 0> is 0

Note:

ECO 129, section 4 states:

> This reduced page table mode does not modify the format of the PTE's (sic) from the base 64KB mode. The lower 13 bits of the PFN are unused and the GH bits must be all ones (a value of 3) in this mode for the VA<47:46> == 1 space.

In reduced page table mode, the OS is required to set the granularity hints in quadrant 1 such that each second-level PTE maps what an entire third level page table would normally map.

C.5 Sign Extension Checking

C.5.1 Previous Implementation

The 21264 implemented sign extension and superpage checking as shown in the following pseudo-code.

As shown in the code, although 48-bit superpage was the only superpage mode that could directly address the entire physical address space of the processor, it could only be used if 48-bit addressing was turned on. Unfortunately, enabling 48-bit addressing lost some sign-extension validation of legitimate addresses in a 43-bit or smaller virtual addressing environment and broke VA_FORM for 43-bit or 32-bit addressing modes.

```
if ((VA_48 && (VA<63:0> != SEXT(VA<47:0>))) ||
    (!VA_48 && (VA<63:0> != SEXT(VA<42:0>)))) {
     DFAULT;  // improperly sign extended address...
}
if (SPE48 && (VA<47:46> == 2)) {
     if (mode == kernel) {
          PA<43:0> = VA<43:0>;
     } else {
          DFAULT; // superpage access in non-kernel mode
     }
} else if (SPE43 && (VA<47:41> == 0x7E)) {
     if (mode == kernel) {
         PA<43:0> = SEXT(VA<40:0>);
     } else {
          DFAULT; // superpage access in non-kernel mode
     }
} else if (SPE32 && (VA<47:30> == 0x3FFFE)) {
     if (mode == kernel) {
          PA<43:30> = 0;
          PA<29:0> = VA<29:0>;
     } else {
          DFAULT; // superpage access in non-kernel mode
     }
} else {
     PA<43:0> = TBLookup(VA);
}
```

C.5.2 Proposed Implementation

To allow any superpage mode (most importantly, the mode that can directly address the entire physical address space of the processor), the pseudo-code in Section C.5.1 can be changed to the following (assuming IPR bits as discussed above):

```
if (SPE52 && (VA<63:50> == 0x3FFE)) {
     if (mode == kernel) {
          PA<47:0> = VA<47:0>;
     } else {
          DFAULT; // superpage access in non-kernel mode
     }
} else if (SPE43 && (VA<63:41> == 0x7FFFFE)) {
     if (mode == kernel) {
          PA<47:0> = SEXT(VA<40:0>);
     } else {
          DFAULT; // superpage access in non-kernel mode
     }
} else if (SPE32 && (VA<63:30> == 0x3FFFFFFE)) {
     if (mode == kernel) {
          PA<47:30> = 0;
          PA<29:0> = VA<29:0>;
     } else {
          DFAULT; // superpage access in non-kernel mode
     }
} else if (((VA_SIZE == 52-bit) && (VA<63:0> == SEXT(VA<51:0>))) ||
           ((VA_SIZE == 43-bit) && (VA<63:0> == SEXT(VA<42:0>)))) {
     PA<47:0> = TBLookup(VA);
} else {
     DFAULT;  // improperly sign extended address
}
```

Note:

Besides including the superpage checks as peers of the traditional virtual address sign-extension check, the superpage checks are changed to check all the way to bit 63. The superpage checks require that the superpage address be a properly sign-extended address for the size of the superpage region.

Glossary

Bank Conflict

The Dcache is implemented as eight independent, interleaved memories (banks), so that they can perform multiple operations per cycle. If two instructions need the same bank at the same time, they cannot both be satisfied. This event is called a bank conflict, and causes one of the instructions to be retried. Similarly, the Scache has several banks (probably 16) which may be needed in different pipeline stages for different kinds of requests. If two requests conflict for use of the same bank in different stages, one request is retried.

Bbox

BIU (see Cbox) Interface unit which controls the Scache (formerly Bcache), the second-level cache, which is shared by data and instructions.

Bcache

External second-level cache, which has been eliminated from the design in favor of an internal second-level cache called the Scache.

Blacklist

A cache containing the PC addresses of load and store instructions which have recently caused memory order traps.

Whenever a load instruction is found to be on the blacklist, it is forced to wait for completion until all older blacklisted stores have been executed.

Block

A contiguous, naturally aligned 64-byte region of the logical memory space. It may be contained in a single cache which is permitted to modify it, or it may be shared by many caches which have read access. A block is the unit of interprocessor communication, and also the unit managed by the coherence protocol.

We don't use the word block to refer to a group of instructions in the pipeline. Groups of instructions in the pipeline are called "chunks" -- as in "Map Chunk", "Fetch Chunk" and others.

Cache Coherence

In a system with multiple processors, each having a cache, correct operation of the software requires that the caches maintain a consistent (or coherent) representation of the contents of memory. This is accomplished by communication among the caches and memories using a coherence protocol.

CAM

Content addressible memory. A structure that takes an input and compares it with a number of tags and automatically reads the contents of every location that has a matching tag. Although similar to a cache, in a CAM, the tag check and data lookup is integrated. Typically, a CAM is fully associative.

Cbox

Secondary cache, external memory, and system interface, including cache coherence. Documentation in Cbox.

Clean Victim

See Victim.

Coherence Message

In order to ensure that all processors see memory modifications in the same order, the system must make sure that all sharing processors have invalidated their copies of a block before the block is passed from the owner to another processor. Sharing proces ors respond to the invalidate message with a coherence message, and the new owner counts coherence messages to make sure all sharers have been invalidated before forwarding the block.

Complete

An instruction has been completed (alt. is complete) when it has produced a value that can be consumed by its dependents and it has passed the point at which it may itself trigger a trap. (e.g. A STore instruction is not complete until the Mbox has determined that execution of the STore it will not result in a DTB miss.) Even speculatively executed instructions can complete.

Dbox

Data Cache (Dcache, *see* Mbox). First-level data cache.

- 64K Bytes
- Writeback
- 2-way set associative
- 8 banks, interleaved on bits 5-3 of address (quadword banks)
- Double-pumped for two reads or one write per cycle per bank
- Write & victim-extract performed in otherwise-idle banks to avoid conflict

DIFT

Directory In-Flight Table. A list of uncompleted requests to the local memory. Any new request which matches the address of a DIFT entry must wait for release of that entry before being processed. Requests which do not collide with existing DIFT entries are eligible for service by the memory, and are scheduled to optimize memory utilization.

Directory

Information associated with each 64-byte block of memory which indicates which node, if any owns the block (meaning that the node has permission to write the block), and which nodes, if any, are sharers (meaning that they may have cached read-only copies of the block).

Dirty

Descriptive of a block which has been modified with respect to the version of the same block held in memory. In general, a dirty block must eventually be written back to memory, but it may be forwarded among caches for an indefinite period before the writeback occurs, and it is even possible that it will be invalidated before being written back. A dirty block is owned by the cache that holds it, but ownership does not imply that the block is dirty, and it is possible for a processor to obtain ownership speculatively and later evict the block without ever modifying it.

Done

Done-ness is a matter of perspective. While rare is acceptable to many, others prefer well-done or charred. We do not use the term "done" to refer to the status of an instruction. The correct technical term is "outa here".

DTB

Dcache Translation Buffer. A 128-entry, fully-associative cache of virtual-to-physical address translations used for data references. The DTB has 4 read ports, so that it can be accessed by four memory references concurrently, plus one write port so that it can be updated.

Ebox

Execution unit for Integer Operate instructions.

Exclusive

One of the four possible states of a cache block. A block in the Exclusive state is a clean copy of the corresponding data in memory, but no other processor has a copy, and this processor has been granted permission to write it.

Fbox

Execution unit for Floating Point Operate instructions.

Fetch Chunk

The Ibox fetches up to sixteen instructions at one time. The group of instructions that the Ibox fetches in a given cycle is referred to as a fetch chunk.

Fill

The process by which data which was not present in the cache is assigned to a location in the cache, and stored there for future access. Documentation in Filling.

Fill Buffer

A small memory which holds cache blocks from the system or Scache while waiting for an opportunity to update the Dcache.

- 32 entries, each 64 bytes
- 2 write ports, 1 read port

FRD

Fill Retry Data Buffer, pronounced "Fred". A small buffer which can bypass fill data to the load result busses before the Dcache is updated.

Home

The node, consisting of CPU and memory, at which a given 64-byte memory block is stored. The home node number is bits 45-36 of the physical address. The home memory stores both the block and the directory, which identifies the owner and sharing nodes, if any.

Ibox

Instruction Fetch Frontend. Instruction cache with prefetcher, line predictor, branch/jump/return predictor, collapsing buffer, and mapper. Documentation in Ibox.

In Flight

The state of an instruction which has been mapped but not yet retired or trapped. This corresponds to the time during which it may be out of order with respect to other in-flight instructions.

In Flight is also used to describe cache coherence messages which have been sent by one node but not yet processed by the intended recipients.

Instruction Execution

The performance of the operation specified by the opcode of an instruction. In general, an instruction begins execution three cycles after it is issued; its completion time may be one or more cycles later, depending on the operation.

Instruction Issue

The process of assigning an instruction whose operands are ready to a particular function unit for execution beginning in a particular cycle. More precisely, an instruction may be issued as soon as its operands are expected to be ready in time for the instruction to meet the operands at the selected function unit. This can lead to complexity when the operand is the result of a load which needs to be retried, because the successor instruction may already have been issued at the time the retry condition is detected.

Instruction Mapping

The process of identifying the physical registers currently associated with the virtual registers used by an instruction. Mapping also assigns each instruction to a subset of the instruction pickers so that it can be sent to an appropriate function unit. Once it has been mapped, an instruction is kept in the instruction queue while waiting for its operands to become ready, then waiting to be picked for issue to a function unit.

Invalid

One of the possible states of a block in its home directory. The name is confusing, because it does not mean that the block in memory is not valid; in fact, it means that the copy in memory is the only one, because the block is known not to be valid in any cache.

Invalidate

(when used as a noun) A variant of probe used by the coherence protocol, which removes a single cache block if present in the dcache or scache of a processor. Also forces a trap of any load which refers to the cache block and is in the shadow of an MB.

I/O Space

That portion of the processor's physical address space which is not assured to have memory-like behavior (reads and writes may have side-effects, and data may change without having been written). I/O space must not be cached, may not be referred to speculatively, and must be referred to in the order given by the program.

Jbox

Instruction Cache (*See* Ibox).

Kbox

Clocks.

Lbox

Phase Locked Loop.

Load Queue

An associative memory in the Mbox which keeps track of the state of, and addresses used by, load instructions which have been issued but not yet retired or trapped. The load queue contains the information necessary to retry a load which failed to complete when it first issued, and it detects and causes a trap in the case in which an older store (that is, earlier in program order) modifies the data used by this load.

- 64 entries, fully associative
- virtual & physical addresses, opcode attributes
- 3 read ports (2 stores, inval), 3 write ports
- detects next-to-retire, store-data-ready
- partitioned by threads, allocated in order within thread
- controls load retries

The components of the load queue are:

- SSB Index CAM (2 ports). Initiate retry of load when matching store data arrives.
- MAF Index CAM (2 ports). Initiates retry when MAF index matches.
- Flags: DTB Miss, MAF Full, Bank Conflict, Retry Ready, Lock, Valid, Done, DC Hit, I/O.
- Ld Opcode.
- Ld VA register (64 bits).
- Ld PA register, with 3 write ports, 3 CAM ports (2 Stores plus Invalidate).
- Ld Inum comparitors: 2 ports for Stores, to detect RAW hazard trap; plus 1 port to initiate retry at retire point and free entry after instruction retired or killed.

Local

Descriptive of requests from a CPU to the memory attached directly to the same chip. Some local memory transactions can be optimized because of knowlege that the Scache state is always coherent with the memory directory, at the cost of probing the Scache for every memory access by an external node.

Long Latency Instruction

Instructions that return results at either an unpredictable time, or with a latency greater than 4 cycles (is it 4? or 5? or 6?) are termed "long latency instructions". Dependents for these instructions do not become data ready (in the instruction queue) until the long latency instruction signals that it will complete via a bubble request to the IQ. Among other instructions, integer multiply, floating divide, and square root are all long-latency instructions.

MAF

Miss Address File, sometimes called Miss Latch. An associative memory in the interface between the Mbox and the Cbox which keeps track of the address and state of all outstanding requests by the Mbox or Ibox.

- 64 entries: address and flags
- Merges new misses with outstanding fill requests
- Detects new misses to blocks in the Write Buffer or Fill Buffer

Map Chunk

The Ibox sends instructions to the Pbox in groups of up to eight instructions. The eight instructions are contiguous and formed in the collapsing buffer. These groups are referred to as map chunks.

MB

Memory Barrier or Write Memory Barrier. An instruction which requires that all previous memory reference instructions (or in the case of WMB, store instructions) be completed before any subsequent memory reference instructions.

Mbox

Dcache/Internal Memory. First-level cache with load and store queues. Tutorial in Mbox.

Merge Buffer

A intermediate memory in which multiple writes to the same Dcache line are buffered so that they can update the Dcache in the same cycle, and where data waits to be written to the Dcache until there is a cycle in which the required bank is not in use. Helps reduce tag bandwidth, as well as letting stores give priority to fills.

- 8 entries, each 64 bytes of data, plus mask & address
- 2 CAM ports (store address merge), 2 write ports (store retire), 1 read port (Dcache update)

Node

A 21464 CPU chip with its directly-connected memory, if any, or in some circumstances, an I/O interface which participates in the interprocessor communication network. I/O nodes may make requests in the cache coherence protocol, and may keep cached copies of memory data, but never serve as the home node for cacheable memory.

Older

Earlier in program order, though not necessarily in order of execution. (*See* OOO).

OOO

Out of Order execution. The 21464, like many modern microprocessors, attempts to find and exploit opportunities for instruction-level parallelism by looking far ahead, and executing those instructions whose operands are known, even if previous instructions in the program order have not been completed. This often permits the processor to find useful work even while waiting for cache misses and other long delays to be completed. Correct operation of the processor requires that results be the same as if instructions were performed in the order written. Out of order execution is possible but not very effective without speculation.

Out of order execution

See OOO.

Ownership

In order to ensure that the memory system behaves in accordance with the rules set out in the Alpha SRM, we require that the system establish and enforce a particular order in which store instructions are serviced by the memory. This is accomplished by identifying, at each instant, a single cache (the owner) as having permission to write any block.

The directory for the block shows it to be in Exclusive state, and the cache holds it as Exclusive (while it has permission to write but has not yet modified), or Dirty (when it has been modified and remains writable).

Ownership carries extra responsibility, in that the home memory must always be able to identify the owner of every block, and must be able to obtain the block from the owner. Therefore the cache notifies the home memory whenever a block it owns is replaced in the cache, even if it had not been modified (is not dirty).

Packet

In the context of interprocessor communication, the message unit. Packets may consist of 1, 3, or 5 ticks, but the ticks are transmitted in consecutive cycles, and are handled as a unit throughout the system.

Pbox

Dependency mapper unit.

PAF

Probe Address File is a queue which records requests which have been received by this cache but have not been serviced yet. Once serviced, PAF entries may change the MAF state, may cause invalidation of the Dcache and/or forwarding of a block, or may generate a Local Probe Response to the DIFT.

Probe

A cache-coherence transaction which tests a processor's Dcache and Scache for the presence of data from a particular address, optionally modifying the valid/shared/dirty state of the data if found, and/or forcing a transfer of the data to another processor.

Processor

Either a TPU or a CPU.

Qbox

Instruction issue and retire unit.

Quiesce

A variant of a load instruction which is used to put a thread "to sleep", so that it doesn't delay the execution of other threads on the same processor while the sleeping thread waits for release of a lock used for interprocessor communication. It is expected that Quiesce will be used in the spin loop in which a process waits after failing to acquire a lock, and before the next attempt.

Rambus

A high-bandwidth, syncronous interface for dynamic memory chips. The bus consists of 18 data and 15 control signals, clocked at 800 MHz. AraÒa will provide a glueless direct interface to two independent, concurrently active arrays of Rambus memory. Each array consists of 4 parallel busses, each of which can interface up to 32 DRAM chips.

Also the company which designed, developed, and promoted the interface. See their technical overview.

Rbox

Router unit.

Register File

The multi-ported memory consisting of the physical registers which contain integer and floating point values in the virtual registers of some thread.

Requestor

In discussions of the memory system and cache coherence, the node which initiated a transaction, for example, by executing a load instruction which missed the cache, resulting in a read request.

Retire Chunk

The Qbox retires instructions in groups of eight or sixteen at a time. A group of instructions retired all at the same time is referred to as a retire chunk.

Retireable

An instruction is retireable when it is complete and the Mbox has determined that no other instruction may cause this instruction to trap because of a litmus test violation or other ordering constraint.

Retired

An instruction is retired after it becomes retireable and all instructions occurring before it (in program order) are retireable.

Retirement

While the 21464 is able to execute instructions out of order, it has only finite resources for keeping track of all the instructions in progress, and so it must release those resources for reuse after instructions have completed correctly. This release process is called retirement, and is performed in the order specified by the program. Before retirement, an instruction is speculative, and all its effects can be undone; after retirement, an instruction is said to be retired, or committed. This can be particularly confusing in the case of store instructions, which have only begun at the time of retirement. Once a store has retired, its write must be performed (unless a subsequent store from the same processor replaces it), but if the processor does not have exclusive write access to the referenced block, such access must be obtained before the Dcache and Bcache can be updated.

Register Renaming

In order to support out-of-order execution, the processor has many more registers (called physical registers) than are required by the instruction-set architecture (the virtual registers), and it associates a physical register with a particular assignment of a value to a virtual register. When a new value is assigned to a virtual register, a new physical register is chosen to hold the value. This permits the calculation and storage of the new value to be performed before the final use of the old value has taken place, and also facilitates roll-back of the processor state in case of mis-speculation.

Retry

The process which completes a load instruction which was issued but could not complete on the normal schedule. The inability to complete was detected in time to prevent execution of the dependents, so they can be held for execution following the retry. Distinguished from replay trap, which occurs when the dependents may already have executed, and therefore must be flushed.

Router

The interprocessor crossbar switch, incorporated in the AraÒa chip, which permits arrays of processors to be directly interconnected and communicate with and through one another.

Scache

Secondary Cache. Internal cache of 2-4MB, replacing the external Bcache. The Scache is smaller than the Bcache, but has much lower latency and higher bandwidth, which has a larger performance benefit on most benchmarks than the Bcache's size.

Set

In logic, to force true; in arithmetic, to force to one.

In caches, the collection of tags and blocks which are are selected by one address comparitor; one column of a cache.

This usage is contrary to another widely-observed usage, in which a set is the collection of blocks and tags selected by a single index value; one row of the cache. Usage within the Alpha engineering organization seems predominantly to favor the column definition, so we chose to accept this convention, with a warning to those who may be used to the alternative.

Shared

One of the four possible states of a cache block. A block in the Shared state is a clean copy of the corresponding data in memory. Other processors may have copies, and this processor does not have permission to write it. The processor uses SharedToDirty to get write permission.

Sharer

In discussions of the memory system and cache coherence, a node which may have a cached copy of a given block, but not ownership.

Ship Passing

A reference to the expression "Like ships passing in the night", refers to the problem that two elements of the system may see related events in different orders, creating confusion about each other's state. A major source of bugs in pipelined systems.

SMT

Simultaneous Multithreading. Provides, on a single CPU, the capability of simultaneously executing instructions from multiple threads.

Snarf

In bus-snooping coherence protocols, the practice of using one bus transaction both to pass a modified block between caches, and to update the memory.

In the 21464, snarfing requires a separate transaction, but is used with the same intent, namely, to update the memory with modified data that is being passed between processors, with the hope of reducing the network traffic required for forwarding and minimizing interference in the writer's cache.

SRM

Alpha System Reference Manual, the ultimate reference for definition of the software-visible architecture (sometimes called "instruction-set architecture") of Alpha processors.

Speculation

The policy of assuming or predicting some condition before it is known. This permits the processor to discover opportunities for parallel execution, but requires the ability to discard the effect of any operation which depends on a condition which was incorrectly speculated (mis-speculation). There are many kinds of unlikely but possible events which could make the result of an instruction executed out of order different from the result that should have occurred if the program were performed in order, including branch misprediction, memory reference order hazards, and exceptions.

Speculatively Complete

The state of an instruction which has produced its result (hence complete) but has not yet retired (hence speculative).

The dependents of an instruction may issue once it has reached this state.

SSB

Speculative Store Buffer. The memory which holds the addresses and data of store instructions which have been issued but not yet retired and propagated to the Dcache. Used to satisfy younger (later in program order) load instructions in the same thread. SSB slots are assigned as store instructions are mapped. The address and data portions

of a store instruction are issued separately, so may arrive at the SSB in any order. An SSB entry is regarded as not valid until the address arrives, valid but not ready when there is a valid address but no data, and ready when both address and data are present.

- 64 entries, fully associative
- 8 byte data with mask
- virtual & physical addresses, opcode attributes
- 3 CAM ports (loads), 2 write ports (store exec), 2 read ports (store retire) partitioned by threads, allocated in order within thread

The components of the SSB are:

- MAF Index register, with 2 CAM's.
- Snum read decoder.
- Snum write decoder. Writes the SSB entry when selected by the execution of a store instruction assigned to this slot.
- Flags: STC, Evict, WMB, DC Hit, Valid, Retry, I/O.
- Store VA register, with 3 CAM's to match Load addresses.
- Store PA register.
- Store Data Register.
- Store Opcode Register, with 3 load opcode compares (for byte control?)
- Store Inum register, with 3 subtractors for load instructions, and 1 for retire/kill.

Spurious

Spurious instructions are encountered when a program branches or jumps to a sequence of bits that may or may not form valid Alpha instructions. The branch or jump may be incorrectly speculated (i.e. a mispredict) or may be the result of a broken or malicious program or programmer. (Watch out for broken programmers.) Such sequences, by definition, are unlikely to obey coding rules and standards. Spurious instructions, if we arrive at them via a mispredicted flow, may not cause any architecturally visible state changes. If a spurious instruction is retired (i.e. it was not incorrectly speculated) then its effect on architectural state is UNPREDICTABLE if the current processor mode is not "kernel". If the current processor mode is "kernel" then non-speculative spurious instructions may cause the processor to perform an UNDEFINED operation.

STAQ

Store Address Queue. The address portion of the SSB.

Tbox

Box that manages testability and diagnostics.

Thread

The state of a program. A thread consists of the PC, registers, address space, and other state that an Alpha program uses to complete its task. Each 21464 processor is capable of running up to four threads simultaneously, with each behaving as if it had a processor to itself, and it is also possible for any subset of the threads to communicate through shared memory in order to cooperate on completing a single task.

Tick

In general, one cycle. Used specifically in the interprocessor interface ports to refer to the 40-bit unit of information transferred in one cycle of the port. Commands used among the processors are encoded in packets, which consist of 1, 3, or 5 consecutive ticks.

TPU

Thread processing unit. On a simultaneously multithreaded processor, the hardware that is capable of executing a thread. A TPU has all the capabilities of a conventional CPU. The TPU holds a full process context while a process or thread is executing on that TPU. The 21464 contains four TPUs.

Trap

The recovery process when possible mis-speculation is detected. A trap is associated with a particular instruction (the trapped instruction), whose result may not be consistent with sequential execution of the program. Instructions prior to the trapped instruction in program order are permitted to retire normally, but the results of the trapped instruction and all subsequent instructions are discarded, and the processor resumes execution with the trapped instruction. Traps may be described as:

- Replay traps, meaning that the appropriate instruction was executed with incorrect data or timing.
- Branch mispredict traps, where the processor has followed the wrong sequence of instructions.
- Exceptions, where the SRM requires a break in the normal instruction flow because of some data condition (divide by zero, NaN) or processor state (access violation).

UNDEFINED

See definition in the Preface.

UNPREDICTABLE

See definition in the Preface.

VAF

Victim Address File stores the address and state of blocks evicted from the cache which were held exclusively by this processor. Corresponding data is kept in the VDB until it has been sent to the home memory and/or requesting processor.

VDB

Victim Data Buffer After the Scache detects a miss, and before the appropriate fill data is written into the cache line, the victim block must be read out of the Scache. While it is waiting to be written to the memory, it is held in the victim data buffer. The VDB is also used to hold forwarded blocks while they are waiting to be sent to a requestor. Contains 64 entries, each of 64 bytes

Victim

Whenever a new block is filled into the cache, the old contents of the cache line is evicted. If the line was dirty, it had the only current copy of the memory location it represents, so its value must be written back to the system. The line which is about to be replaced in the cache is called the victim, and the process is called victimization.

There are significant circumstances under which a processor is granted ownership of data, but never modifies it before displacing it from the cache. In that case, the processor must notify the home memory that the processor is no longer keeping the data; this is called clean victimization, and no data is sent back to the memory, because the memory is actually still valid.

Virtual Channel

A technique for preventing deadlock in a network by ensuring that when resources are scarce, they are assigned to the messages closest to completion.

Valid, Shared, Dirty (VSD) state

Each block in a cache may be in any of four states, which are encoded in the VSD bits associated with that block. Valid means that the block is a useful representation of the memory location; if Valid is not set, the remaining bits are meaningless. Shared means that there may be other caches with copies of the data, and hence that this processor must negotiate for write permission. Dirty means that this data is modified, and supercedes the value in memory.

Younger

Later in program order, but not necessarily in order of execution. (See Out-of-Order execution).

Zbox

Rambus interface unit.

Some terminology (for glossary):
uITB - Micro Istream Translation Buffer
ITB - the main Istream Translation Buffer
TPU - Thread Processing Unit
TG - Thread Group
PTE - Page Table Entry
IPR - Internal Processor Register
ASN - Address Space Number
ASM - Address Space Match
TBIAG - TB Invalidate All Groups

TBIA - TB Invalidate All
TBIS - TB Invalidate Single
TBIAP - TB Invalidate All Process-specific

Index