COMPAQ

Compiler Writer’s Guide for the Alpha 21264

Order Number: EC–RJ66A–TE

This document provides guidance for writing compilers for the Alpha 21264 microprocessor. You can access this document from the following website:

ftp.digital.com/pub/Digital/info/semiconductor/literature/dsc-library.html

Revision/Update Information: This is a new document.

Compaq Computer Corporation

June 1999

Table of Contents

3 Guidelines for Compiler Writers

Figures

Tables

Preface

Audience

This document provides guidance for compiler writers and other programmers who use the Alpha 21264 microprocessor (referred to as the 21264).

Content

This document contains the following chapters and appendixes:

- Chapter 1, Introduction, introduces the 21264 and provides an overview of the Alpha architecture. It is a direct copy of Chapter 1 of the *21264 Specifications.*
- Chapter 2, Internal Architecture, describes the major hardware functions and the internal chip architecture. It is provided mainly as a reference for Chapter 3 and describes performance measurement facilities, coding rules, and design examples. It is a direct copy of Chapter 2 of the *21264 Specifications*.
- Chapter 3, Guidelines for Compiler Writers, provides guidelines for taking advantage of the hardware features of the 21264 that are described in Chapter 2.
- Appendix A, Ebox Slotting Rules, provides rules to follow in scheduling instructions.
- Appendix B, An Example of Carefully Tuned Code, provides an example for the rules described in Chapter 3 and Appendix A.
- Appendix C, Controlling the Execution Order of Loads, provides information for avoiding load order replay traps.
- Appendix D, 21264 Support for IEEE Floating-Point, describes the 21264 support for IEEE floating-point.
- The Glossary lists and defines terms associated with the 21264.
- An Index is also included.

Documentation Included by Reference

The companion volume to this specification, the *Alpha Architecture Handbook, Version 4*, contains the instruction set architecture. You can access this document from the following website: `ftp.digital.com/pub/Digital/info/semiconductor/literature/dsc-library.html`

Also available is the *Alpha Architecture Reference Manual, Third Edition*, which contains the complete architecture information. That manual is available at bookstores from the Digital Press as EY–W938E–DP.

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.
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.

Abbreviation	Meaning
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 2^n is stored in memory at a byte address that is a multiple of 2^n; 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 2^n is *unaligned* if it is stored in a byte address that is not a multiple of 2^n.

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
Octaword	8	16	128	—

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.

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 21264 (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.

Revision History

The following table lists the revision history for this document.

Date	Revision	Comments
June 17, 1999	1.0	First release

1
Introduction

This chapter provides a brief introduction to the Alpha architecture, Compaq's RISC (reduced instruction set computing) architecture designed for high performance. The chapter then summarizes the specific features of the Alpha 21264 microprocessor (hereafter called the 21264) that implements the Alpha architecture.

The companion volume to this specification, the *Alpha Architecture Handbook, Version 4*, contains the instruction set architecture. Also available is the *Alpha Architecture Reference Manual, Third Edition*, which contains the complete architecture information.

1.1 The Architecture

The Alpha architecture is a 64-bit load and store RISC architecture designed with particular emphasis on speed, multiple instruction issue, multiple processors, and software migration from many operating systems.

All registers are 64 bits long and all operations are performed between 64-bit registers. All instructions are 32 bits long. Memory operations are either load or store operations. All data manipulation is done between registers.

The Alpha architecture supports the following data types:

- 8-, 16-, 32-, and 64-bit integers
- IEEE 32-bit and 64-bit floating-point formats
- VAX architecture 32-bit and 64-bit floating-point formats

In the Alpha architecture, instructions interact with each other only by one instruction writing to a register or memory location and another instruction reading from that register or memory location. This use of resources makes it easy to build implementations that issue multiple instructions every CPU cycle.

The 21264 uses a set of subroutines, called privileged architecture library code (PALcode), that is specific to a particular Alpha operating system implementation and hardware platform. These subroutines provide operating system primitives for context switching, interrupts, exceptions, and memory management. These subroutines can be invoked by hardware or CALL_PAL instructions. CALL_PAL instructions use the function field of the instruction to vector to a specified subroutine. PALcode is written in standard machine code with some implementation-specific extensions to provide

direct access to low-level hardware functions. PALcode supports optimizations for multiple operating systems, flexible memory-management implementations, and multi-instruction atomic sequences.

The Alpha architecture performs byte shifting and masking with normal 64-bit, register-to-register instructions. The 21264 performs single-byte and single-word load and store instructions.

1.1.1 Addressing

The basic addressable unit in the Alpha architecture is the 8-bit byte. The 21264 supports a 48-bit or 43-bit virtual address (selectable under IPR control).

Virtual addresses as seen by the program are translated into physical memory addresses by the memory-management mechanism. The 21264 supports a 44-bit physical address.

1.1.2 Integer Data Types

Alpha architecture supports the four integer data types listed in Table 1–1.

Table 1–1 Integer Data Types

Data Type	Description
Byte	A byte is 8 contiguous bits that start at an addressable byte boundary. A byte is an 8-bit value.
Word	A word is 2 contiguous bytes that start at an arbitrary byte boundary. A word is a 16-bit value.
Longword	A longword is 4 contiguous bytes that start at an arbitrary byte boundary. A longword is a 32-bit value.
Quadword	A quadword is 8 contiguous bytes that start at an arbitrary byte boundary.
Octaword	An octaword is 16 contiguous bytes that start at an arbitrary byte boundary.

Note: Alpha implementations may impose a significant performance penalty when accessing operands that are not naturally aligned. Refer to the *Alpha Architecture Handbook, Version 4* for details.

1.1.3 Floating-Point Data Types

The 21264 supports the following floating-point data types:

- Longword integer format in floating-point unit
- Quadword integer format in floating-point unit
- IEEE floating-point formats
  - S_floating
  - T_floating

- VAX floating-point formats
  - F_floating
  - G_floating
  - D_floating (limited support)

1.2 Microprocessor Features

The 21264 microprocessor is a superscalar pipelined processor. It is packaged in a 587-pin PGA carrier and has removable application-specific heat sinks. A number of configuration options allow its use in a range of system designs ranging from extremely simple uniprocessor systems with minimum component count to high-performance multiprocessor systems with very high cache and memory bandwidth.

The 21264 can issue four Alpha instructions in a single cycle, thereby minimizing the average cycles per instruction (CPI). A number of low-latency and/or high-throughput features in the instruction issue unit and the onchip components of the memory subsystem further reduce the average CPI.

The 21264 and associated PALcode implements IEEE single-precision and double-precision, VAX F_floating and G_floating data types, and supports longword (32-bit) and quadword (64-bit) integers. Byte (8-bit) and word (16-bit) support is provided by byte-manipulation instructions. Limited hardware support is provided for the VAX D_floating data type.

Other 21264 features include:

- The ability to issue up to four instructions during each CPU clock cycle.
- A peak instruction execution rate of four times the CPU clock frequency.
- An onchip, demand-paged memory-management unit with translation buffer, which, when used with PALcode, can implement a variety of page table structures and translation algorithms. The unit consists of a 128-entry, fully-associative data translation buffer (DTB) and a 128-entry, fully-associative instruction translation buffer (ITB), with each entry able to map a single 8KB page or a group of 8, 64, or 512 8KB pages. The allocation scheme for the ITB and DTB is round-robin. The size of each translation buffer entry's group is specified by hint bits stored in the entry. The DTB and ITB implement 8-bit address space numbers (ASN), MAX_ASN=255.
- Two onchip, high-throughput pipelined floating-point units, capable of executing both VAX and IEEE floating-point data types.
- An onchip, 64KB virtually-addressed instruction cache with 8-bit ASNs (MAX_ASN=255).
- An onchip, virtually-indexed, physically-tagged dual-read-ported, 64KB data cache.
- Supports a 48-bit or 43-bit virtual address (program selectable).
- Supports a 44-bit physical address.
- An onchip I/O write buffer with four 64-byte entries for I/O write transactions.
- An onchip, 8-entry victim data buffer.

- An onchip, 32-entry load queue.
- An onchip, 32-entry store queue.
- An onchip, 8-entry miss address file for cache fill requests and I/O read transactions.
- An onchip, 8-entry probe queue, holding pending system port probe commands.
- An onchip, duplicate tag array used to maintain level 2 cache coherency.
- A 64-bit data bus with onchip parity and error correction code (ECC) support.
- Support for an external second-level (Bcache) cache. The size and some timing parameters of the Bcache are programmable.
- An internal clock generator providing a high-speed clock used by the 21264, and two clocks for use by the CPU module.
- Onchip performance counters to measure and analyze CPU and system performance.
- Chip and module level test support, including an instruction cache test interface to support chip and module level testing.
- A 2.2-V external interface.

2
Internal Architecture

This chapter provides both an overview of the 21264 microarchitecture and a system designer's view of the 21264 implementation of the Alpha architecture. The combination of the 21264 microarchitecture and privileged architecture library code (PALcode) defines the chip's implementation of the Alpha architecture. If a certain piece of hardware seems to be "architecturally incomplete," the missing functionality is implemented in PALcode.

This chapter describes the major functional hardware units and is not intended to be a detailed hardware description of the chip. It is organized as follows:

- 21264 microarchitecture
- Pipeline organization
- Instruction issue and retire rules
- Load instructions to R31/F31 (software-directed data prefetch)
- Special cases of Alpha instruction execution
- Memory and I/O address space
- Miss address file (MAF) and load-merging rules
- Instruction ordering
- Replay traps
- I/O write buffer and the WMB instruction
- Performance measurement support
- Floating-point control register
- AMASK and IMPLVER instruction values
- Design examples

2.1 Microarchitecture

The 21264 microprocessor is a high-performance third-generation implementation of the Compaq Alpha architecture. The 21264 consists of the following sections, as shown in Figure 2–1:

- Instruction fetch, issue, and retire unit (Ibox)
- Integer execution unit (Ebox)

- Floating-point execution unit (Fbox)
- Onchip caches (Icache and Dcache)
- Memory reference unit (Mbox)
- External cache and system interface unit (Cbox)
- Pipeline operation sequence

2.1.1 Instruction Fetch, Issue, and Retire Unit

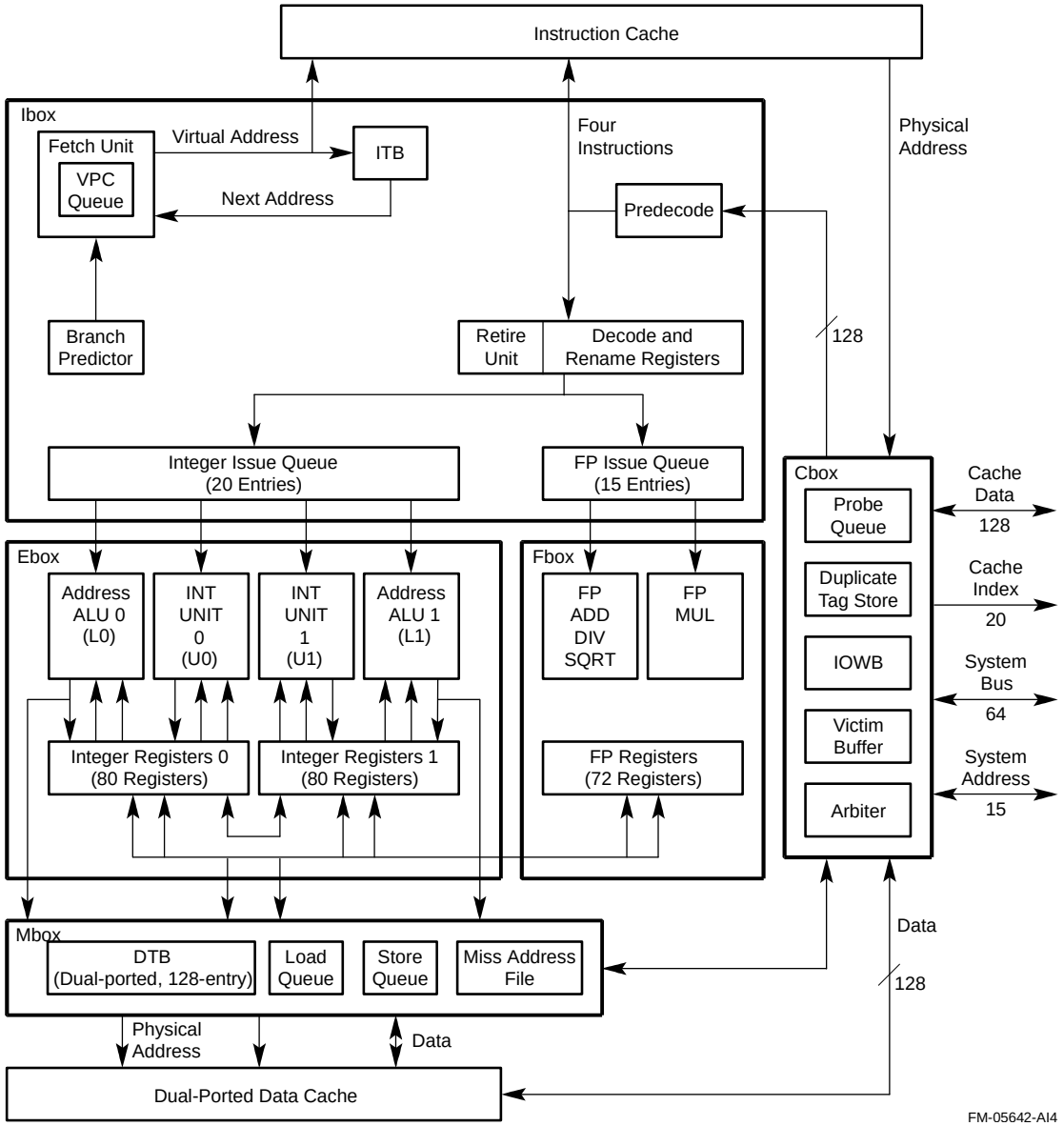
The instruction fetch, issue, and retire unit (Ibox) consists of the following subsections:

- Virtual program counter logic
- Branch predictor
- Instruction-stream translation buffer (ITB)
- Instruction fetch logic
- Register rename maps
- Integer and floating-point issue queues
- Exception and interrupt logic
- Retire logic

2.1.1.1 Virtual Program Counter Logic

The virtual program counter (VPC) logic maintains the virtual addresses for instructions that are in flight. There can be up to 80 instructions, in 20 successive fetch slots, in flight between the register rename mappers and the end of the pipeline. The VPC logic contains a 20-entry table to store these fetched VPC addresses.

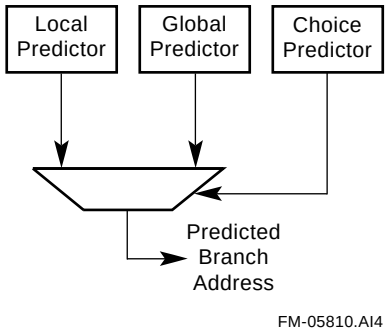
Figure 2–1 21264 Block Diagram



2.1.1.2 Branch Predictor

The branch predictor is composed of three units: the local, global, and choice predictors. Figure 2–2 shows how the branch predictor generates the predicted branch address.

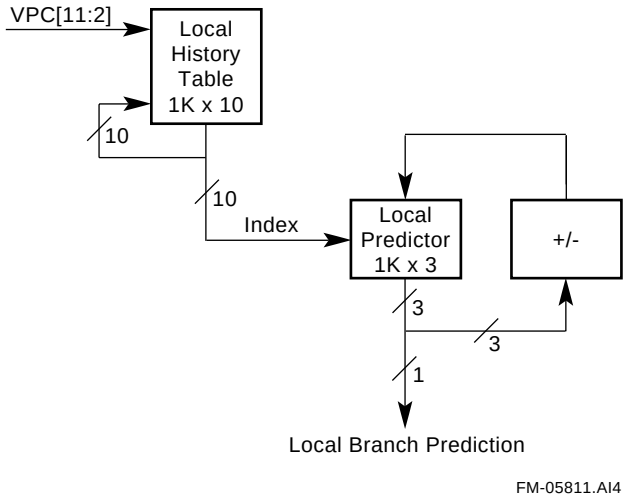
Figure 2–2 Branch Predictor



Local Predictor

The local predictor uses a 2-level table that holds the history of individual branches. The 2-level table design approaches the prediction accuracy of a larger single-level table while requiring fewer total bits of storage. Figure 2–3 shows how the local predictor generates a prediction. Bits [11:2] of the VPC of the current branch are used as the index to a 1K entry table in which each entry is a 10-bit value. This 10-bit value is used as the index to a 1K entry table of 3-bit saturating counters. The value of the saturating counter determines the predication, taken/not-taken, of the current branch.

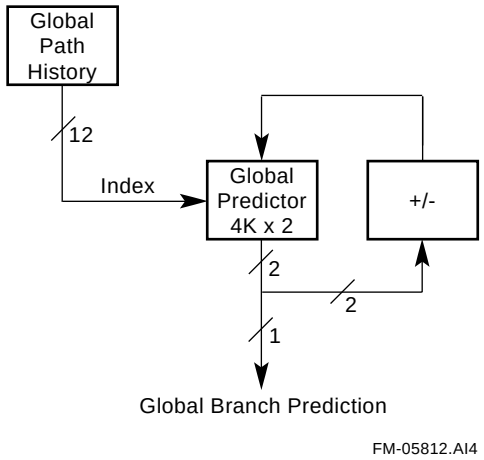
Figure 2–3 Local Predictor



Global Predictor

The global predictor is indexed by a global history of all recent branches. The global predictor correlates the local history of the current branch with all recent branches. Figure 2–4 shows how the global predictor generates a prediction. The global path history is comprised of the taken/not-taken state of the 12 most-recent branches. These 12 states are used to form an index into a 4K entry table of 2-bit saturating counters. The value of the saturating counter determines the predication, taken/not-taken, of the current branch.

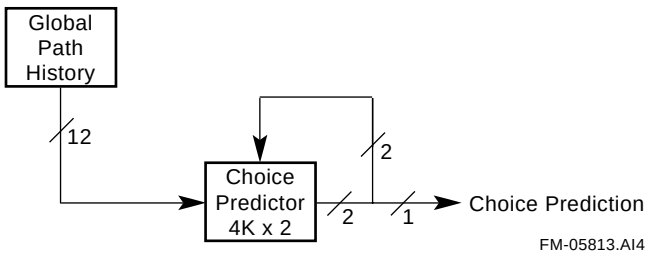
Figure 2–4 Global Predictor



Choice Predictor

The choice predictor monitors the history of the local and global predictors and chooses the best of the two predictors for a particular branch. Figure 2–5 shows how the choice predictor generates its choice of the result of the local or global prediction. The 12-bit global path history (see Figure 2–4) is used to index a 4K entry table of 2-bit saturating counters. The value of the saturating counter determines the choice between the outputs of the local and global predictors.

Figure 2–5 Choice Predictor



2.1.1.3 Instruction-Stream Translation Buffer

The Ibox includes a 128-entry, fully-associative instruction-stream translation buffer (ITB) that is used to store recently used instruction-stream (Istream) address translations and page protection information. Each of the entries in the ITB can map 1, 8, 64, or 512 contiguous 8 KB pages. The allocation scheme is round-robin.

The ITB supports an 8-bit ASN and contains an ASM bit. The Icache is virtually addressed and contains the access-check information, so the ITB is accessed only for Istream references which miss in the Icache.

Istream transactions to I/O address space are UNDEFINED.

2.1.1.4 Instruction Fetch Logic

The instruction prefetcher (predecode) reads an octaword, containing up to four naturally aligned instructions per cycle, from the Icache. Branch prediction and line prediction bits accompany the four instructions. The branch prediction scheme operates most

efficiently when only one branch instruction is contained among the four fetched instructions. The line prediction scheme attempts to predict the Icache line that the branch predictor will generate, and is described in Section 2.2.

An entry from the subroutine return prediction stack, together with set prediction bits for use by the Icache stream controller, are fetched along with the octaword. The Icache stream controller generates fetch requests for additional Icache lines and stores the Istream data in the Icache. There is no separate buffer to hold Istream requests.

2.1.1.5 Register Rename Maps

The instruction prefetcher forwards instructions to the integer and floating-point register rename maps. The rename maps perform the two functions listed here:

- Eliminate register write-after-read (WAR) and write-after-write (WAW) data dependencies while preserving true read-after-write (RAW) data dependencies, in order to allow instructions to be dynamically rescheduled.
- Provide a means of speculatively executing instructions before the control flow previous to those instructions is resolved. Both exceptions and branch mispredictions represent deviations from the control flow predicted by the instruction prefetcher.

The map logic translates each instruction's operand register specifiers from the *virtual* register numbers in the instruction to the *physical* register numbers that hold the corresponding architecturally-correct values. The map logic also renames each instruction's destination register specifier from the virtual number in the instruction to a physical register number chosen from a list of free physical registers, and updates the register maps.

The map logic can process four instructions per cycle. It does not return the physical register, which holds the old value of an instruction's virtual destination register, to the free list until the instruction has been retired, indicating that the control flow up to that instruction has been resolved.

If a branch mispredict or exception occurs, the map logic backs up the contents of the integer and floating-point register rename maps to the state associated with the instruction that triggered the condition, and the prefetcher restarts at the appropriate VPC. At most, 20 valid fetch slots containing up to 80 instructions can be in flight between the register maps and the end of the machine's pipeline, where the control flow is finally resolved. The map logic is capable of backing up the contents of the maps to the state associated with any of these 80 instructions in a single cycle.

The register rename logic places instructions into an integer or floating-point issue queue, from which they are later issued to functional units for execution.

2.1.1.6 Integer Issue Queue

The 20-entry integer issue queue (IQ), associated with the integer execution units (Ebox), issues the following types of instructions at a maximum rate of four per cycle:

- Integer operate
- Integer conditional branch
- Unconditional branch – both displacement and memory format

- Integer and floating-point load and store
- PAL-reserved instructions: HW_MTPR, HW_MFPR, HW_LD, HW_ST, HW_RET
- Integer-to-floating-point (ITOF*x*) and floating-point-to-integer (FTOI*x*)

Each queue entry asserts four request signals—one for each of the Ebox subclusters. A queue entry asserts a request when it contains an instruction that can be executed by the subcluster, if the instruction's operand register values are available within the subcluster.

There are two arbiters—one for the upper subclusters and one for the lower subclusters. (Subclusters are described in Section 2.1.2.) Each arbiter picks two of the possible 20 requesters for service each cycle. A given instruction only requests upper subclusters or lower subclusters, but because many instructions can only be executed in one type or another this is not too limiting.

For example, load and store instructions can only go to lower subclusters and shift instructions can only go to upper subclusters. Other instructions, such as addition and logic operations, can execute in either upper or lower subclusters and are statically assigned before being placed in the IQ.

The IQ arbiters choose between simultaneous requesters of a subcluster based on the age of the request—older requests are given priority over newer requests. If a given instruction requests both lower subclusters, and no older instruction requests a lower subcluster, then the arbiter assigns subcluster L0 to the instruction. If a given instruction requests both upper subclusters, and no older instruction requests an upper subcluster, then the arbiter assigns subcluster U1 to the instruction. This asymmetry between the upper and lower subcluster arbiters is a circuit implementation optimization with negligible overall performance effect.

2.1.1.7 Floating-Point Issue Queue

The 15-entry floating-point issue queue (FQ) associated with the Fbox issues the following instruction types:

- Floating-point operates
- Floating-point conditional branches
- Floating-point stores
- Floating-point register to integer register transfers (FTOI*x*)

Each queue entry has three request lines—one for the add pipeline, one for the multiply pipeline, and one for the two store pipelines. There are three arbiters—one for each of the add, multiply, and store pipelines. The add and multiply arbiters pick one requester per cycle, while the store pipeline arbiter picks two requesters per cycle, one for each store pipeline.

The FQ arbiters pick between simultaneous requesters of a pipeline based on the age of the request—older requests are given priority over newer requests. Floating-point store instructions and FTOI*x* instructions in even-numbered queue entries arbitrate for one store port. Floating-point store instructions and FTOI*x* instructions in odd-numbered queue entries arbitrate for the second store port.

Floating-point store instructions and FTOI*x* instructions are queued in both the integer and floating-point queues. They wait in the floating-point queue until their operand register values are available. They subsequently request service from the store arbiter. Upon being issued from the floating-point queue, they signal the corresponding entry in the integer queue to request service. Upon being issued from the integer queue, the operation is completed.

2.1.1.8 Exception and Interrupt Logic

There are two types of exceptions: faults and synchronous traps. Arithmetic exceptions are precise and are reported as synchronous traps.

The four sources of interrupts are listed as follows:

- Level-sensitive hardware interrupts sourced by the **IRQ_H[5:0]** pins
- Edge-sensitive hardware interrupts generated by the serial line receive pin, performance counter overflows, and hardware corrected read errors
- Software interrupts sourced by the software interrupt request (SIRR) register
- Asynchronous system traps (ASTs)

Interrupt sources can be individually masked. In addition, AST interrupts are qualified by the current processor mode.

2.1.1.9 Retire Logic

The Ibox fetches instructions in program order, executes them out of order, and then retires them in order. The Ibox retire logic maintains the architectural state of the machine by retiring an instruction only if all previous instructions have executed without generating exceptions or branch mispredictions. Retiring an instruction commits the machine to any changes the instruction may have made to the software-visible state. The three software-visible states are listed as follows:

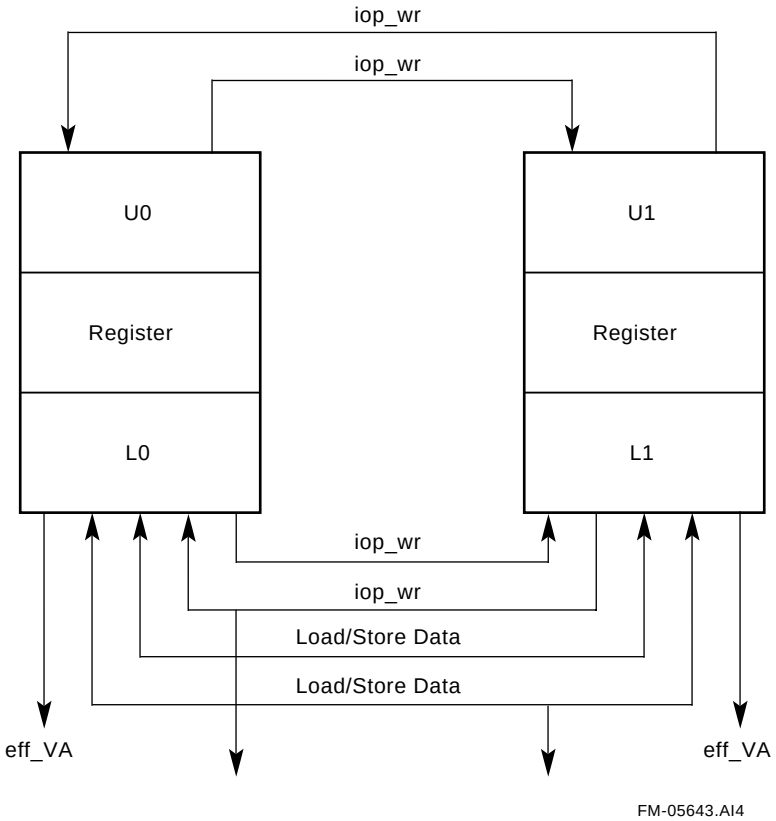
- Integer and floating-point registers
- Memory
- Internal processor registers (including control/status registers and translation buffers)

The retire logic can sustain a maximum retire rate of eight instructions per cycle, and can retire up to as many as 11 instructions in a single cycle.

2.1.2 Integer Execution Unit

The integer execution unit (Ebox) is a 4-path integer execution unit that is implemented as two functional-unit “clusters” labeled 0 and 1. Each cluster contains a copy of an 80-entry, physical-register file and two “subclusters”, named upper (U) and lower (L). Figure 2–6 shows the integer execution unit. In the figure, *iop_wr* is the cross-cluster bus for moving integer result values between clusters.

Figure 2–6 Integer Execution Unit—Clusters 0 and 1



Most instructions have 1-cycle latency for consumers that execute within the same cluster. Also, there is another 1-cycle delay associated with producing a value in one cluster and consuming the value in the other cluster. The instruction issue queue minimizes the performance effect of this cross-cluster delay. The Ebox contains the following resources:

- Four 64-bit adders that are used to calculate results for integer add instructions (located in U0, U1, L0, and L1)
- The adders in the lower subclusters that are used to generate the effective virtual address for load and store instructions (located in L0 and L1)
- Four logic units
- Two barrel shifters and associated byte logic (located in U0 and U1)
- Two sets of conditional branch logic (located in U0 and U1)
- Two copies of an 80-entry register file
- One pipelined multiplier (located in U1) with 7-cycle latency for all integer multiply operations
- One fully-pipelined unit (located in U0), with 3-cycle latency, that executes the following instructions: PERR, MINxxx, MAXxxx, UNPKxx, PKxx

The Ebox has 80 register-file entries that contain storage for the values of the 31 Alpha integer registers (the value of R31 is not stored), the values of 8 PALshadow registers, and 41 results written by instructions that have not yet been retired.

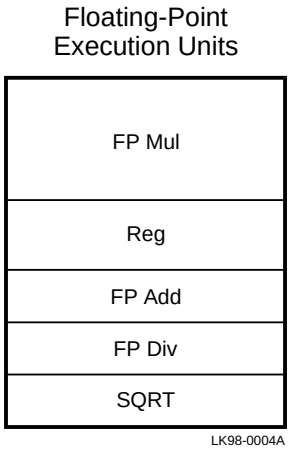
Ignoring cross-cluster delay, the two copies of the Ebox register file contain identical values. Each copy of the Ebox register file contains four read ports and six write ports. The four read ports are used to source operands to each of the two subclusters within a cluster. The six write ports are used as follows:

- Two write ports are used to write results generated within the cluster.
- Two write ports are used to write results generated by the other cluster.
- Two write ports are used to write results from load instructions. These two ports are also used for FTOI*x* instructions.

2.1.3 Floating-Point Execution Unit

The floating-point execution unit (Fbox) has two paths. The Fbox executes both VAX and IEEE floating-point instructions. It supports IEEE S_floating-point and T_floating-point data types and all rounding modes. It also supports VAX F_floating-point and G_floating-point data types, and provides limited support for D_floating-point format. The basic structure of the floating-point execution unit is shown in Figure 2–7.

Figure 2–7 Floating-Point Execution Unit



The Fbox contains the following resources:

- Fully-pipelined multiplier with 4-cycle latency
- 72-entry physical register file
- Fully-pipelined adder with 4-cycle latency
- Nonpipelined divide unit associated with the adder pipeline
- Nonpipelined square root unit associated with the adder pipeline

The 72 Fbox register file entries contain storage for the values of the 31 Alpha floating-point registers (F31 is not stored) and 41 values written by instructions that have not been retired.

The Fbox register file contains six reads ports and four write ports. Four read ports are used to source operands to the add and multiply pipelines, and two read ports are used to source data for store instructions. Two write ports are used to write results generated by the add and multiply pipelines, and two write ports are used to write results from floating-point load instructions.

2.1.4 External Cache and System Interface Unit

The interface for the system and external cache (Cbox) controls the Bcache and system ports. It contains the following structures:

- Victim address file (VAF)
- Victim data file (VDF)
- I/O write buffer (IOWB)
- Probe queue (PQ)
- Duplicate Dcache tag (DTAG)

2.1.4.1 Victim Address File and Victim Data File

The victim address file (VAF) and victim data file (VDF) together form an 8-entry victim buffer used for holding:

- Dcache blocks to be written to the Bcache
- Istream cache blocks from memory to be written to the Bcache
- Bcache blocks to be written to memory
- Cache blocks sent to the system in response to probe commands

2.1.4.2 I/O Write Buffer

The I/O write buffer (IOWB) consists of four 64-byte entries and associated address and control logic used for buffering I/O write data between the store queue and the system port.

2.1.4.3 Probe Queue

The probe queue (PQ) is an 8-entry queue that holds pending system port cache probe commands and addresses.

2.1.4.4 Duplicate Dcache Tag Array

The duplicate Dcache tag (DTAG) array holds a duplicate copy of the Dcache tags and is used by the Cbox when processing Dcache fills, Icache fills, and system port probes.

2.1.5 Onchip Caches

The 21264 contains two onchip primary-level caches.

2.1.5.1 Instruction Cache

The instruction cache (Icache) is a 64KB virtual-addressed, 2-way set-predict cache. Set prediction is used to approximate the performance of a 2-set cache without slowing the cache access time. Each Icache block contains:

- 16 Alpha instructions (64 bytes)
- Virtual tag bits [47:15]
- 8-bit address space number (ASN) field
- 1-bit address space match (ASM) bit
- 1-bit PALcode bit to indicate physical addressing

- Valid bit
- Data and tag parity bits
- Four access-check bits for the following modes: kernel, executive, supervisor, and user (KESU)
- Additional predecoded information to assist with instruction processing and fetch control

2.1.5.2 Data Cache

The data cache (Dcache) is a 64KB, 2-way set-associative, virtually indexed, physically tagged, write-back, read/write allocate cache with 64-byte blocks. During each cycle the Dcache can perform one of the following transactions:

- Two quadword (or shorter) read transactions to arbitrary addresses
- Two quadword write transactions to the same aligned octaword
- Two non-overlapping less-than-quadword writes to the same aligned quadword
- One sequential read and write transaction from and to the same aligned octaword

Each Dcache block contains:

- 64 data bytes and associated quadword ECC bits
- Physical tag bits
- Valid, dirty, shared, and modified bits
- Tag parity bit calculated across the tag, dirty, shared, and modified bits
- One bit to control round-robin set allocation (one bit per two cache blocks)

The Dcache contains two sets, each with 512 rows containing 64-byte blocks per row (that is, 32K bytes of data per set). The 21264 requires two additional bits of virtual address beyond the bits that specify an 8KB page, in order to specify a Dcache row index. A given virtual address might be found in four unique locations in the Dcache, depending on the virtual-to-physical translation for those two bits. The 21264 prevents this aliasing by keeping only one of the four possible translated addresses in the cache at any time.

2.1.6 Memory Reference Unit

The memory reference unit (Mbox) controls the Dcache and ensures architecturally correct behavior for load and store instructions. The Mbox contains the following structures:

- Load queue (LQ)
- Store queue (SQ)
- Miss address file (MAF)
- Dstream translation buffer (DTB)

2.1.6.1 Load Queue

The load queue (LQ) is a reorder buffer for load instructions. It contains 32 entries and maintains the state associated with load instructions that have been issued to the Mbox, but for which results have not been delivered to the processor and the instructions retired. The Mbox assigns load instructions to LQ slots based on the order in which they were fetched from the Icache, then places them into the LQ after they are issued by the IQ. The LQ helps ensure correct Alpha memory reference behavior.

2.1.6.2 Store Queue

The store queue (SQ) is a reorder buffer and graduation unit for store instructions. It contains 32 entries and maintains the state associated with store instructions that have been issued to the Mbox, but for which data has not been written to the Dcache and the instruction retired. The Mbox assigns store instructions to SQ slots based on the order in which they were fetched from the Icache and places them into the SQ after they are issued by the IQ. The SQ holds data associated with store instructions issued from the IQ until they are retired, at which point the store can be allowed to update the Dcache. The SQ also helps ensure correct Alpha memory reference behavior.

2.1.6.3 Miss Address File

The 8-entry miss address file (MAF) holds physical addresses associated with pending Icache and Dcache fill requests and pending I/O space read transactions.

2.1.6.4 Dstream Translation Buffer

The Mbox includes a 128-entry, fully associative Dstream translation buffer (DTB) used to store Dstream address translations and page protection information. Each of the entries in the DTB can map 1, 8, 64, or 512 contiguous 8KB pages. The allocation scheme is round-robin. The DTB supports an 8-bit ASN and contains an ASM bit.

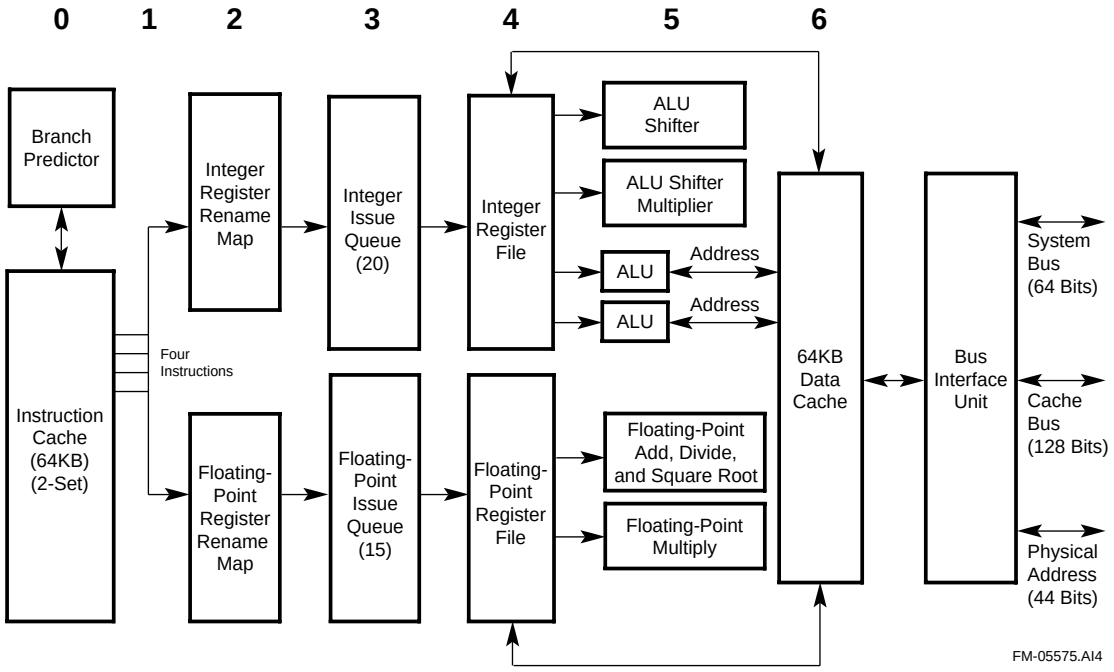
2.1.7 SROM Interface

The serial read-only memory (SROM) interface provides the initialization data load path from a system SROM to the Icache.

2.2 Pipeline Organization

The 7-stage pipeline provides an optimized environment for executing Alpha instructions. The pipeline stages (0 to 6) are shown in Figure 2–8 and described in the following paragraphs.

Figure 2–8 Pipeline Organization



Stage 0 — Instruction Fetch

The branch predictor uses a branch history algorithm to predict a branch instruction target address.

Up to four aligned instructions are fetched from the Icache, in program order. The branch prediction tables are also accessed in this cycle. The branch predictor uses tables and a branch history algorithm to predict a branch instruction target address for one branch or memory format JSR instruction per cycle. Therefore, the prefetcher is limited to fetching through one branch per cycle. If there is more than one branch within the fetch line, and the branch predictor predicts that the first branch will not be taken, it will predict through subsequent branches at the rate of one per cycle, until it predicts a taken branch or predicts through the last branch in the fetch line.

The Icache array also contains a line prediction field, the contents of which are applied to the Icache in the next cycle. The purpose of the line predictor is to remove the pipeline bubble which would otherwise be created when the branch predictor predicts a branch to be taken. In effect, the line predictor attempts to predict the Icache line which the branch predictor will generate. On fills, the line predictor value at each fetch line is initialized with the index of the next sequential fetch line, and later retrained by the branch predictor if necessary.

Stage 1 — Instruction Slot

The Ibox maps four instructions per cycle from the 64KB 2-way set-predict Icache. Instructions are mapped in order, executed dynamically, but are retired in order.

In the slot stage the branch predictor compares the next Icache index that it generates to the index that was generated by the line predictor. If there is a mismatch, the branch predictor wins—the instructions fetched during that cycle are aborted, and the index predicted by the branch predictor is applied to the Icache during the next cycle. Line mispredictions result in one pipeline bubble.

The line predictor takes precedence over the branch predictor during memory format calls or jumps. If the line predictor was trained with a true (as opposed to predicted) memory format call or jump target, then its contents take precedence over the target hint field associated with these instructions. This allows dynamic calls or jumps to be correctly predicted.

The instruction fetcher produces the full VPC address during the fetch stage of the pipeline. The Icache produces the tags for both Icache sets 0 and 1 each time it is accessed. That enables the fetcher to separate set mispredictions from true Icache misses. If the access was caused by a set misprediction, the instruction fetcher aborts the last two fetched slots and refetches the slot in the next cycle. It also retrains the appropriate set prediction bits.

The instruction data is transferred from the Icache to the integer and floating-point register map hardware during this stage. When the integer instruction is fetched from the Icache and slotted into the IQ, the slot logic determines whether the instruction is for the upper or lower subclusters. The slot logic makes the decision based on the resources needed by the (up to four) integer instructions in the fetch block. Although all four instructions need not be issued simultaneously, distributing their resource usage improves instruction loading across the units. For example, if a fetch block contains two instructions that can be placed in either cluster followed by two instructions that must execute in the lower cluster, the slot logic would designate that combination as EELL and slot them as UULL. Slot combinations are described in Section 2.3.1 and Table 2–3.

Stage 2 — Map

Instructions are sent from the Icache to the integer and floating-point register maps during the slot stage and register renaming is performed during the map stage. Also, each instruction is assigned a unique 8-bit number, called an *inum*, which is used to identify the instruction and its program order with respect to other instructions during the time that it is in flight. Instructions are considered to be in flight between the time they are mapped and the time they are retired.

Mapped instructions and their associated inums are placed in the integer and floating-point queues by the end of the map stage.

Stage 3 — Issue

The 20-entry integer issue queue (IQ) issues instructions at the rate of four per cycle. The 15-entry floating-point issue queue (FQ) issues floating-point operate instructions, conditional branch instructions, and store instructions, at the rate of two per cycle. Normally, instructions are deleted from the IQ or FQ two cycles after they are issued. For example, if an instruction is issued in cycle *n*, it remains in the FQ or IQ in cycle *n*+1 but does not request service, and is deleted in cycle *n*+2.

Stage 4 — Register Read

Instructions issued from the issue queues read their operands from the integer and floating register files and receive bypass data.

Stage 5 — Execute

The Ebox and Fbox pipelines begin execution.

Stage 6 — Dcache Access

Memory reference instructions access the Dcache and data translation buffers. Normally load instructions access the tag and data arrays while store instructions only access the tag arrays. Store data is written to the store queue where it is held until the store instruction is retired. Most integer operate instructions write their register results in this cycle.

2.2.1 Pipeline Aborts

The abort penalty as given is measured from the cycle after the fetch stage of the instruction which triggers the abort to the fetch stage of the new target, ignoring any Ibox pipeline stalls or queuing delay that the triggering instruction might experience. Table 2–1 lists the timing associated with each common source of pipeline abort.

Table 2–1 Pipeline Abort Delay (GCLK Cycles)

Abort Condition	Penalty (Cycles)	Comments
Branch misprediction	7	Integer or floating-point conditional branch misprediction.
JSR misprediction	8	Memory format JSR or HW_RET.
Mbox order trap	14	Load-load order or store-load order.
Other Mbox replay traps	13	—
DTB miss	13	—
ITB miss	7	—
Integer arithmetic trap	12	—
Floating-point arithmetic trap	13+latency	Add latency of instruction. See Section 2.3.3 for instruction latencies.

2.3 Instruction Issue Rules

This section defines instruction classes, the functional unit pipelines to which they are issued, and their associated latencies.

2.3.1 Instruction Group Definitions

Table 2–2 lists the instruction class, the pipeline assignments, and the instructions included in the class.

Table 2–2 Instruction Name, Pipeline, and Types

Class Name	Pipeline	Instruction Type
ild	L0, L1	All integer load instructions
fld	L0, L1	All floating-point load instructions
ist	L0, L1	All integer store instructions
fst	FST0, FST1, L0, L1	All floating-point store instructions
lda	L0, L1, U0, U1	LDA, LDAH
mem_misc	L1	WH64, ECB, WMB
rpcc	L1	RPCC
rx	L1	RS, RC
mxpr	L0, L1 (depends on IPR)	HW_MTPR, HW_MFPR
ibr	U0, U1	Integer conditional branch instructions
jsr	L0	BR, BSR, JMP, CALL, RET, COR, HW_RET, CALL_PAL
iadd	L0, U0, L1, U1	Instructions with opcode 10_{16}, except CMPBGE
ilog	L0, U0, L1, U1	AND, BIC, BIS, ORNOT, XOR, EQV, CMPBGE
ishf	U0, U1	Instructions with opcode 12_{16}
cmov	L0, U0, L1, U1	Integer CMOV - either cluster
imul	U1	Integer multiply instructions
imisc	U0	PERR, MINxxx, MAXxxx, PKxx, UNPKxx
fbr	FA	Floating-point conditional branch instructions
fadd	FA	All floating-point operate instructions except multiply, divide, square root, and conditional move instructions
fmul	FM	Floating-point multiply instruction
fcmov1	FA	Floating-point CMOV—first half
fcmov2	FA	Floating-point CMOV— second half
fdiv	FA	Floating-point divide instruction
fsqrt	FA	Floating-point square root instruction
nop	None	TRAP, EXCB, UNOP - LDQ_U R31, 0(Rx)
ftoi	FST0, FST1, L0, L1	FTOIS, FTOIT
itof	L0, L1	ITOFS, ITOFF, ITOFT
mx_fpcr	FM	Instructions that move data from the floating-point control register

2.3.2 Ebox Slotting

Instructions that are issued from the IQ, and could execute in either upper or lower Ebox subclusters, are slotted to one pair or the other during the pipeline mapping stage based on the instruction mixture in the fetch line. The codes that are used in Table 2–3 are as follows:

Code	Meaning
U	The instruction only executes in an upper subcluster.
L	The instruction only executes in a lower subcluster.
E	The instruction could execute in either an upper or lower subcluster, or the instruction does not execute in an integer pipeline (such as floating-point instructions).

Table 2–3 defines the slotting rules. The table field *Instruction Class 3, 2, 1 and 0* identifies each instruction's location in the fetch line by the value of bits [3:2] in its PC.

Table 2–3 Instruction Group Definitions and Pipeline Unit

Instruction Class 3 2 1 0	Slotting 3 2 1 0	Instruction Class 3 2 1 0	Slotting 3 2 1 0
E E E E	U L U L	L L L L	L L L L
E E E L	U L U L	L L L U	L L L U
E E E U	U L L U	L L U E	L L U U
E E L E	U L L U	L L U L	L L U L
E E L L	U U L L	L L U U	L L U U
E E L U	U L L U	L U E E	L U L U
E E U E	U L U L	L U E L	L U U L
E E U L	U L U L	L U E U	L U L U
E E U U	L L U U	L U L E	L U L U
E L E E	U L U L	L U L L	L U L L
E L E L	U L U L	L U L U	L U L U
E L E U	U L L U	L U U E	L U U L
E L L E	U L L U	L U U L	L U U L
E L L L	U L L L	L U U U	L U U U
E L L U	U L L U	U E E E	U L U L
E L U E	U L U L	U E E L	U L U L
E L U L	U L U L	U E E U	U L L U
E L U U	L L U U	U E L E	U L L U
E U E E	L U L U	U E L L	U U L L
E U E L	L U U L	U E L U	U L L U
E U E U	L U L U	U E U E	U L U L

Table 2–3 Instruction Group Definitions and Pipeline Unit (Continued)

Instruction Class 3 2 1 0	Slotting 3 2 1 0	Instruction Class 3 2 1 0	Slotting 3 2 1 0
E U L E	L U L U	U E U L	U L U L
E U L L	U U L L	U E U U	U L U U
E U L U	L U L U	U L E E	U L U L
E U U E	L U U L	U L E L	U L U L
E U U L	L U U L	U L E U	U L L U
E U U U	L U U U	U L L E	U L L U
L E E E	L U L U	U L L L	U L LL
L E E L	L U U L	U L L U	U L L U
L E E U	L U L U	U L U E	U L U L
L E L E	L U L U	U L U L	U L U L
L E L L	L U L L	U L U U	U L U U
L E L U	L U L U	U U E E	U U L L
L E U E	L U U L	U U E L	U U L L
L E U L	L U U L	U U E U	U U L U
L E U U	L L U U	U U L E	U U L L
L L E E	L L U U	U U L L	U U L L
L L E L	L L U L	U U L U	U U L U
L L E U	L L U U	U U U E	U U U L
L L L E	L L L U	U U U L	U U U L
—	—	U U U U	U U U U

2.3.3 Instruction Latencies

After an instruction is placed in the IQ or FQ, its issue point is determined by the availability of its register operands, functional unit(s), and relationship to other instructions in the queue. There are register producer-consumer dependencies and dynamic functional unit availability dependencies that affect instruction issue. The mapper removes register producer-producer dependencies.

The latency to produce a register result is generally fixed. The one exception is for load instructions that miss the Dcache. Table 2–4 lists the latency, in cycles, for each instruction class.

Table 2–4 Instruction Class Latency in Cycles

Class	Latency	Comments
ild	3 13+	Dcache hit. Dcache miss, latency with 6-cycle Bcache. Add additional Bcache loop latency if Bcache latency is greater than 6 cycles.
fld	4 14+	Dcache hit. Dcache miss, latency with 6-cycle Bcache. Add additional Bcache loop latency if Bcache latency is greater than 6 cycles.
ist	—	Does not produce register value.
fst	—	Does not produce register value.
rpcc	1	Possible 1-cycle cross-cluster delay.
rx	1	—
mxpr	1 or 3	HW_MFPR: Ebox IPRs = 1. Ibox and Mbox IPRs = 3. HW_MTPR does not produce a register value.
icbr	—	Conditional branch. Does not produce register value.
ubr	3	Unconditional branch. Does not produce register value.
jsr	3	—
iadd	1	Possible 1-cycle Ebox cross-cluster delay.
ilog	1	Possible 1-cycle Ebox cross-cluster delay.
ishf	1	Possible 1-cycle Ebox cross-cluster delay.
cmov1	1	Only consumer is cmov2. Possible 1-cycle Ebox cross-cluster delay.
cmov2	1	Possible 1-cycle Ebox cross-cluster delay.
imul	7	Possible 1-cycle Ebox cross-cluster delay.
imisc	3	Possible 1-cycle Ebox cross-cluster delay.
fcbr	—	Does not produce register value.
fadd	4 6	Consumer other than fst or ftoi. Consumer fst or ftoi. Measured from when an fadd is issued from the FQ to when an fst or ftoi is issued from the IQ.
fmul	4 6	Consumer other than fst or ftoi. Consumer fst or ftoi. Measured from when an fmul is issued from the FQ to when an fst or ftoi is issued from the IQ.
fcmov1	4	Only consumer is fcmov2.
fcmov2	4 6	Consumer other than fst. Consumer fst or ftoi. Measured from when an fcmov2 is issued from the FQ to when an fst or ftoi is issued from the IQ.

Table 2–4 Instruction Class Latency in Cycles (Continued)

Class	Latency	Comments
fdiv	12	Single precision - latency to consumer of result value.
	9	Single precision - latency to using divider again.
	15	Double precision - latency to consumer of result value.
	12	Double precision - latency to using divider again.
fsqrt	18	Single precision - latency to consumer of result value.
	15	Single precision - latency to using unit again.
	33	Double precision - latency to consumer of result value.
	30	Double precision - latency to using unit again.
ftoi	3	—
itof	4	—
nop	—	Does not produce register value.

2.4 Instruction Retire Rules

An instruction is retired when it has been executed to completion, and all previous instructions have been retired. The execution pipeline stage in which an instruction becomes eligible to be retired depends upon the instruction's class.

Table 2–5 gives the minimum retire latencies (assuming that all previous instructions have been retired) for various classes of instructions.

Table 2–5 Minimum Retire Latencies for Instruction Classes

Instruction Class	Retire Stage	Comments
Integer conditional branch	7	—
Integer multiply	7/13	Latency is 13 cycles for the MUL/V instruction.
Integer operate	7	—
Memory	10	—
Floating-point add	11	—
Floating-point multiply	11	—
Floating-point DIV/SQRT	11 + latency	Add latency of unit reuse for the instruction indicated in Table 2–4. For example, latency for a single-precision fdiv would be 11 plus 9 from Table 2–4. Latency is 11 if hardware detects that no exception is possible (see Section 2.4.1).
Floating-point conditional branch	11	Branch instruction mispredict is reported in stage 7.
BSR/JSR	10	JSR instruction mispredict is reported in stage 8.

2.4.1 Floating-Point Divide/Square Root Early Retire

The floating-point divider and square root unit can detect that, for many combinations of source operand values, no exception can be generated. Instructions with these operands can be retired before the result is generated. When detected, they are retired with the same latency as the FP add class. Early retirement is not possible for the following instruction/operand/architecture state conditions:

- Instruction is not a DIV or SQRT.
- SQRT source operand is negative.
- Divide operand exponent_a is 0.
- Either operand is NaN or INF.
- Divide operand exponent_b is 0.
- Trapping mode is /I (inexact).
- INE status bit is 0.

Early retirement is also not possible for divide instructions if the resulting exponent has any of the following characteristics (EXP is the result exponent):

- DIVT, DIVG: (EXP >= $3FF_{16}$) OR (EXP <= 2_{16})
- DIVS, DIVF: (EXP >= $7F_{16}$) OR (EXP <= 382_{16})

2.5 Retire of Operate Instructions into R31/F31

Many instructions that have R31 or F31 as their destination are retired immediately upon decode (stage 3). These instructions do not produce a result and are removed from the pipeline as well. They do not occupy a slot in the issue queues and do not occupy a functional unit. However, they do affect the slotting of other instructions contained in the same aligned octaword, as defined in Table 2–3. Table 2–6 lists these instructions and some of their characteristics. The instruction type in Table 2–6 is from Table C-6 in Appendix C of the *Alpha Architecture Handbook, Version 4.*

Table 2–6 Instructions Retired Without Execution

Instruction Type	Notes
INTA, INTL, INTM, INTS	All with R31 as destination.
FLTI, FLTL, FLTV	All with F31 as destination. MT_FPCR is not included because it has no destination—it is never removed from the pipeline.
LDQ_U	All with R31 as destination.
MISC	TRAPB and EXCB are always removed. Others are never removed.
FLTS	All (SQRT, ITOF) with F31 as destination.

2.6 Load Instructions to R31 and F31

This section describes how the 21264 processes software-directed prefetch transactions and load instructions with a destination of R31 and F31.

Load operations to R31 and F31 may generate exceptions. These exceptions must be dismissed by PALcode.

2.6.1 Normal Prefetch: LDBU, LDF, LDG, LDL, LDT, LDWU Instructions

The 21264 processes these instructions as normal cache line prefetches. If the load instruction hits the Dcache, the instruction is dismissed, otherwise the addressed cache block is allocated into the Dcache.

2.6.2 Prefetch with Modify Intent: LDS Instruction

The 21264 processes an LDS instruction, with F31 as the destination, as a prefetch with modify intent transaction (ReadBlkModSpec command). If the transaction hits a dirty Dcache block, the instruction is dismissed. Otherwise, the addressed cache block is allocated into the Dcache for write access, with its dirty and modified bits set.

2.6.3 Prefetch, Evict Next: LDQ Instruction

The 21264 processes this instruction like a normal prefetch transaction (ReadBlkSpec command), with one exception—if the load misses the Dcache, the addressed cache block is allocated into the Dcache, but the Dcache set allocation pointer is left pointing to this block. The next miss to the same Dcache line will evict the block. For example, this instruction might be used when software is reading an array that is known to fit in the offchip Bcache, but will not fit into the onchip Dcache. In this case, the instruction ensures that the hardware provides the desired prefetch function without displacing useful cache blocks stored in the other set within the Dcache.

2.7 Special Cases of Alpha Instruction Execution

This section describes the mechanisms that the 21264 uses to process irregular instructions in the Alpha instruction set, and cases in which the 21264 processes instructions in a non-intuitive way.

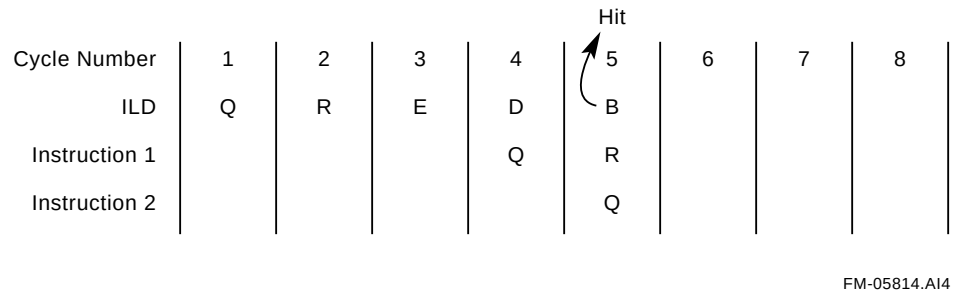
2.7.1 Load Hit Speculation

The latency of integer load instructions that hit in the Dcache is three cycles.

Figure 2–9 shows the pipeline timing for these integer load instructions. In Figure 2–9:

Symbol	Meaning
Q	Issue queue
R	Register file read
E	Execute
D	Dcache access
B	Data bus active

Figure 2–9 Pipeline Timing for Integer Load Instructions



There are two cycles in which the IQ may speculatively issue instructions that use load data before Dcache hit information is known. Any instructions that are issued by the IQ within this 2-cycle speculative window are kept in the IQ with their requests inhibited until the load instruction's hit condition is known, even if they are not dependent on the load operation. If the load instruction hits, then these instructions are removed from the queue. If the load instruction misses, then the execution of these instructions is aborted and the instructions are allowed to request service again.

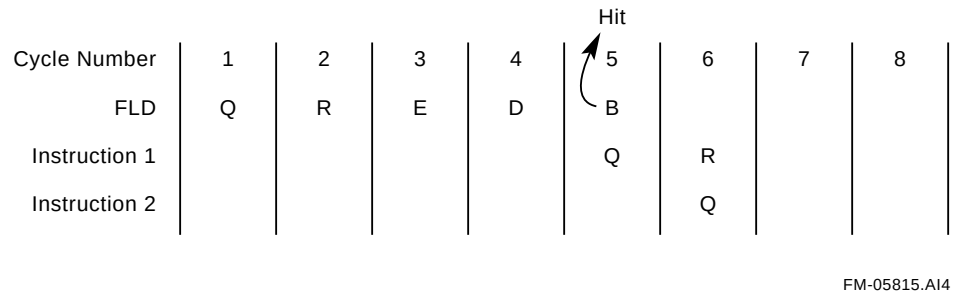
For example, in Figure 2–9, instruction 1 and instruction 2 are issued within the speculative window of the load instruction. If the load instruction hits, then both instructions will be deleted from the queue by the start of cycle 7—one cycle later than normal for instruction 1 and at the normal time for instruction 2. If the load instruction misses, both instructions are aborted from the execution pipelines and may request service again in cycle 6.

IQ-issued instructions are aborted if issued within the speculative window of an integer load instruction that missed in the Dcache, even if they are not dependent on the load data. However, if software misses are likely, the 21264 can still benefit from scheduling the instruction stream for Dcache miss latency. The 21264 includes a saturating counter that is incremented when load instructions hit and is decremented when load instructions miss. When the upper bit of the counter equals zero, the integer load latency is increased to five cycles and the speculative window is removed. The counter is 4 bits wide and is incremented by 1 on a hit and is decremented by two on a miss.

Since load instructions to R31 do not produce a result, they do not create a speculative window when they execute and, therefore, never waste IQ-issue cycles if they miss. Floating-point load instructions that hit in the Dcache have a latency of four cycles. Figure 2–10 shows the pipeline timing for floating-point load instructions. In Figure 2–10:

Symbol	Meaning
Q	Issue queue
R	Register file read
E	Execute
D	Dcache access
B	Data bus active

Figure 2–10 Pipeline Timing for Floating-Point Load Instructions



The speculative window for floating-point load instructions is one cycle wide. FQ-issued instructions that are issued within the speculative window of a floating-point load instruction that has missed, are only aborted if they depend on the load being successful.

For example, in Figure 2–10 instruction 1 is issued in the speculative window of the load instruction.

If instruction 1 is not a user of the data returned by the load instruction, then it is removed from the queue at its normal time (at the start of cycle 7).

If instruction 1 is dependent on the load instruction data and the load instruction hits, instruction 1 is removed from the queue one cycle later (at the start of cycle 8). If the load instruction misses, then instruction 1 is aborted from the Fbox pipeline and may request service again in cycle 7.

2.7.2 Floating-Point Store Instructions

Floating-point store instructions are duplicated and loaded into both the IQ and the FQ from the mapper. Each IQ entry contains a control bit, fpWait, that when set prevents that entry from asserting its requests. This bit is initially set for each floating-point store instruction that enters the IQ, unless it was the target of a replay trap. The instruction's FQ clone is issued when its Ra register is about to become clean, resulting in its IQ clone's fpWait bit being cleared and allowing the IQ clone to issue and be executed by the Mbox. This mechanism ensures that floating-point store instructions are always issued to the Mbox, along with the associated data, without requiring the floating-point register dirty bits to be available within the IQ.

2.7.3 CMOV Instruction

For the 21264, the Alpha CMOV instruction has three operands, and so presents a special case. The required operation is to move either the value in register Rb or the value from the old physical destination register into the new destination register, based upon the value in Ra. Since neither the mapper nor the Ebox and Fbox data paths are otherwise required to handle three operand instructions, the CMOV instruction is decomposed by the Ibox pipeline into two 2-operand instructions:

The Alpha architecture instruction	CMOV Ra, Rb ? Rc
Becomes the 21264 instructions	CMOV1 Ra, oldRc ⇒ newRc1 CMOV2 newRc1, Rb ⇒ newRc2

The first instruction, CMOV1, tests the value of Ra and records the result of this test in a 65th bit of its destination register, newRc1. It also copies the value of the old physical destination register, oldRc, to newRc1.

The second instruction, CMOV2, then copies either the value in newRc1 or the value in Rb into a second physical destination register, newRc2, based on the CMOV *predicate* bit stored in newRc1.

In summary, the original CMOV instruction is decomposed into two dependent instructions that each use a physical register from the free list.

To further simplify this operation, the two component instructions of a CMOV instruction are driven through the mappers in successive cycles. Hence, if a fetch line contains *n* CMOV instructions, it takes *n*+1 cycles to run that fetch line through the mappers.

For example, the following fetch line:

```
ADD CMOVx SUB CMOVy
```

Results in the following three map cycles:

```
ADD CMOVx1

CMOVx2 SUB CMOVy1

CMOVy2
```

The Ebox executes integer CMOV instructions as two distinct 1-cycle latency operations. The Fbox add pipeline executes floating-point CMOV instructions as two distinct 4-cycle latency operations.

2.8 Memory and I/O Address Space Instructions

This section provides an overview of the way the 21264 processes memory and I/O address space instructions.

The 21264 supports, and internally recognizes, a 44-bit physical address space that is divided equally between memory address space and I/O address space. Memory address space resides in the lower half of the physical address space (PA[43]=0) and I/O address space resides in the upper half of the physical address space (PA[43]=1).

The IQ can issue any combination of load and store instructions to the Mbox at the rate of two per cycle. The two lower Ebox subclusters, L0 and L1, generate the 48-bit effective virtual address for these instructions.

An instruction is defined to be *newer* than another instruction if it follows that instruction in program order and is *older* if it precedes that instruction in program order.

2.8.1 Memory Address Space Load Instructions

The Mbox begins execution of a load instruction by translating its virtual address to a physical address using the DTB and by accessing the Dcache. The Dcache is virtually indexed, allowing these two operations to be done in parallel. The Mbox puts information about the load instruction, including its physical address, destination register, and data format, into the LQ.

If the requested physical location is found in the Dcache (a hit), the data is formatted and written into the appropriate integer or floating-point register. If the location is not in the Dcache (a miss), the physical address is placed in the miss address file (MAF) for processing by the Cbox. The MAF performs a merging function in which a new miss address is compared to miss addresses already held in the MAF. If the new miss address points to the same Dcache block as a miss address in the MAF, then the new miss address is discarded.

When Dcache fill data is returned to the Dcache by the Cbox, the Mbox satisfies the requesting load instructions in the LQ.

2.8.2 I/O Address Space Load Instructions

Because I/O space load instructions may have side effects, they cannot be performed speculatively. When the Mbox receives an I/O space load instruction, the Mbox places the load instruction in the LQ, where it is held until it retires. The Mbox replays retired I/O space load instructions from the LQ to the MAF in program order, at a rate of one per GCLK cycle.

The Mbox allocates a new MAF entry to an I/O load instruction and increases I/O bandwidth by attempting to merge I/O load instructions in a merge register. Table 2–7 shows the rules for merging data. The columns represent the load instructions replayed to the MAF while the rows represent the size of the load in the merge register.

Table 2–7 Rules for I/O Address Space Load Instruction Data Merging

Merge Register/ Replayed Instruction	Load Byte/Word	Load Longword	Load Quadword
Byte/Word	No merge	No merge	No merge
Longword	No merge	Merge up to 32 bytes	No merge
Quadword	No merge	No merge	Merge up to 64 bytes

In summary, Table 2–7 shows some of the following rules.

- Byte/word load instructions and different size load instructions are not allowed to merge.
- A stream of ascending non-overlapping, but not necessarily consecutive, longword load instructions are allowed to merge into naturally aligned 32-byte blocks.
- A stream of ascending non-overlapping, but not necessarily consecutive, quadword load instructions are allowed to merge into naturally aligned 64-byte blocks.
- Merging of quadwords can be limited to naturally-aligned 32-byte blocks based on the Cbox WRITE_ONCE chain 32_BYTE_IO field.
- To minimize latency the I/O register merge window is closed when a timer detects no I/O load instruction activity for 14 cycles, or zero cycles if the last QW/LW of the block is addressed.

After the Mbox I/O register has closed its merge window, the Cbox sends I/O read requests offchip in the order that they were received from the Mbox.

2.8.3 Memory Address Space Store Instructions

The Mbox begins execution of a store instruction by translating its virtual address to a physical address using the DTB and by probing the Dcache. The Mbox puts information about the store instruction, including its physical address, its data and the results of the Dcache probe, into the store queue (SQ).

If the Mbox does not find the addressed location in the Dcache, it places the address into the MAF for processing by the Cbox. If the Mbox finds the addressed location in a Dcache block that is not dirty, then it places a ChangeToDirty request into the MAF.

A store instruction can write its data into the Dcache when it is retired, and when the Dcache block containing its address is dirty and not shared. SQ entries that meet these two conditions can be placed into the *writable* state. These SQ entries are placed into the *writable* state in program order at a maximum rate of two entries per cycle. The Mbox transfers *writable* store queue entry data from the SQ to the Dcache in program order at a maximum rate of two entries per cycle. Dcache lines associated with *writable* store queue entries are locked by the Mbox. System port probe commands cannot evict these blocks until their associated writable SQ entries have been transferred into the Dcache. This restriction assists in STx_C instruction and Dcache ECC processing.

SQ entry data that has not been transferred to the Dcache may source data to newer load instructions. The Mbox compares the virtual Dcache index bits of incoming load instructions to queued SQ entries, and sources the data from the SQ, bypassing the Dcache, when necessary.

2.8.4 I/O Address Space Store Instructions

The Mbox begins processing I/O space store instruction, like memory space store instruction, by translating the virtual address and placing the state associated with the store instruction into the SQ.

The Mbox replays retired I/O space store entries from the SQ to the IOWB in program order at a rate of one per GCLK cycle. The Mbox never allows queued I/O space store instructions to source data to subsequent load instructions.

The Cbox maximizes I/O bandwidth when it allocates a new IOWB entry to an I/O store instruction by attempting to merge I/O load instructions in a merge register. Table 2–8 shows the rules for I/O space store instruction data merging. The columns represent the load instructions replayed to the IOWB while the rows represent the size of the store in the merge register.

Table 2–8 Rules for I/O Address Space Store Instruction Data Merging

Merge Register/ Replayed Instruction	Store Byte/Word	Store Longword	Store Quadword
Byte/Word	No merge	No merge	No merge
Longword	No merge	Merge up to 32 bytes	No merge
Quadword	No merge	No merge	Merge up to 64 bytes

Table 2–8 shows some of the following rules:

- Byte/word store instructions and different size store instructions are not allowed to merge.
- A stream of ascending non-overlapping, but not necessarily consecutive, longword store instructions are allowed to merge into naturally aligned 32-byte blocks.
- A stream of ascending non-overlapping, but not necessarily consecutive, quadword store instructions are allowed to merge into naturally aligned 64-byte blocks.
- Merging of quadwords can be limited to naturally-aligned 32-byte blocks based on the Cbox WRITE_ONCE chain 32_BYTE_IO field.
- Issued MB, WMB, and I/O load instructions close the I/O register merge window. To minimize latency, the merge window is also closed when a timer detects no I/O store instruction activity for 1024 cycles.

After the IOWB merge register has closed its merge window, the Cbox sends I/O space store requests offchip in the order that they were received from the Mbox.

2.9 MAF Memory Address Space Merging Rules

Because all memory transactions are to 64-byte blocks, efficiency is improved by merging several small data transactions into a single larger data transaction.
Table 2–9 lists the rules the 21264 uses when merging memory transactions into 64-byte naturally aligned data block transactions. Rows represent the merged instruction in the MAF and columns represent the new issued transaction.

Table 2–9 MAF Merging Rules

MAF/New	LDx	STx	STx_C	WH64	ECB	Istream
LDx	Merge	—	—	—	—	—
STx	Merge	Merge	—	—	—	—
STx_C	—	—	Merge	—	—	—
WH64	—	—	—	Merge	—	—
ECB	—	—	—	—	Merge	—
Istream	—	—	—	—	—	Merge

In summary, Table 2–9 shows that only like instruction types, with the exception of load instructions merging with store instructions, are merged.

2.10 Instruction Ordering

In the absence of explicit instruction ordering, such as with MB or WMB instructions, the 21264 maintains a default instruction ordering relationship between pairs of load and store instructions.

The 21264 maintains the default memory data instruction ordering as shown in Table 2–10 (assume address X and address Y are different).

Table 2–10 Memory Reference Ordering

First Instruction in Pair	Second Instruction in Pair	Reference Order
Load memory to address X	Load memory to address X	Maintained (litmus test 1)
Load memory to address X	Load memory to address Y	Not maintained
Store memory to address X	Store memory to address X	Maintained
Store memory to address X	Store memory to address Y	Maintained
Load memory to address X	Store memory to address X	Maintained
Load memory to address X	Store memory to address Y	Not maintained
Store memory to address X	Load memory to address X	Maintained
Store memory to address X	Load memory to address Y	Not maintained

The 21264 maintains the default I/O instruction ordering as shown in Table 2–11 (assume address X and address Y are different).

Table 2–11 I/O Reference Ordering

First Instruction in Pair	Second Instruction in Pair	Reference Order
Load I/O to address X	Load I/O to address X	Maintained
Load I/O to address X	Load I/O to address Y	Maintained
Store I/O to address X	Store I/O to address X	Maintained
Store I/O to address X	Store I/O to address Y	Maintained
Load I/O to address X	Store I/O to address X	Maintained
Load I/O to address X	Store I/O to address Y	Not maintained
Store I/O to address X	Load I/O to address X	Maintained
Store I/O to address X	Load I/O to address Y	Not maintained

2.11 Replay Traps

There are some situations in which a load or store instruction cannot be executed due to a condition that occurs after that instruction issues from the IQ or FQ. The instruction is aborted (along with all newer instructions) and restarted from the fetch stage of the pipeline. This mechanism is called a replay trap.

2.11.1 Mbox Order Traps

Load and store instructions may be issued from the IQ in a different order than they were fetched from the Icache, while the architecture dictates that Dstream memory transactions to the same physical bytes must be completed in order. Usually, the Mbox manages the memory reference stream by itself to achieve architecturally correct behavior, but the two cases in which the Mbox uses replay traps to manage the memory stream are *load-load* and *store-load* order traps.

2.11.1.1 Load-Load Order Trap

The Mbox ensures that load instructions that read the same physical byte(s) ultimately issue in correct order by using the *load-load* order trap. The Mbox compares the address of each load instruction, as it is issued, to the address of all load instructions in the load queue. If the Mbox finds a newer load instruction in the load queue, it invokes a *load-load* order trap on the newer instruction. This is a replay trap that aborts the target of the trap and all newer instructions from the machine and refetches instructions starting at the target of the trap.

2.11.1.2 Store-Load Order Trap

The Mbox ensures that a load instruction ultimately issues after an older store instruction that writes some portion of its memory operand by using the *store-load* order trap. The Mbox compares the address of each store instruction, as it is issued, to the address of all load instructions in the load queue. If the Mbox finds a newer load instruction in the load queue, it invokes a *store-load* order trap on the load instruction. This is a replay trap. It functions like the *load-load* order trap.

The Ibox contains extra hardware to reduce the frequency of the *store-load* trap. There is a 1-bit by 1024-entry VPC-indexed table in the Ibox called the stWait table. When an Icache instruction is fetched, the associated stWait table entry is fetched along with the Icache instruction. The stWait table produces 1 bit for each instruction accessed from the Icache. When a load instruction gets a *store-load* order replay trap, its associated bit in the stWait table is set during the cycle that the load is refetched. Hence, the trapping load instruction's stWait bit will be set the next time it is fetched.

The IQ will not issue load instructions whose stWait bit is set while there are older unissued store instructions in the queue. A load instruction whose stWait bit is set can be issued the cycle immediately after the last older store instruction is issued from the queue. All the bits in the stWait table are unconditionally cleared every 16384 cycles.

2.11.2 Other Mbox Replay Traps

The Mbox also uses replay traps to control the flow of the load queue and store queue, and to ensure that there are never multiple outstanding misses to different physical addresses that map to the same Dcache or Bcache line. Unlike the order traps, however, these replay traps are invoked on the incoming instruction that triggered the condition.

2.12 I/O Write Buffer and the WMB Instruction

The I/O write buffer (IOWB) consists of four 64-byte entries with the associated address and control logic used to buffer I/O write data between the store queue (SQ) and the system port.

2.12.1 Memory Barrier (MB/WMB/TB Fill Flow)

The Cbox CSR SYSBUS_MB_ENABLE bit determines if MB instructions produce external system port transactions. When the SYSBUS_MB_ENABLE bit equals 0, the Cbox CSR MB_CNT[3:0] field contains the number of pending uncommitted transactions. The counter will increment for each of the following commands:

- RdBlk, RdBlkMod, RdBlkI
- RdBlkSpec (valid), RdBlkModSpec (valid), RdBlkSpecI (valid)

- RdBlkVic, RdBlkModVic, RdBlkVicI
- CleanToDirty, SharedToDirty, STChangeToDirty, InvalToDirty
- FetchBlk, FetchBlkSpec (valid), Evict
- RdByte, RdLw, RdQw, WrByte, WrLW, WrQW

The counter is decremented with the C (commit) bit in the Probe and SysDc commands. Systems can assert the C bit in the SysDc fill response to the commands that originally incremented the counter, or attached to the last probe seen by that command when it reached the system serialization point. If the number of uncommitted transactions reaches 15 (saturating the counter), the Cbox will stall MAF and IOWB processing until at least one of the pending transactions has been committed. Probe processing is not interrupted by the state of this counter.

2.12.1.1 MB Instruction Processing

When an MB instruction is fetched in the predicted instruction execution path, it stalls in the map stage of the pipeline. This also stalls all instructions after the MB, and control of instruction flow is based upon the value in Cbox CSR SYSBUS_MB_ENABLE as follows:

- If Cbox CSR SYSBUS_MB_ENABLE is clear, the Cbox waits until the IQ is empty and then performs the following actions:

  a. Sends all pending MAF and IOWB entries to the system port.

  b. Monitors Cbox CSR MB_CNT[3:0], a 4-bit counter of outstanding committed events. When the counter decrements from one to zero, the Cbox marks the youngest probe queue entry.

  c. Waits until the MAF contains no more Dstream references and the SQ, LQ, and IOWB are empty.

  When all of the above have occurred and a probe response has been sent to the system for the marked probe queue entry, instruction execution continues with the instruction after the MB.

- If Cbox CSR SYSBUS_MB_ENABLE is set, the Cbox waits until the IQ is empty and then performs the following actions:

  a. Sends all pending MAF and IOWB entries to the system port

  b. Sends the MB command to the system port

  c. Waits until the MB command is acknowledged, then marks the youngest entry in the probe queue

  d. Waits until the MAF contains no more Dstream references and the SQ, LQ, and IOWB are empty

  When all of the above have occurred and a probe response has been sent to the system for the marked probe queue entry, instruction execution continues with the instruction after the MB.

  Because the MB instruction is executed speculatively, MB processing can begin and the original MB can be killed. In the internal acknowledge case, the MB may have already been sent to the system interface, and the system is still expected to respond to the MB.

2.12.1.2 WMB Instruction Processing

Write memory barrier (WMB) instructions are issued into the Mbox store-queue, where they wait until they are retired and all prior store instructions become writable. The Mbox then stalls the writable pointer and informs the Cbox. The Cbox closes the IOWB merge register and responds in one of the following two ways:

- If Cbox CSR SYSBUS_MB_ENABLE is clear, the Cbox performs the following actions:
  a. Stalls further MAF and IOWB processing.
  b. Monitors Cbox CSR MB_CNT[3:0], a 4-bit counter of outstanding committed events. When the counter decrements from one to zero, the Cbox marks the youngest probe queue entry.
  c. When a probe response has been sent to the system for the marked probe queue entry, the Cbox considers the WMB to be satisfied.
- If Cbox CSR SYSBUS_MB_ENABLE is set, the Cbox performs the following actions:
  a. Stalls further MAF and IOWB processing.
  b. Sends the MB command to the system port.
  c. Waits until the MB command is acknowledged by the system with a SysDc MBDone command, then sends acknowledge and marks the youngest entry in the probe queue.
  d. When a probe response has been sent to the system for the marked probe queue entry, the Cbox considers the WMB to be satisfied.

2.12.1.3 TB Fill Flow

Load instructions (HW_LDs) to a virtual page table entry (VPTE) are processed by the 21264 to avoid litmus test problems associated with the ordering of memory transactions from another processor against loading of a page table entry and the subsequent virtual-mode load from this processor.

Consider the sequence shown in Table 2–12. The data could be in the Bcache. Pj should fetch datai if it is using PTEi.

Table 2–12 TB Fill Flow Example Sequence 1

Pi	Pj
Write Datai	Load/Store datai
MB	<TB miss>
Write PTEi	Load-PTE <write TB> Load/Store (restart)

Also consider the related sequence shown in Table 2–13. In this case, the data could be cached in the Bcache; Pj should fetch datai if it is using PTEi.

Table 2–13 TB Fill Flow Example Sequence 2

Pi	Pj
Write Datai	Istream read datai
MB	<TB miss>
Write PTEi	Load-PTE <write TB> Istream read (restart) - will miss the Icache

The 21264 processes Dstream loads to the PTE by injecting, in hardware, some memory barrier processing between the PTE transaction and any subsequent load or store instruction. This is accomplished by the following mechanism:

1. The integer queue issues a HW_LD instruction with VPTE.
2. The integer queue issues a HW_MTPR instruction with a DTB_PTE0, that is data-dependent on the HW_LD instruction with a VPTE, and is required in order to fill the DTBs. The HW_MTPR instruction, when queued, sets IPR scoreboard bits [4] and [0].
3. When a HW_MTPR instruction with a DTB_PTE0 is issued, the Ibox signals the Cbox indicating that a HW_LD instruction with a VPTE has been processed. This causes the Cbox to begin processing the MB instruction. The Ibox prevents any subsequent memory operations being issued by not clearing the IPR scoreboard bit [0]. IPR scoreboard bit [0] is one of the scoreboard bits associated with the HW_MTPR instruction with DTB_PTE0.
4. When the Cbox completes processing the MB instruction (using one of the above sequences, depending upon the state of SYSBUS_MB_ENABLE), the Cbox signals the Ibox to clear IPR scoreboard bit [0].

The 21264 uses a similar mechanism to process Istream TB misses and fills to the PTE for the Istream.

1. The integer queue issues a HW_LD instruction with VPTE.
2. The IQ issues a HW_MTPR instruction with an ITB_PTE that is data-dependent upon the HW_LD instruction with VPTE. This is required in order to fill the ITB. The HW_MTPR instruction, when queued, sets IPR scoreboard bits [4] and [0].
3. The Cbox issues a HW_MTPR instruction for the ITB_PTE and signals the Ibox that a HW_LD/VPTE instruction has been processed, causing the Cbox to start processing the MB instruction. The Mbox stalls Ibox fetching from when the HW_LD/VPTE instruction finishes until the probe queue is drained.
4. When the 21264 is finished (SYS_MB selects one of the above sequences), the Cbox directs the Ibox to clear IPR scoreboard bit [0]. Also, the Mbox directs the Ibox to start prefetching.

Inserting MB instruction processing within the TB fill flow is only required for multiprocessor systems. Uniprocessor systems can disable MB instruction processing by deasserting Ibox CSR I_CTL[TB_MB_EN].

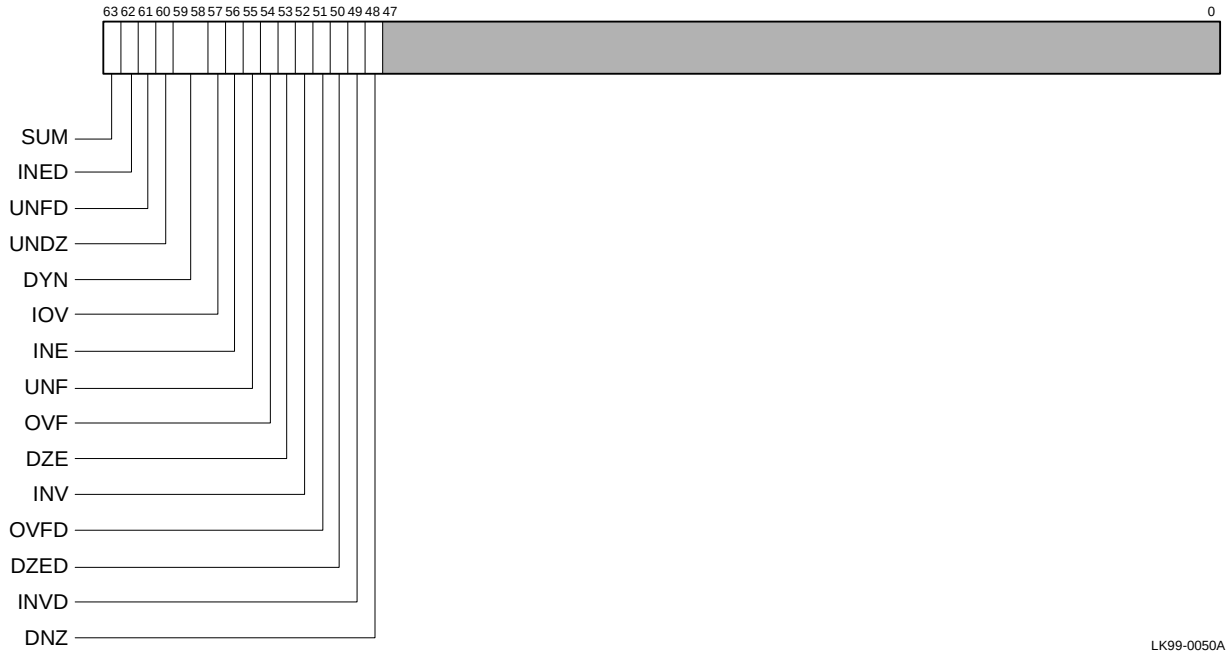
2.13 Performance Measurement Support—Performance Counters

The 21264 provides hardware support for obtaining program performance feedback information without requiring program modification.

2.14 Floating-Point Control Register

The floating-point control register (FPCR) is shown in Figure 2–11.

Figure 2–11 Floating-Point Control Register



The floating-point control register fields are described in Table 2–14.

Table 2–14 Floating-Point Control Register Fields

Name	Extent	Type	Description
SUM	[63]	RW	Summary bit. Records bit-wise OR of FPCR exception bits.
INED	[62]	RW	Inexact Disable. If this bit is set and a floating-point instruction that enables trapping on inexact results generates an inexact value, the result is placed in the destination register and the trap is suppressed.
UNFD	[61]	RW	Underflow Disable. The 21264 hardware cannot generate IEEE compliant denormal results. UNFD is used in conjunction with UNDZ as follows:

UNFD	UNDZ	Result
0	X	Underflow trap.
1	0	Trap to supply a possible denormal result.
1	1	Underflow trap suppressed. Destination is written with a true zero (+0.0).

Table 2–14 Floating-Point Control Register Fields (Continued)

Name	Extent	Type	Description
UNDZ	[60]	RW	Underflow to zero. When UNDZ is set together with UNFD, underflow traps are disabled and the 21264 places a true zero in the destination register. See UNFD, above.
DYN	[59:58]	RW	Dynamic rounding mode. Indicates the rounding mode to be used by an IEEE floating-point instruction when the instruction specifies dynamic rounding mode:
IOV	[57]	RW	Integer overflow. An integer arithmetic operation or a conversion from floating-point to integer overflowed the destination precision.
INE	[56]	RW	Inexact result. A floating-point arithmetic or conversion operation gave a result that differed from the mathematically exact result.
UNF	[55]	RW	Underflow. A floating-point arithmetic or conversion operation gave a result that underflowed the destination exponent.
OVF	[54]	RW	Overflow. A floating-point arithmetic or conversion operation gave a result that overflowed the destination exponent.
DZE	[53]	RW	Divide by zero. An attempt was made to perform a floating-point divide with a divisor of zero.
INV	[52]	RW	Invalid operation. An attempt was made to perform a floating-point arithmetic operation and one or more of its operand values were illegal.
OVFD	[51]	RW	Overflow disable. If this bit is set and a floating-point arithmetic operation generates an overflow condition, then the appropriate IEEE nontrapping result is placed in the destination register and the trap is suppressed.
DZED	[50]	RW	Division by zero disable. If this bit is set and a floating-point divide by zero is detected, the appropriate IEEE nontrapping result is placed in the destination register and the trap is suppressed.
INVD	[49]	RW	Invalid operation disable. If this bit is set and a floating-point operate generates an invalid operation condition and 21264 is capable of producing the correct IEEE nontrapping result, that result is placed in the destination register and the trap is suppressed.
DNZ	[48]	RW	Denormal operands to zero. If this bit is set, treat all Denormal operands as a signed zero value with the same sign as the Denormal operand.
Reserved	[47:0][1]	—	—

Bits	Meaning
00	Chopped
01	Minus infinity
10	Normal
11	Plus infinity

[1] Alpha architecture FPCR bit 47 (DNOD) is not implemented by the 21264.

2.15 AMASK and IMPLVER Values

The AMASK and IMPLVER instructions return processor type and supported architecture extensions, respectively.

2.15.1 AMASK

The 21264 returns the AMASK instruction values provided in Table 2–15.

Table 2–15 21264 AMASK Values

21264 Pass Level	AMASK Value Returned
Pass 1	001_{16}
Pass 2	303_{16}

The AMASK bit definitions provided in Table 2–15 are defined in Table 2–16.

Table 2–16 AMASK Bit Assignments

Bit	Meaning
0	Support for the byte/word extension (BWX) The instructions that comprise the BWX extension are LDBU, LDWU, SEXTB, SEXTW, STB, and STW.
1	Support for the square-root and floating-point convert extension (FIX) The instructions that comprise the FIX extension are FTOIS, FTOIT, ITOFF, ITOFS, ITOFT, SQRTF, SQRTG, SQRTS, and SQRTT.
8	Support for the multimedia extension (MVI) The instructions that comprise the MVI extension are MAXSB8, MAXSW4, MAXUB8, MAXUW4, MINSB8, MINSW4, MINUB8, MINUW4, PERR, PKLB, PKWB, UNPKBL, and UNPKBW.
9	Support for precise arithmetic trap reporting in hardware. The trap PC is the same as the instruction PC after the trapping instruction is executed.

2.15.2 IMPLVER

For the 21264, the IMPLVER instruction returns the value 2.

2.16 Design Examples

The 21264 can be designed into many different uniprocessor and multiprocessor system configurations. Figures 2–12 and 2–13 illustrate two possible configurations. These configurations employ additional system/memory controller chipsets.

Figure 2–12 shows a typical uniprocessor system with a second-level cache. This system configuration could be used in standalone or networked workstations.

Figure 2–12 Typical Uniprocessor Configuration

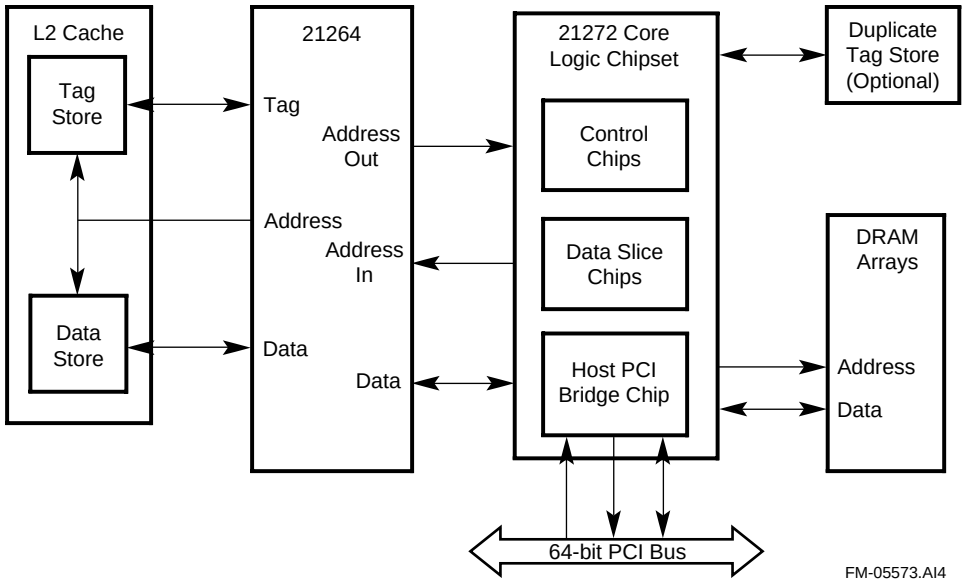
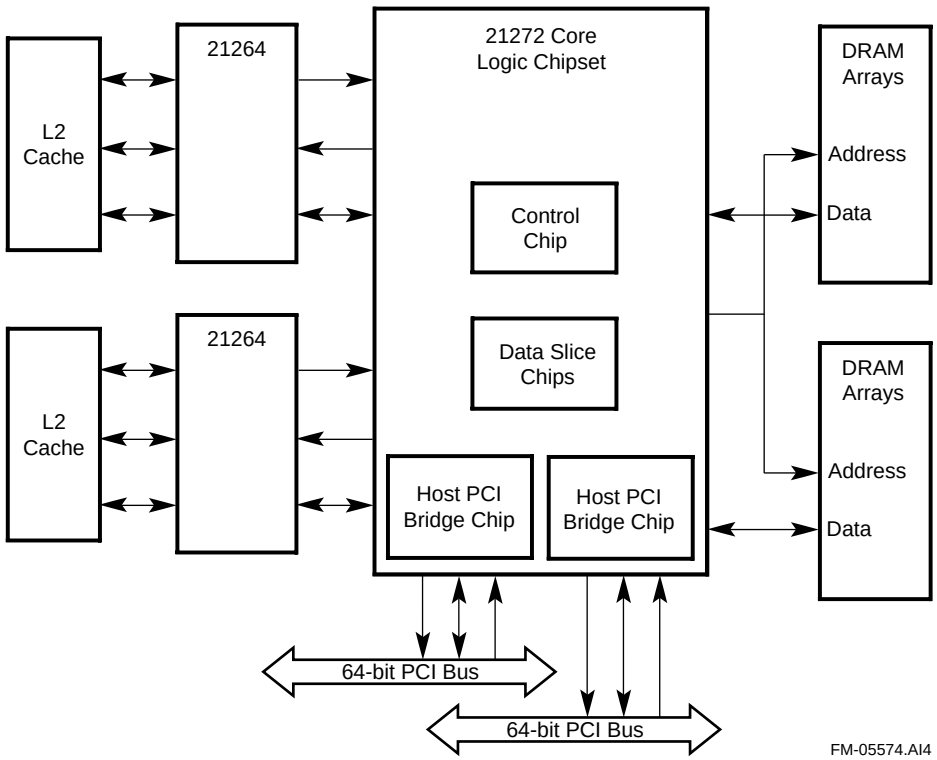


Figure 2–13 shows a typical multiprocessor system, each processor with a second-level cache. Each interface controller must employ a duplicate tag store to maintain cache coherency. This system configuration could be used in a networked database server application.

Figure 2–13 Typical Multiprocessor Configuration



3
Guidelines for Compiler Writers

This chapter is a supplement to Appendix A of the *Alpha Architecture Handbook, Version 4*. That appendix presents general guidelines for software that are independent of the processor implementation. This chapter identifies some of the specific features of the 21264 that affect performance and that can be controlled by a compiler writer or assembly-language programmer. Chapter 2 describes the specific hardware features of the 21264 for which this chapter provides programming guidelines.

3.1 Architecture Extensions

The 21264 provides four extensions to the Alpha architecture, consisting of three instruction extensions and precise exception reporting for arithmetic operations. The three instruction extensions are:

- BWX for byte/word operations
- FIX for floating-point operations
- MVI for multimedia

Use the AMASK instruction (see Section 2.15) to test for the presence of these extensions. Using AMASK makes it possible to generate efficient code that uses the extensions, while still running correctly on implementations that do not contain them. See the *Alpha Architecture Handbook, Version 4* for more details.

The 21264 also has new instructions for memory prefetch, described in Section 3.6.

3.2 Instruction Alignment

Where possible, branch targets should be octaword (16-byte) aligned. Use NOP instructions to pad code for alignment. The 21264 discards NOP instructions early in the pipeline, so the main costs are space in the instruction cache and instruction fetch bandwidth. Appendix A of the *Alpha Architecture Handbook, Version 4*, defines three stylized NOPs: a universal NOP, an integer NOP, and a floating-point NOP:

```
UNOP    ==   LDQ_U R31, 0(Rx)
NOP     ==   BIS   R31, R31, R31
FNOP    ==   CPYS  F31, F31, F31
```

From the standpoint of instruction scheduling, the 21264 treats all three NOP forms identically. See Section 2.5.

Always align routine beginnings and branch targets that are preceded in program order by:

- Computed jumps
- Unconditional branches
- Return instructions

Always align targets of computed jumps (JMP and JSR), even if there is a fall-through path to the target.

Although not generally recommended, it may be beneficial to align branch targets that can also be reached by a fall through.

3.3 Data Alignment

As in previous implementations, references to unaligned data continue to trap and are completed in software. Programmers are encouraged to align their data on natural boundaries. When data cannot be aligned, use the nontrapping sequences listed in the *Alpha Architecture Handbook, Version 4.*

Because the 21264 implements the BWX extension, it is beneficial to do unaligned word (16-bit) operations with two byte operations. For example, the following sequence loads an unsigned, unaligned word:

```
LDBU    T3,    1(T0)
LDBU    T2,    (T0)
SLL     T3,    8,     T3
BIS     T2,    T3,    V0
```

3.4 Control Flow

As in previous implementations, the compiler should lay out code so that fall through is the common path. For the 21264, the line predictor is initialized to favor a fall-through path. Furthermore, on a demand miss, the next three lines are prefetched into the instruction cache. However, code layout does not affect branch prediction. Branch prediction on the 21264 is fully dynamic; the direction of the branch does not affect the prediction.

3.4.1 Need for Single Successors

Code should be arranged so that each aligned octaword (group of four instructions) has at most one likely successor, because each of the following predictors stores only one prediction for each octaword:

- The line predictor
- The JMP/JSR predictor (which uses the line predictor)
- Parts of the branch predictor

To ensure that there is only one successor, include at most one change of control flow instruction in each octaword. BSR and JSR instructions should be the last instruction in the octaword, so that the octaword does not have both the call target and the fall-through octaword as successors. If an octaword has a JMP or JSR, there should not be another control flow instruction, CMOV, LDx_L, STx_C, WMB, MB, RS, RC, or

RPCC instruction; these instructions prevent the line predictor from training. If the compiler puts multiple rarely taken conditional branches in the same octaword, there will not be a problem with aliasing in the line predictor or the branch predictor.

3.4.2 Branch Prediction

The branch predictor in the 21264 is sophisticated and can predict branch behavior where the behavior depends on past history of the same branch or previous branches. For example, branches are predicted that tend to go in the same direction or that have patterns. However, the following instructions interfere with the branch predictor and cause it to predict fall through when placed in the same octaword as conditional branch instructions: LDx_L, STx_C, WMB, MB, RS, RC, and RPCC.

Branches that cannot be predicted are costly, so try to use the conditional move instruction (CMOV) or logical operations to eliminate branch instructions. If a conditional branch guards a few instructions, it is almost always beneficial to eliminate the branch. For larger blocks of code, the benefit depends on whether the branch is predictable.

3.4.3 Filling Instruction Queues

Normally, the 21264 can fetch one aligned octaword per cycle and fill the instruction queues. There are some situations where it fetches less, which can reduce performance if the 21264 is removing instructions from the queues (issuing them) faster than they can be filled. The 21264 can predict at most one branch per cycle; if an aligned octaword contains *n* branches, it takes *n* cycles to fetch the entire aligned octaword. Thus, there can be a penalty for placing more than one branch in an octaword, even if all the branches are rarely taken. However, spacing out branches by padding the octaword with NOPs does not speed up the fetch. This is usually only a problem for code with very high ILP (instruction-level parallelism), where instruction fetch cannot keep up with execution.

3.4.4 Branch Elimination

Removing branches eliminates potential branch mispredicts, improves instruction fetch, and removes barriers to optimization in the compiler. Many branches can be removed by using the CMOV instruction or logical instructions. The following sections describe some techniques for eliminating branches that are specific to the Alpha instruction set.

3.4.4.1 Example of Branch Elimination with CMOV

The C code in the following example can be implemented without branches by using the CMOV instruction.

In the example, the variable *d* is assigned on both paths, so it is replaced with an unconditional assignment — the value from one path followed by a CMOV to conditionally overwrite it. The variable *c* is not live out of the conditional, so its assignment can also be done unconditionally. To conditionalize the store (*p=a), a dummy location called the bit bucket (*BB*) is created on the stack, and the address register for the store is overwritten with the bit bucket address to prevent the store from occurring when the condition is false.

The C code:

```
if (a < b) {
        c = a + b;
        d = c + 1;
        *p = a;
}
else {
        d = 2;
}
```

Implementation using the CMOV instruction:

```
CMPLT   A,      B,      R0
ADDL    A,      B,      C
ADDL    C,      1,      R1
MOV     2,      D
CMOVNE  R0,     R1,     D
CMOVEQ  R0,     BB,     P
STL     A,      (P)
```

3.4.4.2 Replacing Conditional Moves with Logical Instructions

If an octaword contains *n* CMOV instructions, it takes *n*+1 cycles to put that aligned octaword into the instruction queues. This is only a problem for code with very high ILP. When executing, the CMOV instruction is treated like two dependent instructions. If possible, it is usually a good idea to replace a CMOV instruction with one or two logical instructions. Integer compare instructions produce a value of zero or one. By subtracting one from the result of a compare, the values are all zeroes or all ones, which makes a convenient mask in evaluating conditional expressions. For example:

```
if (a < b) c = 0
```

could be implemented with:

```
CMPLT   A,      B,      R0
CMOVNE  R0,     R31,    C
```

But a better sequence that consumes the same amount of execution resources but less fetch resources is:

```
CMPLT   A,      B,      R0
SUBQ    R0,     1,      R0
AND     R0,     C,      C
```

3.4.4.3 Combining Branches

Multiple dependent branches can often be combined into a single branch. For example, the C expression `(a < b && c < d)` can be computed with:

```
CMPLT   A,      B,      R1
BEQ     R1,     L1
CMPLT   C,      D,      R1
BEQ     R1,     L1
```

or equivalently as:

```
CMPLT   A,      B,      R1
CMPLT   C,      D,      R2
AND     R1,     R2,     R2
BEQ     R2,     L1
```

Combining the two branches into one branch avoids the problems caused by multiple branches in the same aligned octaword. Of even greater benefit, the combined branch is usually more predictable than the two original branches.

3.4.5 Computed Jumps and Returns

The targets of computed jumps (JMP and JSR instructions) are predicted differently than PC-relative branches and require special attention. The first time a JMP or JSR instruction is brought into the cache, the target is computed by using the predicted target field contained in the jump instruction to compute an index into the cache, combined with the tag currently contained in that index. If that prediction is wrong, the processor quickly switches to another prediction mode that uses the line predictor for future occurrences of that jump.

Because the line predictor predicts aligned octawords and not individual instructions, it always predicts the beginning of an aligned octaword even if the target is not the first instruction. Thus, it is important to align targets of computed jumps. Note that even if the predicted target field is correct in the JMP instruction, it still mispredicts if the target is not in the cache because the tag is wrong. Therefore, the compiler should both set the hint field bit and align the jump target so that line prediction will work.

The target of a RET instruction is predicted with a return stack.

- BSR and JSR instructions push the return address on the prediction stack.
- RET instructions pop the current stack entry as prediction for the return address.
- JSR_COROUTINE instructions both pop and push the stack.

Calls and returns must be nested and appropriately paired for the stack to generate correct predictions. The stack holds 32 predictions and is actually implemented as a linked list on top of a circular buffer. Each push to the stack allocates a new entry in the buffer and the prediction for a given call is overwritten and lost if 32 or more calls are fetched before the matching return.

3.5 SIMD Parallelism

Programs can do SIMD-style (single instruction stream, multiple data stream) parallelism in registers. SIMD parallelism can greatly reduce the number of instructions executed. The MVI instructions support SIMD parallelism and some non-MVI instructions are also useful. A simple example is implementing a byte-at-a-time copy loop with quadword copies. Another example is testing for a nonzero byte in an array of eight bytes with a single quadword load — a BNE instruction can determine if any byte is nonzero; a CMPBGE instruction can determine which byte is nonzero. See Appendix B for an example of SIMD parallelism.

3.6 Prefetching

Prefetching is very important (by a factor of 2) for loops dominated by memory latency or bandwidth. The 21264 has three styles of prefetch:

- Prefetch for loading data that is expected to be read only (normal prefetch). Reduces the latency to read memory.
- Prefetch for data that will probably be written (prefetch with modify intent). Reduces the latency to read memory and bus traffic.
- A special WH64 instruction to execute if the program intends to write an entire aligned block of 64 bytes (write hint 64). Reduces the amount of memory bandwidth required to write a block of data.

The prefetch instructions and write hints are recognized as prefetches or ignored on pre-21264 implementations, so it is always safe for a compiler to use them. The load prefetches have no architecturally visible effect, so inserting prefetches never causes a program error. Because of its more powerful memory system, prefetches on a 21264 have more potential benefit than previous Alpha implementations and unnecessary prefetching is less costly. See Section 2.6 for a description of the instructions.

Always prefetch ahead at least two cache blocks. Prefetch farther ahead if possible, unless doing so requires more than eight offchip references to be in progress at the same time. That is, for a loop that references *n* streams, prefetch ahead two blocks or 8/*n* blocks, whichever is greater. Note, however, that for short trip count loops, it may be beneficial to reduce the prefetch distance, so that the prefetched data is likely to be used.

Prefetches to invalid addresses are dismissed by PALcode, so it is safe to prefetch off the end of an array, but it does incur a small (less than 30 cycle) performance penalty. Prefetches can have alignment traps, so align the pointer used to prefetch.

The WH64 instruction sets an aligned 64-byte block to an unknown state. Use WH64 when the program intends to completely write an aligned 64-byte area of memory. Unlike load prefetches, the WH64 instruction modifies data, and it is not safe to execute WH64 off the end of an array. Although a conditional branch can guard the WH64 instruction so that it does not go beyond the end of an array, a better solution is to create a dummy aligned block of 64 bytes of memory (the bit bucket) on the stack and use a CMOV instruction to select the bitbucket address when nearing the end of the array. For example:

```
CMPLT   R0,     R1,     R2      # test if there are at least 64 bytes left
CMOVEQ  R2,     R3,     R4      # if not, overwrite R4 with address
                                #    of the bit bucket
WH64    R4
```

3.7 Avoiding Replay Traps

The 21264 can have several memory operations in progress at the same time, rather than waiting for one memory operation to complete before starting another. The 21264 can reorder memory operations if one operation is delayed because its input operands are not data ready or because of system dynamics.

There are some situations where the execution of a memory operation must be aborted, together with all newer instructions in progress. When the situation is corrected, the instruction is refetched and execution continues. This is called a replay trap and is described in Section 2.11.

A replay trap is a hardware mechanism for aborting speculative work and is not the same as a software exception or trap. Typically, the main cost of a replay trap is that the processor must wait for the condition that caused the trap (such as a cache miss or a store queue drain) to clear before executing any instructions after the trapping instruction. In addition, instructions must be restarted in the pipeline, which adds to the penalty listed in Table 2–1. The actual effect on performance depends on the length of the stall and how much the processor can overlap the stall with other work, such as restarting the pipeline.

Replay traps occur when there are multiple concurrent loads and/or stores in progress to the same address or same cache index. The farther apart the loads and/or stores are in the instruction stream, the less likely they will be active at the same time. It is impossible to predict exactly how much distance is needed, but 40 instructions should be safe if the data is in the level 2 cache. The best way to avoid replay traps is to keep values in registers so that multiple references to the same address are not in progress at the same time.

Generally, there are three causes for multiple loads and stores to the same address. The following lists these causes and suggests remedies:

- High register pressure causes repeated spills and reloads of variables. Profile information is especially useful to ensure that frequently referenced values are kept in registers.
- Memory aliasing prevents the compiler from keeping values in registers. Pointer and interprocedural analysis are important techniques for eliminating unnecessary memory references.
- Reuse of stack location for temporaries leads to repeated references to the stack address. Immediate reuse of stack locations is discouraged because it creates a dependence through memory that the 21264 is unable to remove.

Section 2.11 describes the general concept of replay traps and provides some examples. The following sections describe the replay traps that have been found to occur frequently and contains specific recommendations for avoiding them.

3.7.1 Store-Load Order Replay Trap

Stores go into the store queue, and loads to the same address can get the data from the store queue. Operations tend to be executed in program order, unless an operation is not data ready. However, if the processor reorders the instructions so that a newer load to the same address executes before the store, a replay trap occurs and execution restarts at the load. This is called a store-load order replay trap. If this happens frequently enough, the processor will learn to delay issuing the load until all previous stores have completed. Delaying the load can decrease performance because it must wait for all stores, rather than just stores to the same address. However, the delay is usually faster than replay trapping.

The new instructions, FTOI*x* and ITOF*x*, transfer data between the floating-point and integer register files. Since they avoid situations where data is stored and immediately loaded back, they avoid store-load order replay traps and should be used wherever possible.

3.7.2 Wrong-Size Replay Trap

If there is a store followed by a load that reads the same data, and the load data type is larger than the store, then the load must get some of the data from the store queue and the rest from the cache. The processor replay traps until the store queue drains into the Dcache and then gets all the data from the cache. This is called a wrong-size replay trap. Unlike the store-load order replay trap, the wrong-size replay trap occurs even if the store and load execute in order. The trap can take over 20 cycles and can be avoided by widening the store, narrowing the load, or eliminating the load and getting the value from a register. If the store data is larger than the load data, a wrong-size replay trap does not occur.

3.7.3 Load-Load Order Replay Trap

If two loads to the same address issue out of order, the processor will replay trap and execute starting from the newer load. Unlike the store-load order replay trap, there is no mechanism that learns to delay the newer load until the first has issued. However, instructions tend to issue in program order, unless they are not data ready. See Appendix C for a code sequence that can enforce an ordering between loads.

3.7.4 Load-Miss Load Replay Trap

If there is a load followed by another load to the same address and the first load misses, then the processor replay traps until the data is loaded into the register for the first load. This is called a load-miss load replay trap. This trap occurs even if the loads are issued in program order, so long as the first load is waiting for a Dcache miss when the second load issues. See Appendix C.

3.7.5 Mapping to the Same Cache Line

Loads and stores that are in progress at the same time and map to the same cache line (32KB apart) can replay trap. This is similar to the problem that direct-mapped caches have, the difference being that the 21264 cache can hold two data items that map to the same index, but can have only one memory operation in progress at a time that maps to any one cache index.

If possible, avoid loops where a single iteration or nearby interations touch data that is 32KB apart. Avoid creating arrays with dimensions that are multiples of 32KB; pad the array with extra cache blocks if necessary. Also note that prefetches can cause these traps and out-of-order execution can cause multiple iterations of a loop to overlap in execution, so when padding or spacing data references apart, one must consider factors such as the prefetch distance and store delay in computing a safe distance.

3.7.6 Store Queue Overflow

Each store instruction is buffered in the store queue until it retires, up to a maximum of 32 stores. If the store queue overflows, the processor replay traps. To avoid overflow, avoid code with a burst of more than 32 stores and do not expect the processor to sustain more than one store per cycle.

3.8 Scheduling

The 21264 can rearrange instruction execution order to achieve maximum throughput. However, it has limited resources: instruction queue slots and physical registers. The closer the compiler's static schedule is to the actual desired issue order, the less likely the processor will run out of resources and stall. Therefore, it is still beneficial to schedule the code as if the 21264 were an in-order microprocessor, such as the 21164. Software pipelining is also beneficial for loops.

The basic model is a processor that can execute 4 aligned instructions per cycle. Schedule for the resources as described in Table 2–2 and the latencies in Table 2–4 and assume a cross-cluster delay will occur. When determining load latency, assume that scalar references are Dcache hits and array and pointer references are not. Load latencies in Table 2–4 are best case, so schedule for longer latencies if register pressure is not high. Prefetch data where possible and assume the actual load is a Dcache hit.

To reduce Dcache bus traffic, loads should be grouped with loads, stores with stores, two per cycle. Memory operations to different parts of the same cache block can combine together. Group operations with different offsets off the same pointer where possible. Also, do operations in memory address order (such as a bunch of stack saves) where possible.

3.9 Detailed Modeling of the Pipeline

Section 3.8 describes a simple model of the pipeline for use by a compiler. More detailed models must take into account physical register allocation and Ebox slotting and clustering. For most programs, the increase in speed from getting these details right is minimal because the out-of-order issue mechanisms of the 21264 can smooth-out minor scheduling mistakes without an effect on performance. Also, it is often difficult for the compiler to accurately model the physical register usage and Ebox clustering because the compiler cannot easily predict the order in which instructions will be executed. However, for tightly scheduled loops, such as the example in Appendix B, it is possible to produce schedules that achieve higher performance by more accurately modeling the 21264. This section describes such a model.

3.9.1 Physical Registers

Physical registers are a resource that the compiler must manage to achieve optimal performance. As described in Section 2.1.1.5, architectural registers are renamed to physical registers. A physical register is allocated when an instruction is placed in the instruction queue and a physical register is released when the instruction is retired; the physical register that is released is the prior mapping of the destination register. A distinct physical register is required to hold the result of each instruction that has not yet

retired; instructions that do not write a register (such as stores, conditional branches, prefetches, and other instructions that target R31 or F31) do not allocate a physical register.

Table 3–1 presents the minimum latency between an instruction allocating a physical register and the instruction releasing the physical register. That latency is divided into the latency from the map stage to the retire stage and an additional latency from the retire stage until the physical register is actually released. Note that instructions retire in order — a delay in the retire of one instruction delays the retire and the release of physical registers for all subsequent instructions. Table 3–1 is an approximation and ignores several special cases and edge conditions in the register mapper.

Table 3–1 Minimum Latencies From Map to Release of a Physical Register

Instruction Class	Map-to-Retire	Retire-to-Release	Map-to-Release
Integer conditional branch	5	2[1]	7[1]
Integer multiply	5/11[2]	2	7/13[2]
Integer operate	5	2	7
Integer load	8	2	10
Integer store	8	2[1]	10[1]
Floating-point load	8	2	10
Floating-point store	12	4[1]	16[1]
Floating-point add	9	4	13
Floating-point multiply	9	4	13
Floating-point divide/square root	9+latency[3]	4	13+latency[3]
Floating-point conditional branch	9	4	13
BSR/JSR	8	2	10

[1] Conditional branches and stores do not release physical registers. However, their retire point delays the release of registers from subsequent instructions.

[2] Without/with the /V qualifier.

[3] See Table 2–5 and Section 2.4.1.

3.9.1.1 Integer Execution Unit

Of the 80 physical registers in the integer execution unit, 33 are available to hold the results of instructions in flight.

The 80 physical registers are allocated as follows:

- 39 registers to hold the values of the 31 Alpha architectural registers — the value of R31 is not stored — and the values of 8 PALshadow registers.
- 41 registers to hold results that are written by instructions that have not retired and released a physical register. Of those 41, the mapper holds 8 in reserve to map the instructions presented in the next two cycles[1]. This leaves 33 registers to hold the results of instructions in flight.

1 Reserving 8 registers is an approximation of a more complicated algorithm.

If 33 instructions that require an integer physical register have been mapped and have not retired and released a physical register, stage 2 of the pipeline (see Section 2.2) stalls if an additional integer physical register is requested.

For a schedule of integer instructions that contains loads or stores, the peak sustainable rate of physical register allocation is 3.3 registers per cycle. (This is obtained by dividing 33 instructions by a 10-cycle map-to-release latency.) Experiments have confirmed that 3.2 physical registers per cycle is a sustainable rate for integer schedules containing loads or stores. This assumes the loads and stores are best-case Dcache hits. If there are no loads or stores, it is possible to sustain 4 physical registers per cycle. Sometimes the best schedule has loads and stores grouped together and has significant stretches of register-to-register instructions.

3.9.1.2 Floating-Point Execution Unit

Of the 72 physical registers in the floating-point execution unit, 37 are available to hold the results of instructions in flight.

The 72 physical registers are allocated as follows:

- 31 registers to hold the values of the 31 Alpha architectural registers — the value of F31 is not stored.
- 41 registers to hold results that are written by instructions that have not yet retired and released a physical register. Of these 41, the mapper holds 4 in reserve to map the instructions presented in the next two cycles[1]. This leaves 37 registers to hold the results of instructions in flight.

If 37 instructions that require a floating-point physical register have been mapped and have not retired and released a physical register, stage 2 of the pipeline (see Section 2.2) stalls if an additional floating-point physical register is requested.

For a schedule of floating-point instructions that contains floating-point loads, the peak sustainable rate of physical register allocation is 2.85 registers per cycle. (This is obtained by dividing 37 instructions by a 13-cycle map-to-release latency.) Experiments have confirmed that 2.9 physical registers per cycle[2]is a sustainable rate for floating-point schedules containing loads. This assumes the loads and stores are best-case Dcache hits.

Floating-point stores take 3 cycles longer to retire than a floating-point operate. Even though a store does not free a register, it delays the retiring of subsequent instructions. For schedules of floating-point instructions that contain floating-point stores, the peak sustainable rate of physical register allocation is 2.31 registers per cycle. (This is obtained by dividing 37 instructions by a 16-cycle map-to-release latency.) Experiments have confirmed that 2.3 physical registers per cycle is a sustainable rate.

For schedules with no load or stores, only 2 floating-point operate instructions can be executed per cycle, and physical register allocation should not be a limit for schedules that respect the latencies of the instructions. This is true for square root and divide only if the instructions retire early (see Section 2.4.1).

---

1 Reserving 4 registers is an approximation of a more complicated algorithm.

2 The fact that the experimental result is larger than our analytic result is due to approximations of the map-to-release latencies and number of reserved registers.

3.9.1.3 Register Files

The integer and floating-point register files are separate. Schedules that intermix integer and floating-point instructions must separately meet the limits for allocating integer physical registers and floating-point physical registers. For example, a schedule that requires two integer physical registers and two floating-point physical registers per cycle is sustainable.

3.9.2 Ebox Slotting and Clustering

As described in Section 2.1.2, the integer execution unit has four functional units, implemented as two nearly-identical functional unit clusters labeled 0 and 1. Each cluster has an upper (U) and lower (L) functional unit called a subcluster. When instructions are decoded, they are statically assigned (or *slotted*) to an upper or lower subcluster. When instructions are issued, they are dynamically assigned (or *clustered*) to cluster 0 or cluster 1. To obtain optimal performance, the programmer must understand the algorithms used for slotting and clustering.

The slotting of an instruction is determined by its opcode and its position in the aligned octaword that contains the instruction. The details of the slotting algorithm are described in Section 2.3.2 and in Appendix A.

Most integer instructions have a one-cycle latency for consumers that execute within the same cluster. There is an additional one-cycle delay associated with producing a value in one cluster and consuming the value in the other cluster. If it is not possible to provide two cycles of latency for an integer instruction, controlling the cluster assignment of the producer and consumer is necessary to avoid a stall.

The following rules are used to issue an instruction:

- An instruction is a candidate to be issued when its operands are data ready.
  - Values produced by integer instructions will be data ready in one cluster before another.
  - Values loaded from cache or memory are available in both clusters at the same time.
- Older data-ready instructions have priority over younger instructions.
- An instruction assigned to the upper subcluster (U) will first check if it can issue on cluster 1, then on cluster 0.
- An instruction assigned to the lower subcluster (L) will first check if it can issue on cluster 0, then on cluster 1.

Appendix B contains an example of scheduled code that considers these issue rules.

A
Ebox Slotting Rules

Ebox slotting is the assignment of integer instructions to upper (U) and lower (L) subclusters (see Section 2.1.2). The slotting of an instruction is determined by its opcode (see Section 2.3.1) and by the contents of the aligned octaword that contains the instruction (see Section 2.3.2).

Table 2–2 classifies the subcluster requirements of each instruction. There are three categories:

Code	Meaning
U	The instruction only executes in an upper subcluster.
L	The instruction only executes in a lower subcluster.
E	The instruction could execute in either an upper or lower subcluster, or the instruction does not execute in an integer pipeline (such as floating-point instructions).

If all of the instructions in an aligned octaword are classified as U or L, the slotting assignments are completely determined by the instruction classification. If one or more of the instructions are classified as E, the slotting assignments are determined by all of the instructions in the aligned octaword, as defined in Table 2–3. Instructions that do not execute on an integer functional unit, such as floating-point instructions and NOPs, do affect the slotting of integer instructions contained in the same aligned octaword.

This appendix provides some slotting rules that can be inferred from Table 2–3. Table A–1 presents the slotting assignments from Table 2–3, sorted by the relevant rule.

Sections A.1 through A.4 list the rules for slotting four instructions contained in an aligned octaword.

A.1 Rule 1 — Four of a Kind

a. If all four instructions in an octaword are classified U, the classification is honored, and the slotting is all U.

b. If all four instructions in an octaword are classified L, the classification is honored, and the slotting is all L.

A.2 Rule 2 — Three of a Kind

a. If three and only three instructions are classified U, the classification is honored and the other instruction is slotted to L.

b. If three and only three instructions are classified L, the classification is honored and the other instruction is slotted to U.

A.3 Rule 3 — Two of a Kind

a. If two and only two instructions are classified U, the classification is honored and the other two instructions are slotted to L.

b. If two and only two instructions are classified L, the classification is honored and the other two instructions are slotted to U.

A.4 Rule 4 — One of a Kind and None of a Kind

If an aligned octaword contains at most one instruction classified as U and at most one instruction classified as L, then the slotting is done in terms of the two aligned quadwords contained in the octaword. Note we use little endian ordering of quadwords within octawords and instructions within quadwords; see the *Alpha Architecture Handbook, Version 4*, Section 2.3.

a. If one and only one instruction in an octaword is classified U and at most one instruction in the octaword is classified as L, then in the aligned quadword containing the instruction classified as U, the classification of the instruction to U is honored, and the other instruction in the quadword is slotted to L.

b. If one and only one instruction in an octaword is classified L and at most one instruction in the octaword is classified as U, then in the aligned quadword containing the instruction classified as L, the classification of the instruction to L is honored, and the other instruction in the quadword is slotted to U.

c. If both instructions in the second quadword of an octaword are classified as E, then the first instruction in the second quadword is slotted to L and the second is slotted to U.

d. If both instructions in the first quadword of an octaword are classified as E, then the slotting of these instructions is the same as the slotting of the second quadword. That is, the first instruction in the first quadword is given the same slotting as the first instruction in the second quadword, and the second instruction in the first quadword is given the same slotting as the second instruction in the second quadword.

Table A–1 Instruction Slotting for an Aligned Octaword

Rules from Sections A.1 Through A.4	Classification[1] 3 2 1 0	Slotting 3 2 1 0
Rule 1a	U U U U	U U U U
Rule 1b	L L L L	L L L L
Rule 2a	E U U U	L U U U
	U E U U	U L U U
	U U E U	U U L U
	U U U E	U U U L
	L U U U	L U U U

Table A–1 Instruction Slotting for an Aligned Octaword (Continued)

Rules from Sections A.1 Through A.4	Classification[1] 3 2 1 0	Slotting 3 2 1 0
	U L U U	U L U U
	U U L U	U U L U
	U U U L	U U U L
Rule 2b	E L L L	U L L L
	L E L L	L U L L
	L L E L	L L U L
	L L L E	L L L U
	U L L L	U L L L
	L U L L	L U L L
	L L U L	L L U L
	L L L U	L L L U
Rule 3a	U U E E	U U L L
	U U E L	U U L L
	U U L E	U U L L
	U E U E	U L U L
	U E U L	U L U L
	U L U E	U L U L
	U E E U	U L L U
	U E L U	U L L U
	U L E U	U L L U
	E U U E	L U U L
	E U U L	L U U L
	L U U E	L U U L
	E U E U	L U L U
	L U E U	L U L U
	E U L U	L U L U
	E E U U	L L U U
	E L U U	L L U U
	L E U U	L L U U
Rule 3b	L L E E	L L U U
	L L E U	L L U U
	L L U E	L L U U
	L E L E	L U L U

Table A–1 Instruction Slotting for an Aligned Octaword (Continued)

Rules from Sections A.1 Through A.4	Classification[1] 3 2 1 0	Slotting 3 2 1 0
	L U L E	L U L U
	L E L U	L U L U
	L E E L	L U U L
	L U E L	L U U L
	L E U L	L U U L
	E L L E	U L L U
	E L L U	U L L U
	U L L E	U L L U
	E L E L	U L U L
	E L U L	U L U L
	U L E L	U L U L
	E E L L	U U L L
	E U L L	U U L L
	U E L L	U U L L
Rule 3a, 3b	U U L L	U U L L
	U L U L	U L U L
	U L L U	U L L U
	L U U L	L U U L
	L U L U	L U L U
	L L U U	L L U U
Rule 4a, 4b	E L E U	U L L U
	E L U E	U L U L
	E U E L	L U U L
	E U L E	L U L U
	U E E L	U L U L
	U E L E	U L L U
	L E E U	L U L U
	L E U E	L U U L
Rule 4a, 4d	E U E E	L U L U
	U E E E	U L U L
Rule 4b, 4d	E L E E	U L U L
	L E E E	L U L U
Rule 4a, 4b, 4d	L U E E	L U L U

Table A–1 Instruction Slotting for an Aligned Octaword (Continued)

Rules from Sections A.1 Through A.4	Classification[1] 3 2 1 0	Slotting 3 2 1 0
	U L E E	U L U L
Rule 4c, 4a	E E E U	U L L U
	E E U E	U L U L
Rule 4c, 4b	E E L E	U L L U
	E E E L	U L U L
Rule 4c, 4a, 4b	E E U L	U L U L
	E E L U	U L L U
Rule 4c, 4d	E E E E	U L U L

[1] Instructions are ordered right to left within an octaword, in little-endian order, as defined in the *Alpha Architecture Handbook, Version 4*, Section 2.3. The first instruction is on the right, and the last is on the left.

B
An Example of Carefully Tuned Code

This appendix provides an example of high performance coding techniques for the 21264. As the example, we consider a loop to compute the sum of a vector of 16-bit integers. After presenting the loop written in C code, we introduce a succession of coding techniques that tune the loop for the 21264.

B.1 Initial Example C Code

The following is the loop written in C:

```
unsigned int sum( unsigned short *p, int length)
{
        unsigned int csum = 0;
        for (i= 0; i < length; i++) {
          csum += *p++;
        }
        return csum;
}
```

B.2 Inner Loop as Alpha Assembly Language

The following is a simple translation of the inner loop from Section B.1 into Alpha assembly language:

```
# $16 is the pointer p
# $17 is the counter length
# $0 is csum, assume it is initalized to zero

loop:
        LDWU    $18,    ($16)               # Load *p
        LDA     $16,    2($16)              # P++
        LDA     $17     -1($17)             # Decrement length
        ADDQ    $0,     $18,    $0          # csum += *p
        BGT     $17,    loop                # Loop back
```

To obtain high performance we need to apply a number of standard optimizations to this loop, such as loop unrolling, software pipelining, and prefetching. Before we do this, we introduce SIMD parallelism and change the loop to operate on four 16-bit words at a time. To keep the example simple, we assume that the vector *p* is quadword-aligned, that the vector length is a multiple of four, and that the length is less than 65536. Our strategy is to load a quadword HGFEDCBA and split it into two chunks

HG00DC00 and 00FE00BA. By splitting the quadword, we introduce 16 guard bits between the data chunks. We then sum each chunk within the loop, accumulating two sums per chunk. On loop exit, we will need to accumulate all of the partial sums.

B.3 Applying SIMD Parallelism to the Inner Loop

The following is a SIMD version of the inner loop from Section B.2:

```
# $16 is the pointer p
# $17 is the counter length
# $24, $25 hold the partial sums on loop exit

loop:
        LDQ     $18,    ($16)           # Load *p: HGFEDCBA
        ZAPNOT  $18,    0x33,   $0      # Chunk 0: 00FE00BA
        ZAP     $18,    0x33,   $1      # Chunk 1: HG00DC00
        SRL     $1,     16,     $1      # Shift: 00HG00DC
        ADDQ    $24,    $0,     $24     # Accumulate 0
        ADDQ    $25,    $1,     $25     # Accumulate 1
        LDA     $16,    8($16)          # p++
        LDA     $17,    -4($17)         # Decrement length
        BGT     $17,    loop            # Loop back
```

At the end of the loop, the partial sums are in $24 and $25, and they need to be accumulated.

B.4 Optimizing the SIMD Loop

We now consider an optimized version of the SIMD loop from Section B.3. We apply the standard optimizations: unroll the SIMD loop by two, software pipeline the loop, and introduce a prefetch. We carefully schedule the code to control physical register utilization and Ebox slotting and clustering, as described in Section 3.9.

The following code is blocked into five aligned octaword groups of four instructions; each group is contained in an aligned octaword. For each instruction, we indicate the functional unit on which the instruction will execute (one of U0, U1, L0, L1).

The goal of this loop is to execute in five cycles; it is scheduled as if the target is an in-order processor.

```
# $16 is the pointer p
# $17 is the counter length
# $24, $25 hold the partial sums on loop exit
# $18, $19 have the first two quadwords of data:
#    $18: HGFEDCBA
#    $19: PONMLKJI

.align 4 # Octaword alignment
loop:
        ZAPNOT  $18,    0x33,   $0      # U1 chunk 0: 00FE00BA
        BIS     $31,    $31,    $31     # L  NOP
        ZAP     $18,    0x33,   $1      # U0 chunk 1: HG00DC00
        LDQ     $18,    16($16)         # L1 load 2 ahead *p: HGFEDCBA
```

```
ADDQ    $24,    $0,     $24         # U1 accumulate 0
BIS     $31,    $31,    $31         # L  NOP
SRL     $1,     16,     $1          # U0 shift: 00HG00DC
LDA     $17,    -8($17)             # L0 countdown

ZAPNOT  $19,    0x33,   $0          # U1 chunk 3: 00NM00JI
BIS     $31,    $31,    $31         # L  NOP
ZAP     $19,    0x33,   $27         # U0 chunk 4: PO00LK00
LDQ     $19,    24($16)             # L0 load 2 ahead *p: PONMLKJIL1

ADDQ    $24,    $0,     $24         # U1 accumulate 0
ADDQ    $25,    $1,     $25         # L0 accumulate 1
SRL     $27,    16,     $27         # U0 shift: 00PO00LK
LDL     $31,    512($16)            # L1 prefetch

LDA[1]  $16,    16($16)             # U1 p++
ADDQ    $25,    $27,    $25         # L0 accumulate 1
BGT     $17,    loop                # U0 loop control
BIS     $31,    $31,    $31         # L  NOP (replace with fall through)
```

Sections B.5 through B.8 discuss how this loop is optimized for the 21264, considering branch prediction, instruction latency, physical register utilization, memory bandwidth, and Ebox slotting and clustering.

B.5 Branch Prediction Considerations

We are assuming the loop from Section B.4 has a large trip count. The branch predictor will quickly train on a loop-closing branch (within a few iterations) and essentially no cycles will be spent in branch mispredict penalties.

B.6 Instruction Latency Considerations

When an integer ALU instruction produces a result in one octaword that is consumed in the next, both the producer and consumer are carefully scheduled so that they are assigned to the same cluster. The scheduling avoids stalls due to the one cycle cross-cluster delay. For example, the result of the ZAPNOT in the first octaword in the code example from Section B.4 is read by the ADDQ in the second octaword; both the ZAP-NOT and the ADDQ will execute on cluster 1. Cluster assignment is controlled by observing the rules described in Section 3.9.2; we describe this process in Section B.9.

---

1 In the fifth aligned octaword group, consider that the fourth instruction, rather than being a NOP, was the beginning of the epilog of the loop. If this fourth instruction was an L or E, it would function as shown. If the fourth instruction was U, that octaword would slot LLUU. To preserve the five-cycle schedule, the first two instructions would need to be swapped:

```
ADDQ    L0
LDA     L1
BGT     U1
XXX     U0
```

The vector data is prefetched with a LDL in the fourth octaword. We are assuming each LDQ is a Dcache hit. We have allowed five cycles from the LDQ to the consumers of the load data. This is longer than the minimum latency. However, there will be contention for the Dcache with the prefetched data returning from memory. Some stalls may occur. It is best to separate as far as possible loads from consumers. In this software pipelined schedule, we read the data from the previous load in the same cycle that we issue the next instance of the load. We could only increase the distance from load to consumer by further unrolling the loop.

B.7 Physical Register Considerations

As described in Section 3.9.1, a physical register is allocated when an instruction is mapped and released when the instruction retires. Instructions that do not write a register, or that write register R31, do not require a physical register. Each of the first four octawords in the schedule described in Section B.9 require allocation of three physical registers. The fifth octaword requires two physical registers. On average, the schedule requires 2.8 physical registers per octaword. As described in Section 3.9.1, this is a sustainable rate for integer schedules containing loads. This schedule will not stall due to physical register requirements.

We added 3 NOPs to the schedule to reduce the physical register requirements. Without the NOPs, the requirement is 3.5 physical registers per octaword, which is not a sustainable rate for a schedule with loads. If we know we are going to stall for other reasons (for example, we exceed the memory bandwidth of the machine), then the NOPs serve no purpose.

Physical registers are released when instructions retire, and instructions retire in order. The latency from map-to-release (see Table 3–1) is longer for integer loads and stores than for integer operates. In the example in Section B.4, we place the loads as the last instruction in each octaword, to permit the integer operates that precede the load in the octaword to retire without waiting for the load.

Branches might require additional processing when they are retired. To minimize retire latency, it is best not to put an instruction with a long retire latency (such as a load) in the same octaword as a branch. This is a small effect, and should only be done if it does not introduce any other stalls in the schedule.

B.8 Memory Bandwidth Considerations

The loop in Section B.4 is attempting to do two quadword fetches (16 bytes) every five cycles. For example, on a 500-MHz machine, this is 16 bytes per 10 nanoseconds or 1.6 GB/second. This rate is comfortably within the Bcache bandwidth of a 500-MHz machine, but might be slightly beyond the sustainable memory bandwidth. For memory-resident data, we cannot achieve a loop iteration in five cycles. However, by prefetching 512 bytes ahead, we will continually have eight cache-line requests outstanding, which will fully utilize the available memory bandwidth and achieve the peak performance of the system.

B.9 Ebox Slotting Considerations

This section describes how the instructions from Section B.4 are slotted and clustered, using the rules defined in Appendix A and Section 3.9.2.

First Aligned Octaword

For the following first aligned octaword from Section B.4:

```
ZAPNOT  $18,    0x33,   $0          # U1 chunk 0: 00FE00BA
BIS     $31,    $31,    $31         # L  NOP
ZAP     $18,    0x33,   $1          # U0 chunk 1: HG00DC00
LDQ     $18,    16($16)             # L1 load 2 ahead *p: HGFEDCBA
```

- Slotting

  ZAPNOT and ZAP are classified U, LDQ is L, and BIS is classified E. By rule 3a, BIS is assigned L.

- Clustering

  All of the instructions in the octaword are data ready, so they are issued in order as follows:

  – ZAPNOT

    Slotted upper (U). R18 is generated by a load and is available on both clusters simultaneously. Upper instructions try cluster 1 first. This succeeds. Assigned to U1.

  – BIS

    Because the BIS is a NOP, it is not clustered.

  – ZAP

    Slotted upper (U). R18 is available on both clusters. Upper instructions try cluster 1 first. This fails. Cluster 0 is available. Assigned to U0.

  – LDQ

    Slotted lower (L). R16 is available only on cluster 1 (except for the first loop iteration) because R16 was written into cluster 1 in the fifth aligned octaword. Cluster 1 is available. Assigned to L1.

Second Aligned Octaword

For the following second aligned octaword from Section B.4:

```
ADDQ    $24,    $0,     $24         # U1 accumulate 0
BIS     $31,    $31,    $31         # L  NOP
SRL     $1,     16,     $1          # U0 shift: 00HG00DC
LDA     $17,    -8($17)             # L0 countdown
```

- Slotting

  SRL is classfied U and the remaining operations are classified E. By Rule 4a, LDA is assigned L and, by rule 4d, ADDQ and BIS are assigned U and L.

- Clustering

  All of the instructions in the octaword are data ready, so they are issued in order as follows:

  – ADDQ

Slotted upper (U). R24 is available on both clusters. R0 is available only on cluster 1. Assigned to U1.

– BIS

Because the BIS is a NOP, it is not clustered.

– SRL

Slotted upper (U). R1 is available only on cluster 0. Assigned to U0.

– LDA

Slotted lower (L). R17 is available on both clusters. Lower instructions try cluster 0 first. This succeeds. Assigned to L0.

Third Aligned Octaword

For the following third aligned octaword from Section B.4:

```
ZAPNOT  $19,    0x33,   $0          # U1 chunk 3: 00NM00JI
BIS     $31,    $31,    $31         # L  NOP
ZAP     $19,    0x33,   $27         # U0 chunk 4: PO00LK00
LDQ     $19,    24($16)             # L0 load 2 ahead *p: PONMLKJIL1
```

- Slotting

  The pattern is the same as the first octaword. ZAP and ZAPNOT are classified U, LDQ is L, and BIS is classified E. By rule 3a, BIS is assigned L.

- Clustering

  All of the instructions in the octaword are data ready, so they are issued in order as follows:

  – ZAPNOT

  Slotted upper (U). R19 is generated by a load, and is available on both clusters simultaneously. Upper instructions try cluster 1 first. This succeeds. Assigned to U1.

  – BIS

  Because the BIS is a NOP, it is not clustered.

  – ZAP

  Slotted upper (U). R19 is available on both clusters. Upper instructions try cluster 1 first. This fails. Cluster 0 is available. Assigned to U0.

  – LDQ

  Slotted lower (L). R16 is available on both clusters. Lower instructions try cluster 0 first. This succeeds. Assigned to L0.

Fourth Aligned Octaword

For the following fourth aligned octaword from Section B.4:

```
ADDQ    $24,    $0,     $24         # U1 accumulate 0
ADDQ    $25,    $1,     $25         # L0 accumulate 1
SRL     $27,    16,     $27         # U0 shift: 00PO00LK
LDL     $31,    512($16)            # L1 prefetch
```

- Slotting

  The fourth octaword has the same pattern as the second. SRL is classfied U, LDL is classified L, and the remaining operations are classified E. By rule 4d, ADDQ and ADDQ are assigned U and L.

- Clustering

  All of the instructions in the octaword are data ready, so they are issued in order as follows:

  – ADDQ

    Slotted upper (U). R24 is available on both clusters. R0 is available only on cluster 1. Assigned to U1.

  – ADDQ

    Slotted lower (L). R25 and R1 are available on both clusters. Lower instructions try cluster 0 first. This succeeds. Assigned to L0.

  – SRL

    Slotted upper (U). R27 is available only on cluster 0. Assigned to U0.

  – LDL

    Slotted lower (L). R16 is available on both clusters. Lower instructions try cluster 0 first. This fails. Assigned to L1.

Fifth Aligned Octaword

For the following fifth aligned octaword from Section B.4:

```
LDA     $16,    16($16)             # U1 p++
ADDQ    $25,    $27,    $25         # L0 accumulate 1
BGT     $17,    loop                # U0 loop control
BIS     $31,    $31,    $31         # L  NOP (replace with fall through)
```

- Slotting

  In the fifth octaword, the branch is classified U and the remaining operations are classified E. By Rule 4a, BIS is assigned L, and, by rule 4d, LDA and ADDQ are assigned U and L, respectively.

- Clustering

  All of the instructions in the octaword are data ready, so they are issued in order as follows:

  – LDA

Slotted upper (U). R16 is available on both clusters. Upper instructions try cluster 1 first. This succeeds. Assigned to U1.

– ADDQ

Slotted lower (L). R25 is available only on cluster 0. R27 is available only on cluster 0. Assigned to L0.

– BGT

Slotted upper (U). R17 is available on both clusters. Upper instructions try cluster 1 first. This fails. Cluster 0 is available. Assigned to U0.

– BIS

Because the BIS is a NOP, it is not clustered.

C
Controlling the Execution Order of Loads

The 21264 has no training mechanism that is similar to the stWait table, for avoiding a load-miss load replay trap or a load-load order replay trap (see Section 3.7). A load-miss load replay trap occurs when a load is followed by another load to the same address and the first load misses. The processor replay traps until the data is loaded into the register for the first load. A load-load order replay trap occurs when two loads to the same address issue out of order. Of the two traps, a load-miss load replay trap is more costly, because it must wait for data to return from the Bcache or memory.

The best way to avoid either load replay trap is to keep the value from the first load in a register and avoid doing the second load. Most reloads can be identified and removed by a compiler that uses partial redundancy elimination. However, some constructs require additional work. We present an example and some solutions below.

Caution: Most repeated loads do not cause a trap. Extra instructions should be inserted to remove a possible replay trap only when it is known through profiling that a replay trap is occurring or in very carefully scheduled code. Introducing extra instructions to guard against every possible load-miss load or load-load order replay trap can be very expensive. Experimentation has determined that it usually lowers performance.

Sample Problem

Consider the following C fragment:

```
long *p, *q,s;
...
if (*p) {
*q = 0;
s = *p;
}
...
```

A straightforward translation is:

```
        LDQ     $0,     0($16)          # Load *p
        BEQ     $0,     skip            # Skip ahead if zero
        STQ     $31,    0($17)          # Possibly aliasing store to *q
        LDQ     $0,     0($16)          # Load *p
skip:
```

We need to reload *p because the store to *q may change the value of *p, if *p* equals *q*. Assume that *p* and *q* are very unlikely to be equal[1]. Assume that we also know that the first reference to *p is likely to miss the data cache; for example, it could be the first reference to an array element. When we execute this code fragment, the first LDQ will miss the Dcache and the second LDQ will replay trap until the data from the first load returns.

Introducing a Test

We can avoid the second load by testing if *p* equals *q*. To keep our example simple, we assume both *p* and *q* are aligned pointers. If *p* and *q* are not equal, there is no alias and we do not need the second load; we can reuse the register holding the loaded value. If *p* and *q* are equal, we can use the register that holds the value that is stored in *q*.

```
        LDQ     $0,     0($16)              # Load *p
        BEQ     $0,     skip                # Skip ahead if zero
        STQ     $31,    0($17)              # Possibly aliasing store to *q
        CMPEQ   $16,    $17,    $1          # Test if p == q?
        CMOVNE  $1,     $31,    $0          # If yes, place *q value into $0
skip:
```

On the frequently executed path, we have replaced a LDQ with a CMPEQ and a CMOVNE. This should usually be faster than doing the LDQ. If a replay trap is occurring, the replacement will be much faster.

Introducing a Delay

An alternative method is to delay the second LDQ by introducing an explicit data dependency between the two loads. This delays the second load, but avoids a replay trap, which would delay all subsequent instructions.

We exploit the identity: `A xor 0 = A`

```
        LDQ     $0,     0($16)              # Load *p
        BEQ     $0,     skip                # Skip ahead if zero
        STQ     $31,    0($17)              # Possibly aliasing store to *q
        XOR     $0,     $0,     $1          # $1 is zero
        XOR     $16,    $1,     $16         # $16 is unchanged
        LDQ     $0,     0($16)              # Load *p
skip:
```

The result of the first load contributes to the address calculation of the second load, and the second load cannot issue until the data from the first load is available. This avoids the replay trap; the second LDQ is delayed, but all subsequent instructions that are not dependent on the load can continue issuing.

We can avoid using an extra register by overwriting and restoring $16. (Note that this is not safe to do with $30, the stack pointer.)

1 If they are equal, we may have a store-load order replay trap.

We exploit the identity: `(A xor B) xor B = A`

```
        LDQ     $0,     0($16)              # Load *p
        BEQ     $0,     skip                # Skip ahead if zero
        STQ     $31,    0($17)              # Possibly aliasing store to *q
        XOR     $16,    $0,     $16         #
        XOR     $16,    $0,     $16         # $16 is unchanged
        LDQ     $0,     0($16)              # Load *p
skip:
```

This same technique can be used for tightly scheduled code, where a specific timing is desired. For example, in a tight software pipelined loop, we can issue a load every 7 cycles, as follows. The following code fragment illustrates the technique. We use a MULQ to generate a zero.

```
loop:
        XOR     $16,    $0,     $16         # L1 7 cycles from last MUL
        MULQ    $0,     $31,    $0          # U1 makes a zero in 7 cycles
                                            #   on cluster 1
        LDA     $17,    -1($17)             # L0 decrement length
        BIS     $31,    $31,    $31         # U  NOP

        LDQ     $2,     0($16)              # L1 one iteration behind MUL
                                            #    every seventh cycle
        LDA     $16,    8($16)              # U1 increment pointer
        .                                   # Other possible code......
        .
        .

        BGT     $17,    loop                # U0 loop back
```

Multiple Load Quadword-Unaligned Instructions

Multiple load quadword-unaligned instructions can also introduce multiple references to the same quadword. For example, the standard Alpha sequence for loading an unaligned quadword begins:

```
LDQ_U   $0,     ($16)
LDQ_U   $1,     7($16)
```

If the data referenced by R16 is actually aligned, these loads will be to the same address. If the first load misses in the Dcache, a load-miss load replay trap occurs. If the trap does occur, it can be avoided by using the techniques described above. Either we can explicitly test for alignment and perform only one load when the pointer is aligned, or we can introduce a register dependence between the two uses of R16.

D

21264 Support for IEEE Floating-Point

The 21264 supports the IEEE floating-point operations as described in the *Alpha Architecture Handbook, Version 4*. Appendix B of that document describes how to construct a complete IEEE implementation for Alpha.

Support in the 21264 for a complete implementation of the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754 1985) is provided by a combination of hardware and software. The 21264 provides much more hardware support for the complete implementation of IEEE floating-point than was provided by earlier Alpha implementations. The IEEE floating-point instructions with the /S qualifier that are executed by the 21264 only need assistance from software completion when handling denormal values or results.

NaNs, Infinities, and Denormals

The 21264 provides the following support for NaNs, infinities, and denormals for IEEE floating-point instructions with the /S qualifier:

- The 21264 accepts both Signaling and Quiet NaNs as input operands and propagates them as specified by the Alpha architecture. In addition, the 21264 delivers a canonical Quiet NaN when an operation is required to produce a NaN value and none of its inputs are NaNs. Encodings for Signaling NaN and Quiet NaN are defined by the *Alpha Architecture Handbook, Version 4*.
- The 21264 accepts infinity operands and implements infinity arithmetic as defined by the IEEE standard and the *Alpha Architecture Handbook, Version 4*.
- The 21264 implements the FPCR[DNZ] bit. When FPCR[DNZ] is set, denormal input operand traps can be avoided for arithmetic operations that include the /S qualifier. When FPCR[DNZ] is clear, denormal input operands for arithmetic operations produce an unmaskable denormal trap. CPYSE/CPYSN, FCMOVxx, and MF_FPCR/MT_FPCR are not arithmetic operations, and pass denormal values without initiating arithmetic traps.

Exceptions

The 21264 implements precise exception handling in hardware. For IEEE floating-point instructions with the /S qualifier, this is denoted by the AMASK instruction returning bit 9 set. TRAPB instructions are treated as NOPs and are not issued.

For IEEE floating-point instructions with the /S qualifier, the 21264 implements the following disable bits in the floating-point control register (FPCR):

- Underflow disable (UNFD) and underflow to zero (UNDZ)

- Overflow disable (OVFD)
- Inexact result disable (INED)
- Division by zero disable (DZED)
- Invalid operation disable (INVD)

If one of these bits is set, and an instruction with the /S qualifier set generates the associated exception, the 21264 produces the IEEE nontrapping result and suppresses the trap. These nontrapping responses include correctly signed infinity, largest finite number, and Quiet NaNs as specified by the IEEE standard.

For IEEE floating-point instructions with the /S qualifier, the 21264 will not produce a denormal result for the underflow exception. Instead, a true zero (+0) is written to the destination register. In the 21264, the FPCR underflow to zero (UNDZ) bit must be set if underflow disable (UNFD) bit is set. If desired, trapping on underflow can be enabled by the instruction and the FPCR, and software may compute the denormal value as defined in the IEEE standard.

For IEEE floating-point instructions with the /S qualifier, the 21264 records floating-point exception information in two places:

- The FPCR status bits record the occurrence of all exceptions that are detected, whether or not the corresponding trap is enabled. The status bits are cleared only through an explicit clear command (MT_FPCR); hence, the exception information they record is a summary of all exceptions that have occurred since the last time they were cleared.
- If an exception is detected and the corresponding trap is enabled by the instruction, and is not disabled by the FPCR control bits, the 21264 will record the condition in the EXC_SUM register and initiate an arithmetic trap.

See Section 2.14 for information about the floating-point control register (FPCR).

Square Root

The 21264 implements IEEE SQRT for single (SQRTS) and double (SQRTT) precision in hardware. Note that the 21264 also implements the VAX SQRTF and SQRTG instructions.

Exceptional Input and Output Conditions

For IEEE floating-point instructions with the /S qualifier, Table D–1 lists all exceptional input and output conditions recognized by the 21264, along with the result and exception generated for each condition.

The following items apply to Table D–1:

- The 21264 traps on a Denormal input operand for all arithmetic operations unless FPCR[DNZ] = 1.
- Input operand traps take precedence over arithmetic result traps.
- The following abbreviations are used:

  Inf: Infinity

  QNaN: Quiet NaN

SNaN: Signalling NaN

CQNaN: Canonical Quiet NaN

Table D–1 Exceptional Input and Output Conditions

Alpha Instructions	21264 Hardware Supplied Result	Exception
ADDx SUBx INPUT		
Inf operand	±Inf	(none)
QNaN operand	QNaN	(none)
SNaN operand	QNaN	Invalid Op
Effective subtract of two Inf operands	CQNaN	Invalid Op
ADDx SUBx OUTPUT		
Exponent overflow	±Inf or ±MAX	Overflow
Exponent underflow	+0	Underflow
Inexact result	Result	Inexact
MULx INPUT		
Inf operand	±Inf	(none)
QNaN operand	QNaN	(none)
SNaN operand	QNaN	Invalid Op
0 * Inf	CQNaN	Invalid Op
MULx OUTPUT (same as ADDx)		
DIVx INPUT		
QNaN operand	QNaN	(none)
SNaN operand	QNaN	Invalid Op
0/0 or Inf/Inf	CQNaN	Invalid Op
A/0 (A not 0)	±Inf	Div Zero
A/Inf	±0	(none)
Inf/A	±Inf	(none)
DIVx OUTPUT (same as ADDx)		
SQRTx INPUT		
+Inf operand	+Inf	(none)
QNaN operand	QNaN	(none)
SNaN operand	QNaN	Invalid Op
-A (A not 0)	CQNaN	Invalid Op
-0	-0	(none)

Table D–1 Exceptional Input and Output Conditions (Continued)

Alpha Instructions	21264 Hardware Supplied Result	Exception
SQRTx OUTPUT		
Inexact result	root	Inexact
CMPTEQ CMPTUN INPUT		
Inf operand	True or False	(none)
QNaN operand	False for EQ, True for UN	(none)
SNaN operand	False for EQ, True for UN	Invalid Op
CMPTLT CMPTLE INPUT		
Inf operand	True or False	(none)
QNaN operand	False	Invalid Op
SNaN operand	False	Invalid Op
CVTfi INPUT		
Inf operand	0	Invalid Op
QNaN operand	0	Invalid Op
SNaN operand	0	Invalid Op
CVTfi OUTPUT		
Inexact result	Result	Inexact
Integer overflow	Truncated result	Invalid Op
CVTif OUTPUT		
Inexact result	Result	Inexact
CVTff INPUT		
Inf operand	±Inf	(none)
QNaN operand	QNaN	(none)
SNaN operand	QNaN	Invalid Op
CVTff OUTPUT (same as ADDx)		
FBEQ FBNE FBLT FBLE FBGT FBGE LDS LDT STS STT CPYS CPYSN FCMOVx		

Glossary

This glossary provides definitions for specific terms and acronyms associated with the Alpha 21264 microprocessor and chips in general.

abort

The unit stops the operation it is performing, without saving status, to perform some other operation.

address space number (ASN)

An optionally implemented register used to reduce the need for invalidation of cached address translations for process-specific addresses when a context switch occurs. ASNs are processor specific; the hardware makes no attempt to maintain coherency across multiple processors.

address translation

The process of mapping addresses from one address space to another.

ALIGNED

A datum of size 2**N is stored in memory at a byte address that is a multiple of 2**N (that is, one that has N low-order zeros).

ALU

Arithmetic logic unit.

ANSI

American National Standards Institute. An organization that develops and publishes standards for the computer industry.

ASIC

Application-specific integrated circuit.

ASM

Address space match.

ASN

See address space number.

assert

To cause a signal to change to its logical true state.

AST

See asynchronous system trap.

asynchronous system trap (AST)

A software-simulated interrupt to a user-defined routine. ASTs enable a user process to be notified asynchronously, with respect to that process, of the occurrence of a specific event. If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine exits, the system resumes execution of the process at the point where it was interrupted.

bandwidth

Bandwidth is often used to express the rate of data transfer in a bus or an I/O channel.

barrier transaction

A transaction on the external interface as a result of an MB (memory barrier) instruction.

Bcache

See second-level cache.

bidirectional

Flowing in two directions. The buses are bidirectional; they carry both input and output signals.

BiSI

Built-in self-initialization.

BiST

Built-in self-test.

bit

Binary digit. The smallest unit of data in a binary notation system, designated as 0 or 1.

bit time

The total time that a signal conveys a single valid piece of information (specified in ns). All data and commands are associated with a clock and the receiver's latch on both the rise and fall of the clock. Bit times are a multiple of the 21264 clocks. Systems must produce a bit time identical to 21264's bit time. The bit time is one-half the period of the forwarding clock.

BIU

Bus interface unit. *See* Cbox.

Block exchange

Memory feature that improves bus bandwidth by paralleling a cache victim write-back with a cache miss fill.

board-level cache

See second-level cache.

boot

Short for bootstrap. Loading an operating system into memory is called booting.

BSR

Boundary-scan register.

buffer

An internal memory area used for temporary storage of data records during input or output operations.

bugcheck

A software condition, usually the response to software's detection of an "internal inconsistency," which results in the execution of the system bugcheck code.

bus

A group of signals that consists of many transmission lines or wires. It interconnects computer system components to provide communications paths for addresses, data, and control information.

byte

Eight contiguous bits starting on an addressable byte boundary. The bits are numbered right to left, 0 through 7.

byte granularity

Memory systems are said to have byte granularity if adjacent bytes can be written concurrently and independently by different processes or processors.

cache

See cache memory.

cache block

The smallest unit of storage that can be allocated or manipulated in a cache. Also known as a cache line.

cache coherence

Maintaining cache coherence requires that when a processor accesses data cached in another processor, it must not receive incorrect data and when cached data is modified, all other processors that access that data receive modified data. Schemes for maintaining consistency can be implemented in hardware or software. Also called cache consistency.

cache fill

An operation that loads an entire cache block by using multiple read cycles from main memory.

cache flush

An operation that marks all cache blocks as invalid.

cache hit

The status returned when a logic unit probes a cache memory and finds a valid cache entry at the probed address.

cache interference

The result of an operation that adversely affects the mechanisms and procedures used to keep frequently used items in a cache. Such interference may cause frequently used items to be removed from a cache or incur significant overhead operations to ensure correct results. Either action hampers performance.

cache line

See cache block.

cache line buffer

A buffer used to store a block of cache memory.

cache memory

A small, high-speed memory placed between slower main memory and the processor. A cache increases effective memory transfer rates and processor speed. It contains copies of data recently used by the processor and fetches several bytes of data from memory in anticipation that the processor will access the next sequential series of bytes. The 21264 microprocessor contains two onchip internal caches. *See also* write-through cache and write-back cache.

cache miss

The status returned when cache memory is probed with no valid cache entry at the probed address.

CALL_PAL instructions

Special instructions used to invoke PALcode.

Cbox

External cache and system interface unit. Controls the Bcache and the system ports.

central processing unit (CPU)

The unit of the computer that is responsible for interpreting and executing instructions.

CISC

Complex instruction set computing. An instruction set that consists of a large number of complex instructions. *Contrast with* RISC.

clean

In the cache of a system bus node, refers to a cache line that is valid but has not been written.

clock

A signal used to synchronize the circuits in a computer.

clock offset (or clkoffset)

The delay intentionally added to the forwarded clock to meet the setup and hold requirements at the Receive Flop.

CMOS

Complementary metal-oxide semiconductor. A silicon device formed by a process that combines PMOS and NMOS semiconductor material.

conditional branch instructions

Instructions that test a register for positive/negative or for zero/nonzero. They can also test integer registers for even/odd.

control and status register (CSR)

A device or controller register that resides in the processor's I/O space. The CSR initiates device activity and records its status.

CPI

Cycles per instruction.

CPU

See central processing unit.

CSR

See control and status register.

cycle

One clock interval.

data bus

A group of wires that carry data.

Dcache

Data cache. A cache reserved for storage of data. The Dcache does not contain instructions.

DDR

Dual-data rate. A dual-data rate SSRAM can provide data on both the rising and falling edges of the clock signal.

denormal

An IEEE floating-point bit pattern that represents a number whose magnitude lies between zero and the smallest finite number.

DIP

Dual inline package.

direct-mapping cache

A cache organization in which only one address comparison is needed to locate any data in the cache, because any block of main memory data can be placed in only one possible position in the cache.

direct memory access (DMA)

Access to memory by an I/O device that does not require processor intervention.

dirty

One status item for a cache block. The cache block is valid and has been written so that it may differ from the copy in system main memory.

dirty victim

Used in reference to a cache block in the cache of a system bus node. The cache block is valid but is about to be replaced due to a cache block resource conflict. The data must therefore be written to memory.

DMA

See direct memory access.

DRAM

Dynamic random-access memory. Read/write memory that must be refreshed (read from or written to) periodically to maintain the storage of information.

DTB

Data translation buffer. *Also defined as* Dstream translation buffer.

DTL

Diode-transistor logic.

dual issue

Two instructions are issued, in parallel, during the same microprocessor cycle. The instructions use different resources and so do not conflict.

ECC

Error correction code. Code and algorithms used by logic to facilitate error detection and correction. *See also* ECC error.

ECC error

An error detected by ECC logic, to indicate that data (or the protected “entity”) has been corrupted. The error may be correctable (soft error) or uncorrectable (hard error).

ECL

Emitter-coupled logic.

EEPROM

Electrically erasable programmable read-only memory. A memory device that can be byte-erased, written to, and read from. *Contrast with* FEPROM.

external cache

See second-level cache.

FEPROM

Flash-erasable programmable read-only memory. FEPROMs can be bank- or bulk-erased. *Contrast with* EEPROM.

FET

Field-effect transistor.

FEU

The unit within the 21264 microprocessor that performs floating-point calculations.

firmware

Machine instructions stored in nonvolatile memory.

floating point

A number system in which the position of the radix point is indicated by the exponent part and another part represents the significant digits or fractional part.

flush

See cache flush.

forwarded clock

A single-ended differential signal that is aligned with its associated fields. The forwarded clock is sourced and aligned by the sender with a period that is two times the bit time. Forwarded clocks must be 50% duty cycle clocks whose rising and falling edges are aligned with the changing edge of the data.

FPGA

Field-programmable gate array.

FPLA

Field-programmable logic array.

FQ

Floating-point issue queue.

framing clock

The framing clock defines the start of a transmission either from the system to the 21264 or from the 21264 to the system. The framing clock is a power-of-2 multiple of the 21264 **GCLK** frequency, and is usually the system clock. The framing clock and the input oscillator can have the same frequency. The add_frame_select IPR sets that ratio of bit times to framing clock. The frame clock could have a period that is four times the bit time with a add_frame_select of 2X. Transfers begin on the rising and falling edge of the frame clock. This is useful for systems that have system clocks with a period too small to perform the synchronous reset of the clock forward logic. Additionally, the framing clock can have a period that is less than, equal to, or greater than the time it takes to send a full four cycle command/address.

GCLK

Global clock within the 21264.

granularity

A characteristic of storage systems that defines the amount of data that can be read and/or written with a single instruction, or read and/or written independently.

hardware interrupt request (HIR)

An interrupt generated by a peripheral device.

high-impedance state

An electrical state of high resistance to current flow, which makes the device appear not physically connected to the circuit.

hit

See cache hit.

Icache

Instruction cache. A cache reserved for storage of instructions. One of the three areas of primary cache (located on the 21264) used to store instructions. The Icache contains 8KB of memory space. It is a direct-mapped cache. Icache blocks, or lines, contain 32 bytes of instruction stream data with associated tag as well as a 6-bit ASM field and an 8-bit branch history field per block. Icache does not contain hardware for maintaining cache coherency with memory and is unaffected by the invalidate bus.

IDU

A logic unit within the 21264 microprocessor that fetches, decodes, and issues instructions. It also controls the microprocessor pipeline.

IEEE Standard 754

A set of formats and operations that apply to floating-point numbers. The formats cover 32-, 64-, and 80-bit operand sizes.

IEEE Standard 1149.1

A standard for the Test Access Port and Boundary Scan Architecture used in board-level manufacturing test procedures.

ILP

Instruction-level parallelism.

Inf

Infinity.

Instruction queues

Both the integer issue queue (IQ) and the floating-point issue queue (FQ).

INT *nn*

The term INT*nn*, where *nn* is one of 2, 4, 8, 16, 32, or 64, refers to a data field size of *nn* contiguous NATURALLY ALIGNED bytes. For example, INT4 refers to a NATURALLY ALIGNED longword.

interface reset

A synchronously received reset signal that is used to preset and start the clock forwarding circuitry. During this reset, all forwarded clocks are stopped and the presettable count values are applied to the counters; then, some number of cycles later, the clocks are enabled and are free running.

Internal processor register (IPR)

Special registers that are used to configure options or report status.

IOWB

I/O write buffer.

IPGA

Interstitial pin grid array.

IQ

Integer issue queue.

ITB

Instruction translation buffer.

JFET

Junction field-effect transistor.

latency

The amount of time it takes the system to respond to an event.

LCC

Leadless chip carrier.

LFSR

Linear feedback shift register.

load/store architecture

A characteristic of a machine architecture where data items are first loaded into a processor register, operated on, and then stored back to memory. No operations on memory other than load and store are provided by the instruction set.

longword (LW)

Four contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 31.

LQ

Load queue.

LSB

Least significant bit.

machine check

An operating system action triggered by certain system hardware-detected errors that can be fatal to system operation. Once triggered, machine check handler software analyzes the error.

MAF

Miss address file.

main memory

The large memory, external to the microprocessor, used for holding most instruction code and data. Usually built from cost-effective DRAM memory chips. May be used in connection with the microprocessor's internal caches and an external cache.

masked write

A write cycle that only updates a subset of a nominal data block.

MBO

See must be one.

Mbox

This section of the processor unit performs address translation, interfaces to the Dcache, and performs several other functions.

MBZ

See must be zero.

MESI protocol

A cache consistency protocol with full support for multiprocessing. The MESI protocol consists of four states that define whether a block is modified (M), exclusive (E), shared (S), or invalid (I).

MIPS

Millions of instructions per second.

miss

See cache miss.

module

A board on which logic devices (such as transistors, resistors, and memory chips) are mounted and connected to perform a specific system function.

module-level cache

See second-level cache.

MOS

Metal-oxide semiconductor.

MOSFET

Metal-oxide semiconductor field-effect transistor.

MSI

Medium-scale integration.

multiprocessing

A processing method that replicates the sequential computer and interconnects the collection so that each processor can execute the same or a different program at the same time.

must be one (MBO)

A field that must be supplied as one.

must be zero (MBZ)

A field that is reserved and must be supplied as zero. If examined, it must be assumed to be UNDEFINED.

NaN

Not-a-Number. An IEEE floating-point bit pattern that represents something other than a number. This comes in two forms: signaling NaNs (for Alpha, those with an initial fraction bit of 0) and quiet NaNs (for Alpha, those with an initial fraction bit of 1).

NATURALLY ALIGNED

See ALIGNED.

NATURALLY ALIGNED data

Data stored in memory such that the address of the data is evenly divisible by the size of the data in bytes. For example, an ALIGNED longword is stored such that the address of the longword is evenly divisible by 4.

NMOS

N-type metal-oxide semiconductor.

NVRAM

Nonvolatile random-access memory.

OBL

Observability linear feedback shift register.

octaword

Sixteen contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 127.

OpenVMS Alpha operating system

The version of the open VMS operating system for Alpha platforms.

operand

The data or register upon which an operation is performed.

output mux counter

Counter used to select the output mux that drives address and data. It is reset with the Interface Reset and incremented by a copy of the locally generated forwarded clock.

PAL

Programmable array logic (hardware). A device that can be programmed by a process that blows individual fuses to create a circuit.

PALcode

Alpha privileged architecture library code, written to support Alpha microprocessors. PALcode implements architecturally defined behavior.

PALmode

A special environment for running PALcode routines.

parameter

A variable that is given a specific value that is passed to a program before execution.

parity

A method for checking the accuracy of data by calculating the sum of the number of ones in a piece of binary data. Even parity requires the correct sum to be an even number; odd parity requires the correct sum to be an odd number.

PGA

Pin grid array.

pipeline

A CPU design technique whereby multiple instructions are simultaneously overlapped in execution.

PLA

Programmable logic array.

PLCC

Plastic leadless chip carrier or plastic-leaded chip carrier.

PLD

Programmable logic device.

PLL

Phase-locked loop.

PMOS

P-type metal-oxide semiconductor.

PQ

Probe queue.

PQFP

Plastic quad flat pack.

primary cache

The cache that is the fastest and closest to the processor. The first-level caches, located on the CPU chip, composed of the Dcache and Icache.

program counter

That portion of the CPU that contains the virtual address of the next instruction to be executed. Most current CPUs implement the program counter (PC) as a register. This register may be visible to the programmer through the instruction set.

PROM

Programmable read-only memory.

pull-down resistor

A resistor placed between a signal line and a negative voltage.

pull-up resistor

A resistor placed between a signal line and a positive voltage.

QNaN

Quiet Nan. *See* NaN.

quad issue

Four instructions are issued, in parallel, during the same microprocessor cycle. The instructions use different resources and so do not conflict.

quadword

Eight contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 63.

RAM

Random-access memory.

RAS

Row address select.

RAW

Read-after-write.

READ_BLOCK

A transaction where the 21264 requests that an external logic unit fetch read data.

read data wrapping

System feature that reduces apparent memory latency by allowing read data cycles to differ the usual low-to-high sequence. Requires cooperation between the 21264 and external hardware.

read stream buffers

Arrangement whereby each memory module independently prefetches DRAM data prior to an actual read request for that data. Reduces average memory latency while improving total memory bandwidth.

receive counter

Counter used to enable the receive flops. It is clocked by the incoming forwarded clock and reset by the Interface Reset.

receive mux counter

The receive mux counter is preset to a selectable starting point and incremented by the locally generated forward clock.

register

A temporary storage or control location in hardware logic.

reliability

The probability a device or system will not fail to perform its intended functions during a specified time interval when operated under stated conditions.

reset

An action that causes a logic unit to interrupt the task it is performing and go to its initialized state.

RISC

Reduced instruction set computing. A computer with an instruction set that is paired down and reduced in complexity so that most can be performed in a single processor cycle. High-level compilers synthesize the more complex, least frequently used instructions by breaking them down into simpler instructions. This approach allows the RISC architecture to implement a small, hardware-assisted instruction set, thus eliminating the need for microcode.

ROM

Read-only memory.

RTL

Register-transfer logic.

SAM

Serial access memory.

SBO

Should be one.

SBZ

Should be zero.

scheduling

The process of ordering instruction execution to obtain optimum performance.

SDRAM

Synchronous dynamic random-access memory.

second-level cache

A cache memory provided outside of the microprocessor chip, usually located on the same module. Also called board-level, external, or module-level cache.

set-associative

A form of cache organization in which the location of a data block in main memory constrains, but does not completely determine, its location in the cache. Set-associative organization is a compromise between direct-mapped organization, in which data from a given address in main memory has only one possible cache location, and fully associative organization, in which data from anywhere in main memory can be put anywhere in the cache. An "*n*-way set-associative" cache allows data from a given address in main memory to be cached in any of *n* locations.

SIMD

Single instruction stream, multiple data stream.

SIMM

Single inline memory module.

SIP

Single inline package.

SIPP

Single inline pin package.

SMD

Surface mount device.

SNaN

Signaling NaN. *See* Nan.

SRAM

See SSRAM.

SROM

Serial read-only memory.

SSI

Small-scale integration.

SSRAM

Synchronous static random-access memory.

stack

An area of memory set aside for temporary data storage or for procedure and interrupt service linkages. A stack uses the last-in/first-out concept. As items are added to (pushed on) the stack, the stack pointer decrements. As items are retrieved from (popped off) the stack, the stack pointer increments.

STRAM

Self-timed random-access memory.

superpipelined

Describes a pipelined machine that has a larger number of pipe stages and more complex scheduling and control. *See also* pipeline.

superscalar

Describes a machine architecture that allows multiple independent instructions to be issued in parallel during a given clock cycle.

system clock

The primary skew controlled clock used throughout the interface components to clock transfer between ASICs, main memory, and I/O bridges.

tag

The part of a cache block that holds the address information used to determine if a memory operation is a hit or a miss on that cache block.

target clock

Skew controlled clock that receives the output of the RECEIVE MUX.

TB

Translation buffer.

tristate

Refers to a bused line that has three states: high, low, and high-impedance.

TTL

Transistor-transistor logic.

UART

Universal asynchronous receiver-transmitter.

UNALIGNED

A datum of size 2**N stored at a byte address that is not a multiple of 2**N.

unconditional branch instructions

Instructions that change the flow of program control without regard to any condition. *Contrast with* conditional branch instructions.

UNDEFINED

An operation that may halt the processor or cause it to lose information. Only privileged software (that is, software running in kernel mode) can trigger an UNDEFINED operation. (This meaning only applies when the word is written in all uppercase.)

UNPREDICTABLE

Results or occurrences that do not disrupt the basic operation of the processor; the processor continues to execute instructions in its normal manner. Privileged or unprivileged software can trigger UNPREDICTABLE results or occurrences. (This meaning only applies when the word is written in all uppercase.)

UVPROM

Ultraviolet (erasable) programmable read-only memory.

VAF

See victim address file.

valid

Allocated. Valid cache blocks have been loaded with data and may return cache hits when accessed.

VDF

See victim data file.

VHSIC

Very-high-speed integrated circuit.

victim

Used in reference to a cache block in the cache of a system bus node. The cache block is valid but is about to be replaced due to a cache block resource conflict.

victim address file

The victim address file and the victim data file, together, form an 8-entry buffer used to hold information for transactions to the Bcache and main memory.

victim data file

The victim address file and the victim data file, together, form an 8-entry buffer used to hold information for transactions to the Bcache and main memory.

virtual cache

A cache that is addressed with virtual addresses. The tag of the cache is a virtual address. This process allows direct addressing of the cache without having to go through the translation buffer making cache hit times faster.

VLSI

Very-large-scale integration.

VPC

Virtual program counter.

VRAM

Video random-access memory.

WAR

Write-after-read.

word

Two contiguous bytes (16 bits) starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 15.

write data wrapping

System feature that reduces apparent memory latency by allowing write data cycles to differ the usual low-to-high sequence. Requires cooperation between the 21264 and external hardware.

write-back

A cache management technique in which write operation data is written into cache but is not written into main memory in the same operation. This may result in temporary differences between cache data and main memory data. Some logic unit must maintain coherency between cache and main memory.

write-back cache

Copies are kept of any data in the region; read and write operations may use the copies, and write operations use additional state to determine whether there are other copies to invalidate or update.

write-through cache

A cache management technique in which a write operation to cache also causes the same data to be written in main memory during the same operation. Copies are kept of any data in a region; read operations may use the copies, but write operations update the actual data location and either update or invalidate all copies.

WRITE_BLOCK

A transaction where the 21264 requests that an external logic unit process write data.

Index

Numerics

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X