COMPAQ

Compiler Writer's Guide for the 21264/21364

Part Number: EC–0100A–TE

This document provides guidance for writing compilers for the 21264 and 21364 Alpha microprocessors. It is available at:

`ftp.compaq.com/pub/public/alphaCPUdocs`

Revision/Update Information: Revision 2.0, January 2002

Compaq Computer Corporation
Shrewsbury, Massachusetts

January 2002

Table of Contents

Figures

Tables

Preface

This guide should be thought of as one of a set of three documents. The other two are:

- The *Alpha Architecture Reference Manual, 4th Edition*, which contains the complete architecture information
- The appropriate hardware reference manual for a particular processor, which contains the complete hardware specification for that processor

All three documents are available at:

```
ftp.compaq.com/pub/products/alphaCPUdocs
```

Audience

This document provides guidance for compiler writers and other programmers who use the Alpha 21264 and 21364 microprocessors (referred to as the 21264/21364).

Content

This document contains the following chapters and appendixes:

Chapter 1, Introduction to the 21264 and 21364

> Provides an overview of the Alpha architecture and introduces the 21264 and 21364.

Chapter 2, Common 21264/21364 Hardware Features

> Contains sections of Chapter 2 of the 21264 and 21364 hardware reference manuals that are common to all processors and, most importantly, directly referenced within this guide. This information is correct but not complete. The complete information resides in the appropriate hardware reference manual.

Chapter 3, Guidelines for Compiler Writers

> Provides guidelines for taking advantage of the hardware features of the 21264 and 21364.

Appendix A, 21264/21364 Upper-Lower Rules Summary

> Provides rules to follow in scheduling instructions.

Appendix B, Checksum Inner Loop Schedule

> Provides an example for the rules described in Appendix A.

Appendix C, IEEE Floating-Point Conformance

Describes the 21264/21364 support for IEEE floating-point. It is directly based on Appendix A of the 21264 and 21364 Specifications.

The Glossary lists and defines terms associated with the 21264 and 21364.

An Index is also included.

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.

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

- Sign extension

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

Addresses

Unless otherwise noted, all addresses and offsets are hexadecimal.

Aligned and Unaligned

The terms *aligned* and *naturally aligned* are interchangeable and refer to data objects that are powers of two in size. An aligned datum of size 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

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 21264/21364 (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.

1 Introduction to the 21264 and 21364

This chapter provides a brief introduction to the Alpha architecture, Compaq Computer Corporation's RISC (reduced instruction set computing) architecture designed for high performance. The chapter then summarizes specific features of the Alpha 21264 and 21364 microprocessors.

The companion volumes to this guide:

- The *Alpha Architecture Reference Manual, 4th Edition* contains the complete architecture information.
- The appropriate hardware reference manual contains the complete hardware specification.

All three documents are available at:

```
ftp.compaq.com/pub/products/alphaCPUdocs
```

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 and 21364 use 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 and 21364 perform 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 and 21364 support a 48-bit or 43-bit virtual address (selectable under Internal Processor Register (IPR) control).

Virtual addresses as seen by the program are translated into physical memory addresses by the memory-management mechanism. The 21264 and 21364 support 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. A quadword is a 64-bit value.

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

1.1.3 Floating-Point Data Types

The following floating-point data types are supported:

- 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 21264 Microprocessor Features

The 21264 microprocessor is a superscalar pipelined processor. It is packaged in a 587-pin pin grid array (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 six instructions (peak) or four instructions (sustained) 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.

Refer to the *Alpha Architecture Reference Manual, 4th Edition,* Appendix E, for waivers and any other implementation-dependent information

1.3 21364 Microprocessor Features

The 21364 microprocessor is a superscalar pipelined processor manufactured using 0.18 µm CMOS 6-layer metal technology. It is packaged in a 1439-contact land grid array (LGA) 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 to large multiprocessor systems.

The 21364 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.

With the exception of an enlarged MAF and VAF (to support the enlarged MAF), the 21364 shares the same core as the 21264/EV68CB microprocessor.

Other 21364 features include:

- The ability to issue up to six instructions (peak) or four instructions (sustained) during each CPU clock cycle.
- A peak instruction execution rate of six times the CPU clock frequency.
- Affords a glueless, scalable multiprocessor system with directory-based coherence protocol.

- A 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 high-throughput pipelined floating-point units, capable of executing both VAX and IEEE floating-point data types.
- A 64KB virtually-addressed L1 instruction cache with 8-bit ASNs (MAX_ASN=255).
- A virtually-indexed, physically-tagged dual-read-ported, 64KB L1 data cache.
- An integrated 1.75MB 7-way associative L2 cache with the following performance characteristics:
  - Sustained access rate at 16 bytes/cycle, fully pipelined with no “bubbles”, resulting in 16GB/sec total read/write bandwidth at 1 GHz.
  - 16 victim buffers for L1 to L2 caches
  - 16 victim buffers for L2 cache to local or remote memory
  - ECC single-bit error correction, double-bit error detection (SECDED) code
  - 12 ns load-to-use latency at 1 GHz (a 12-cycle latency at 1 GHz)
- Supports a 48-bit or 43-bit virtual address (program selectable).
- Supports a 44-bit physical address.
- Has a four-point integrated network interface for direct interprocessor interconnect.
  - Each processor can directly connect to up to 4 other processors.
  - 10-GB/sec per processor.
  - 15 ns processor-to-processor latency.
  - Out-of-order network with adaptive routing.
  - Asynchronous clocking between processors.
- Has 8 to 10 channels (2 controllers $\times$ 4 to 5 channels each) of Rambus (RDRAM) memory.
  - Up to 800 MHz operation.
  - 30 ns CAS latency pin-to-pin.
  - 6-GB/sec read or write bandwidth – aggregate 12-GB/sec.
  - Directory-based cache coherence.
  - ECC SECDED code.
  - A fifth channel on each controller offers RAID-like memory redundancy protection.
- Has one I/O connection per processor with a 3-GB/sec interface.

- Has an onchip I/O write buffer with four 64-byte entries for I/O write transactions.
- An onchip, 32-entry load queue.
- An onchip, 32-entry store queue.
- An onchip, 15-entry miss address file for cache fill requests and I/O read transactions.
- A 15-entry onchip L1 Dcache victim buffer and a 16-entry onchip system victim buffer.
- An onchip, 32-entry probe queue, holding pending system port probe commands.
- Hardware cache/system memory coherence support.
- 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 1.5-V external interface.

Refer to the *Alpha Architecture Reference Manual, 4th Edition,* Appendix E, for waivers and any other implementation-dependent information.

2

Common 21264/21364 Hardware Features

This chapter contains sections of Chapter 2 of the 21264 and 21364 hardware reference manual that are common to all processors and referenced within this guide. This information is correct but should not be thought of as complete. You should download the appropriate hardware reference manual for the processor(s). You should also download the *Alpha Architecture Reference Manual*. All these documents are available in the same directory:

```
ftp.compaq.com/pub/products/alphaCPUdocs
```

2.1 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.

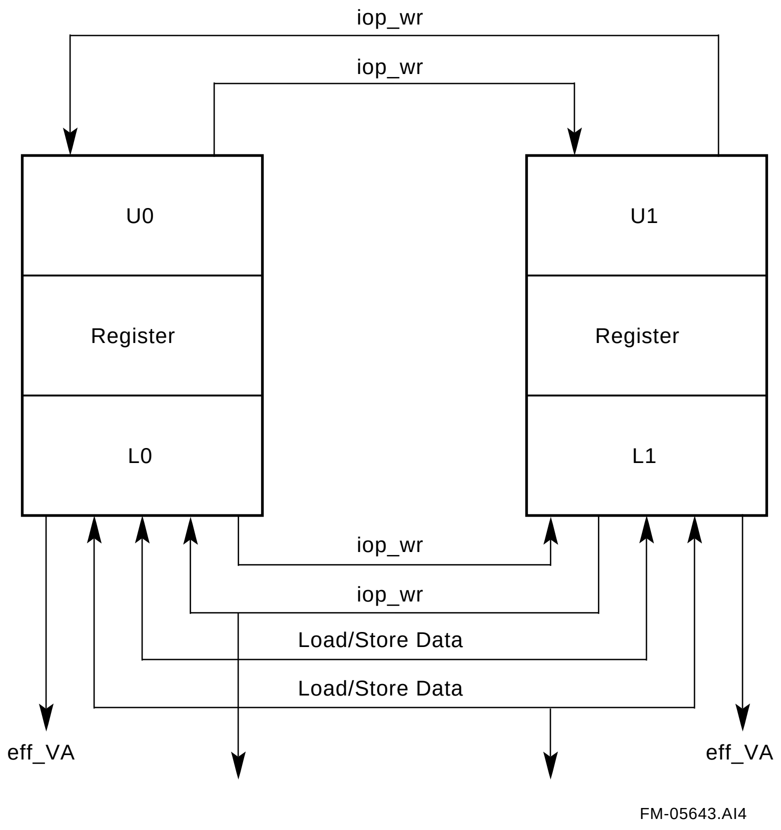
If a branch mispredict or exception occurs, the map logic backs up the contents of the integer and floating-point register rename maps to the state associated with the instruction that triggered the condition, and the prefetcher restarts at the appropriate virtual program counter (VPC). At most, 20 valid 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.

2.2 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 2–1 shows the integer execution unit. In the figure, *iop_wr* is the cross-cluster bus for moving integer result values between clusters.

Figure 2–1 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:
  - CTLZ, CTPOP, CTTZ
  - 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.

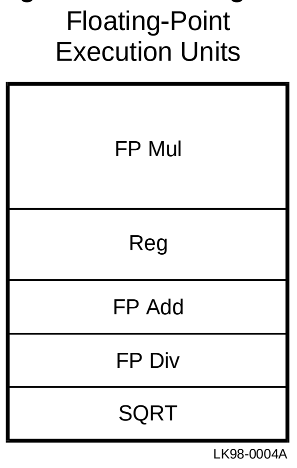
Ignoring cross-cluster delay, the two copies of the Ebox register file contain identical values. Each copy of the Ebox register file contains four read ports and six write ports. The four read ports are used to source operands to each of the two subclusters within a cluster. The six write ports are used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

2.3 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 2–2.

Figure 2–2 Floating-Point Execution Units



The Fbox contains the following resources:

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

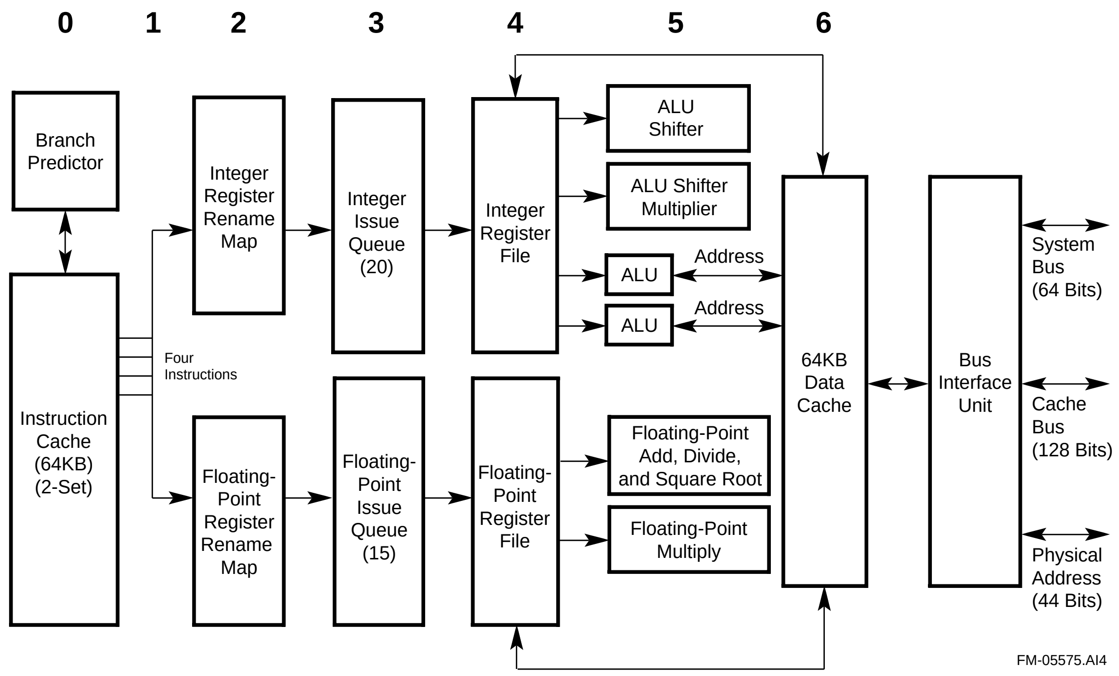
The 72 Fbox register file entries contain storage for the values of the 31 Alpha floating-point registers (F31 is not stored) and 41 values written by instructions that have not been retired.

The Fbox register file contains six read 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.

2.4 Pipeline Organization

The 7-stage pipeline provides an optimized environment for executing Alpha instructions. The pipeline stages (0 to 6) are shown in Figure 2–3 and described in the following paragraphs.

Figure 2–3 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 2.5.2 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-point 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.

2.4.1 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	—

Table 2–1 Pipeline Abort Delay (GCLK Cycles) (Continued)

Abort Condition	Penalty (Cycles)	Comments
ITB miss	7	—
Integer arithmetic trap	12	—
Floating-point arithmetic trap	13+latency	Add latency of instruction. See Section 2.5.3 for instruction latencies.

2.5 Instruction Issue Rules

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

2.5.1 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
cmov	L0, U0, L1, U1	Integer CMOV — either cluster
fadd	FA	All floating-point operate instructions except multiply, divide, square root, and conditional move instructions
fcbr	FA	Floating-point conditional branch instructions
fcmov1	FA	Floating-point CMOV—first half
fcmov2	FA	Floating-point CMOV— second half
fdiv	FA	Floating-point divide instruction
fld	L0, L1	All floating-point load instructions
fmul	FM	Floating-point multiply instruction
fsqrt	FA	Floating-point square root instruction
fst	FST0, FST1, L0, L1	All floating-point store instructions
ftoi	FST0, FST1, L0, L1	FTOIS, FTOIT
iadd	L0, U0, L1, U1	Instructions with opcode 10_{16}, except CMPBGE
icbr	U0, U1	Integer conditional branch instructions
ild	L0, L1	All integer load instructions
ilog	L0, U0, L1, U1	AND, BIC, BIS, ORNOT, XOR, EQV, CMPBGE
imisc	U0	CTLZ, CTPOP, CTTZ, PERR, MINxxx, MAXxxx, PKxx, UNPKxx
imul	U1	Integer multiply instructions
ishf	U0, U1	Instructions with opcode 12_{16}
ist	L0, L1	All integer store instructions

Table 2–2 Instruction Name, Pipeline, and Types (Continued)

Class Name	Pipeline	Instruction Type
itof	L0, L1	ITOFS, ITOFF, ITOFT
jsr	L0	BR, BSR, JMP, CALL, RET, COR, HW_RET, CALL_PAL
lda	L0, L1, U0, U1	LDA, LDAH
mem_misc	L1	WH64, ECB, WMB
mx_fpcr	FM	Instructions that move data from the floating-point control register
mxpr	L0, L1 (depends on IPR)	HW_MTPR, HW_MFPR
nop	None	TRAP, EXCB, UNOP - LDQ_U R31, 0(Rx)
rpcc	L1	RPCC
rx	L1	RS, RC

2.5.2 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:

- 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.

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

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 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
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

2.5.3 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
cmov1	1	Only consumer is cmov2. Possible 1-cycle Ebox cross-cluster delay.
cmov2	1	Possible 1-cycle Ebox cross-cluster delay.
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.
fcbr	—	Does not produce register value.
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.
fdiv	12 9 15 12	Single precision - latency to consumer of result value. Single precision - latency to using divider again. Double precision - latency to consumer of result value. Double precision - latency to using divider again.
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.
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.
fsqrt	18 15 33 30	Single precision - latency to consumer of result value. Single precision - latency to using unit again. Double precision - latency to consumer of result value. Double precision - latency to using unit again.
fst	—	Does not produce register value.
ftoi	3	—
iadd	1	Possible 1-cycle Ebox cross-cluster delay.
icbr	—	Conditional branch. Does not produce register value.

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

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.
ilog	1	Possible 1-cycle Ebox cross-cluster delay.
imisc	3	Possible 1-cycle Ebox cross-cluster delay.
imul	7	Possible 1-cycle Ebox cross-cluster delay.
ishf	1	Possible 1-cycle Ebox cross-cluster delay.
ist	—	Does not produce register value.
itof	4	—
jsr	3	—
lda	1	Possible 1-cycle Ebox cross-cluster delay.
mem_misc	—	Does not produce register value.
mxpr	1 or 3	HW_MFPR: Ebox IPRs = 1. Ibox and Mbox IPRs = 3. HW_MTPR does not produce a register value.
nop	—	Does not produce register value.
rpcc	1	Possible 1-cycle cross-cluster delay.
rx	1	—
ubr	3	Unconditional branch. Does not produce register value.

2.6 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
BSR/JSR	10	JSR instruction mispredict is reported in stage 8.
Floating-point add	11	—
Floating-point conditional branch	11	Branch instruction mispredict is reported in stage 7.
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 2.6.1).
Floating-point multiply	11	—
Integer conditional branch	7	—

Table 2–5 Minimum Retire Latencies for Instruction Classes (Continued)

Instruction Class	Retire Stage	Comments
Integer multiply	7/13	Latency is 13 cycles for the MUL/V instruction.
Integer operate	7	—
Memory	10	—

2.6.1 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})

2.7 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. 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 Reference Manual, 4th Edition*.

Table 2–6 Instructions Retired Without Execution

Instruction Type	Notes
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.
FLTS	All (SQRT, ITOF) with F31 as destination.

Table 2–6 Instructions Retired Without Execution

Instruction Type	Notes
INTA, INTL, INTM, INTS	All with R31 as destination.
LDQ_U	All with R31 as destination.
MISC	TRAPB and EXCB are always removed. Others are never removed.

2.8 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.

2.8.1 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.

2.8.1.1 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.

2.8.1.2 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, or every 65536 cycles if I_CTL[ST_WAIT_64K] is set.

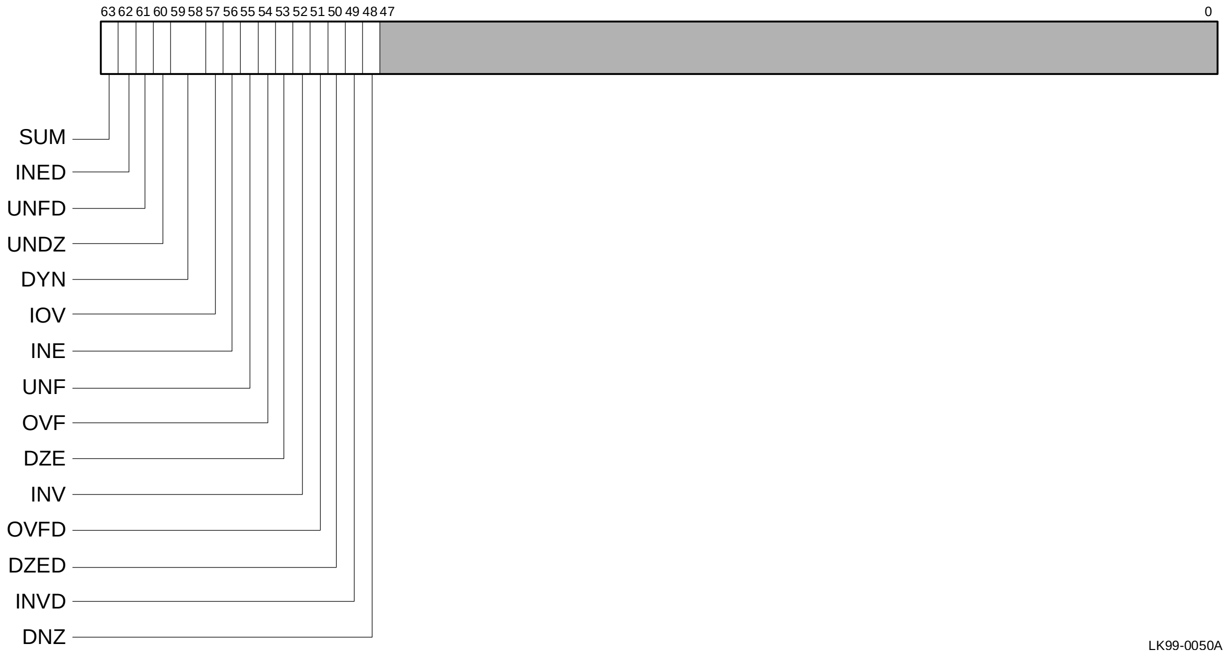
2.8.2 Other Mbox Replay Traps

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.

2.9 Floating-Point Control Register

The floating-point control register (FPCR) is shown in Figure 2–4.

Figure 2–4 Floating-Point Control Register



The floating-point control register fields are described in Table 2–7.

Table 2–7 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.

Table 2–7 Floating-Point Control Register Fields (Continued)

Name	Extent	Type	Description
UNFD	[61]	RW	Underflow Disable. The 21264/21364 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).

Name	Extent	Type	Description
UNDZ	[60]	RW	Underflow to zero. When UNDZ is set together with UNFD, underflow traps are disabled and the 21264/21364 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:

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

Name	Extent	Type	Description
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 21264/21364 is capable of producing the correct IEEE nontrapping result, that result is placed in the destination register and the trap is suppressed.

Table 2–7 Floating-Point Control Register Fields (Continued)

Name	Extent	Type	Description
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]	—	—

[1] Alpha architecture FPCR bit 47 (DNOD) is not implemented by the 21264 or 21364.

3
Guidelines for Compiler Writers

This chapter is a supplement to Appendix A of the *Alpha Architecture Reference Manual, 4th Edition*. That appendix presents general guidelines for software that are less dependent on the processor implementation. This chapter identifies some of the specific features of the 21264 and 21364 that affect performance and that can be controlled by a compiler writer or assembly-language programmer.

Note: Chapter 2 in the appropriate hardware reference manual describes the specific hardware features for which this chapter provides programming guidelines. Sections of that chapter are included in this guide and referenced in this chapter. Consult the appropriate hardware reference manual for complete information on hardware features for a particular Alpha processor. (The sections of Chapter 2 included in this guide are correct but not complete for all 21264 and 21364 processors.) You can download the hardware reference manual for your processor from:

```
ftp.compaq.com/pub/products/alphaCPUdocs/
```

3.1 Architecture Extensions

Various extensions have been provided to the Alpha architecture.

Use the AMASK instruction (see Section 2.15 of the hardware reference manual that is appropriate for your particular processor) to test for the presence of these extensions.

If you using this document before the 21364 hardware reference manual is available, you can use the AMASK values that are described in the 21264/EV68CB hardware reference manual.

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 Reference Manual, 4th Edition*, for more details on AMASK.

There are also new instructions for memory prefetch, described in Section 3.6.

3.2 Instruction Alignment

Where possible, branch targets should be octaword aligned. Although any NOP instruction can be used to pad code for alignment, the UNOP is recommended because it ensures backwards compatibility. The 21264/21364 discards NOP instructions early in the pipeline, so the main costs are space in the instruction cache and instruction fetch bandwidth. See Appendix A of the *Alpha Architecture Reference Manual, 4th Edition*, for the encodings to use for NOP instructions.

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 Reference Manual, 4th Edition*.

Because the 21264/21364 implements the BWX extension, it is beneficial to do unaligned word 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 21264/21364, 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.

3.4.1 Need for Single Successors

Code should be arranged so that each aligned octaword 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 21264/21364 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 21264/21364 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 21264/21364 is removing instructions from the queues (issuing them) faster than they can be filled. The 21264/21364 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 the branches are rarely all 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 be done unconditionally. To conditionalize the store (*p=a), a dummy location called the bitbucket is created on the stack, and the address register for the store is overwritten with the bitbucket 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   B,A,R0
CMOVNE  R0,R31,C
```

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

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

or equivalently as:

```
CMPLT   B,A,R1
CMPLT   D,C,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 predication will work.

The target of a RET instruction is predicted with a return stack, as described in Appendix A of the *Alpha Architecture Reference Manual, 4th Edition.*

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 all the bytes are nonzero or a `CMPBGE` instruction can determine which byte is nonzero. See Appendix B for an example.

3.6 Prefetching

Prefetching is very important (by a factor of 2) for loops dominated by memory latency or bandwidth. The 21264 and 21364 both support three styles of prefetch, but the 21364 has more highly developed support for marking a cache block as having a short temporal cache life with the *evict next* qualifier.

Table 3–1 Prefetch Support Summary

Prefetch Type	Instruction	Processor Support	Description
Normal prefetch	PREFETCH	21264 and 21364	Prefetch for loading data that is expected to be read only. Reduces the latency to read memory.
Normal prefetch, evict next	PREFETCH_EN	21264 and 21364	Normal prefetch and mark for preferential eviction in future cache fills.
Prefetch with modify intent	PREFETCH_M	21264 and 21364	Prefetch for data that will probably be written. Reduces the latency to read memory and bus traffic.
Prefetch with modify intent, evict next	PREFETCH_MEN	21364 only	Prefetch with modify intent and mark for preferential eviction in future cache fills.
Write hint – 64 bytes	WH64	21264 and 21364	Execute if the program intends to write an entire aligned block of 64 bytes. Reduces the amount of memory bandwidth required to write a block of data.
Write hint – 64 bytes, evict next	WH64EN	21364 only	Hint to the processor that the corresponding block should be marked for preferential eviction in future cache fills.

The actual cache eviction policy is implementation-dependent and described in the corresponding implementation's hardware reference manual.

The prefetch instructions and write hints are recognized as prefetches or ignored on pre-21264/21364 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 21264/21364 have more potential benefit than previous Alpha implementations and unnecessary prefetching is less costly. Support for prefetching varies between implementations; consult the appropriate hardware reference manual for particular support and the *Alpha Architecture Reference Manual, 4th Edition* for general information.

How far ahead to prefetch depends on whether the processor is a 21264 or 21364. The number of instructions that are prefetched can be normally controlled by a compiler switch. The 21264 has an 8-entry miss address file (MAF); the 21364 a 15-entry MAF. Therefore, prefetch ahead further with the 21364, as follows:

21264

Always prefetch ahead at least two cache blocks for each stream. 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 2 blocks for each stream 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.

21364

Always prefetch ahead at least two cache blocks for each stream. Prefetch farther ahead if possible (up to 10 blocks), unless doing so requires more than 15 offchip references to be in progress at the same time. That is, for a loop that references *n* streams, prefetch ahead 2 blocks for each stream or 15/*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 on the stack (bitbucket) 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 bit bucket
WH64    R4
```

3.7 Avoiding Replay Traps

The 21264/21364 can have several memory operations in progress at the same time, rather than necessarily waiting for one memory operation to complete before starting another. The 21264/21364 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 2.8.

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 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 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 those 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 21264/21364 is unable to break.

Section 2.8 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 contain specific recommendations for avoiding them.

3.7.1 Store-Load Order 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 the load executes before the store, a replay trap occurs and execution restarts at the load. This is called a store-load order 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 faster than replay trapping.

The FTOI*x* and ITOF*x* instructions transfer data between the floating-point and integer register files. Because 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-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 comes back from the cache. This is called a load-miss load replay trap.

3.7.4 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 21264/21364 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. See Section A.3.3 of the *Alpha Architecture Reference Manual, 4th Edition,* for a discussion of laying out data to avoid direct-mapped cache thrashes. If possible, avoid loops where a single iteration or nearby interations touch data that is 32KB apart. Avoid creating data that is a multiple of 32KB and pad it with extra cache blocks if possible. 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.

3.7.5 Store Queue Overflow

Each store instruction is buffered in the store queue until it retires, up to a maximum of 32. 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 21264/21364 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 21264/21364 is 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 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. 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 for a compiler. More detailed models must take into account physical register allocation and Ebox slotting and clustering. Such models are difficult to get right because the compiler cannot easily predict the order in which instructions will be executed. However, it is possible to produce schedules that achieve higher performance by more accurately modeling the 21264/21364. This section describes such a model.

3.9.1 Physical Registers

Physical registers are a resource that need to be managed to achieve optimal performance. As described in Section 2.1, 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 3–2 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 3–2 is an approximation; the register mapper has a number of special cases and edge conditions that are ignored.

Table 3–2 Minimum Latencies from Map to Release of a Physical Register

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

[1] See Table 2–5 and Section 2.6.1.

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

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 eight 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 eight in reserve to map the instructions presented in the next two cycles[1]. That leaves the 33 registers to hold the results of instructions in flight.

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 2.4) 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 are 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[2]. 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 2.4) 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.) Experi-

1 Reserving 8 registers is an approximation of a more complicated algorithm.
2 Reserving 4 registers is an approximation of a more complicated algorithm.

ments have confirmed that 2.9 physical registers per cycle[1]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 2.6.1).

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 2.2, 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 they are decoded, instructions are statically assigned (or *slotted*) to an upper or lower subcluster. When they are issued, instructions are dynamically assigned (or *clustered*) to cluster 0 or cluster 1. To obtain optimal performance, the scheduler 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 2.5.2 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.

1 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.

- 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
21264/21364 Upper-Lower Rules Summary

As required for instructions, the subclusters *upper, lower,* and *either* are defined in Table 2–3.

Cases are organized by what is the maximum number of requirements for either upper or lower.

Cases Quad-pack 3&4

When your quad-pack of instructions has more than two instructions requiring upper, or more than two instructions requiring lower, the more-than-two's will meet their requirement; others will go to the other level. The machine can't execute this in one cycle; there are only two uppers and two lowers.

```
SLL             U   Requires upper
ZAP      =>     U   Requires upper
addq            L   Either; forced to lower
BGT             U   Requires upper
```

Case Quad-pack 2

When your quad-pack of instructions has two instructions requiring upper, and/or two instructions requiring lower, these two's will meet their requirement; others will go to the other level.

```
LDQ             L   Requires lower
STT      =>     L   Requires lower
NOP             U   Vacated slot for NOP; forced to upper
XOR             U   Either; forced to upper
```

Cases Quad-pack 0&1

When your quad-pack has less than two instructions requiring upper, and less than two instructions requiring lower, each of the first and last PAIR's of instructions will map one to upper and one to lower. So the CASES here are handled by pairs. Many loops can be pair-wise assigned.

Case Either Pair 1 (Also Fits Quad-pack 2, Requires Split by Pairs)

When one or more of the instruction pair has a requirement, that requirement is met; the other instruction go to the other level.

```
MB        =>    L   Requires lower
CMOVEQ          U   Either; pair forced to upper
```

Case Last Pair 0

When the third and fourth instructions of the quad-pack have NO requirements (could be either L or U), the third is lower and the fourth is upper.

```
ADDT      =>    L   Vacated slot for float; rule assigns to lower
ADDQ            U   Either; rule assigns upper
```

Case First Pair 0

When the first and second instructions of the quad-pack have NO requirements (could be either L or U), their UL pattern is the same as that of the third and fourth instructions (whether or not the third and fourth instructions had any requirement)!

```
BIS             U   Either; upper same as third
BIC      =>     L   Either; lower same as fourth
S4ADDQ          U   Either; pair forced to upper
BR              L   Requires lower
```

B
Checksum Inner Loop Schedule

The following algorithm illustrates 21264/21364 scheduling and SIMD parallelism. A significantly more efficient algorithm is provided aftewards.

```
# Here we have the inner loop for a one's
#   compliment checksum of 16-bit data.
# $16 is the pointer.
# $17 is the counter.
# $18, and $19 are input quadwords.
#
# We split the input DCBA to 0C0A and D0B0.
# The latter gets shifted down
#   and they are added into dual acc's $24 and $25.
# The LDBU is the two-block-ahead-pfetch.
# We would want that to be larger for a big loop
#   going to memory. This two-at-a-time speeds the
#   code and we can only do this for 65536 times without
#   overflow problems.
#
# NOPs bring the code density to 2.8 physical registers
#   per cycle (quadpack). Only do this if you are sure.
# If you are going to stall ANYWAY, you don't
#   want the NOPs, just code it packed.

# In the first aligned octaword, the zaps are upper,
#   the load is lower, and the ALU is forced to lower
#   to meet 2 upper and 2 lower. The 1's and 0's
#   on the sides show precedence.

# In the second half of the second aligned octaword, the
#   shift is upper, forcing partner LDA (an ALU operation)
#   to lower. The indeterminate first 2, then
#   'follow' the second 2.

# The third and fourth quad packs have the same pattern
#   as the first and second.

# For the fifth quad pack, the branch is upper, forcing
#   its 'nothing' partner to lower. The first two
#   'follow' the second two.
```

```
 # Note that register usage in sequential cycles
 #   has been held to the same side.

 # Note, loads can be delayed to prefetch filling
 #   traffic, so leave the MT's in lower wherever
 #   possible to relieve rescheduling strain.

 # For iteration counts of moderate size, the BGT will
 #   be trained, so we only take a mispredict on the
 #   final fall through. For (consistent) iteration counts
 #   of 10 or less, the branch can learn to exit without
 #   a mispredict. (There still remains a line mispredict
 #   of one cycle.)

 # So this will predominantly execute in the proposed
 #   five cycles per loop iteration.

 # The use, use, refill of r18 is totally legal.
 # There is no impact on the physical registers of
 #   which architectual names are used. so you can
 #   create new R1's every cycle, or do the R18 trick
 #   with no cost. The R18 trick allows 5 cycles of
 #   latency to bring up data from the Dcache. This
 #   is tight. For a heavy-duty loop, we would have
 #   unrolled to do 16 data in 9 cycles (2.9 reg's).
 # We could then have provided more latency
 #   coverage for the stores. Only one prefetch would
 #   still be needed in the larger loop. It is still
 #   only 1/2 cache-block per loop. The 21264 releases
 #   the 'irrelevant' extra prefetches cheaply. For
 #   a heavy-duty memory loop, we would have the prefetch
 #   at 512($16), eight cache blocks.

 # Note, side can affect load/store operations.
 # We do the lda $16, 16($16).
 # It is NOT possible to do two loads in the next
 #   cycle because $16 is only valid on one side and loads
 #   can only be done in lower.

 # Note, the data ARRIVING from loads is posted in
 #   both sides.

 # Note, the extra blank line has no function to
 #   the assembler. But it is great for a person to
 #   see where the quad-packs are.

 # Note that the example uses Tru64 Unix assembler syntax.

loop:
```

```
ZAPNOT  $18,    51,     $0      # U1 low zebra
UNOP    $31,    $31,    $31     # L (NOPs not clustered)
ZAP     $18,    51,     $1      # U0 hi zebra
LDQ     $18,    ($16)           # L1 get next data
                                # LDA result side 1

ADDQ    $24,    $0,     $24     # U1 accum 0
UNOP    $31,    $31,    $31     # L (NOPs not clustered)
SRL     $1,     16,     $1      # U0 hi=>lo
LDA     $17,    -8($17)         # L0 countdown

ZAPNOT  $19,    51,     $0      # U1 low zebra
UNOP    $31,    $31,    $31     # L (NOPs not clustered)
ZAP     $19,    51,     $27     # U0 hi zebra
LDQ     $19,    8($16)          # L0 get next data

ADDQ    $24,    $0,     $24     # U1 accum 0
ADDQ    $25,    $1,     $25     # L0 accum 1
SRL     $27,    16,     $27     # U0 hi=>lo
LDBU    $31,    128($16)        # L1 prefetch

LDA     $16,    16($16)         # U1 move pointer
ADDQ    $25,    $27,    $25     # L0 accum 1
BGT     $17,    loop            # U0 loop cntl
Wrapup code....
```

Better Design Algorithm

The following unscheduled algorithm provides superior performance to the previous algorithm.

```
LDQ     $18,    ($16)           # Fetch quadword,
                                #   provide latency, of course.
ADDQ    $24,    $18,    $24     # Full quadword add
                                #   each 16 carries to the bottom
                                #   of some other 16.....
CMPULT  $24,    $18,    $18     # Except the one, out of the top
                                #   $24 is only less, unsigned, than
                                #   either argument, if there
                                #   was overflow!
ADDQ    $24,    $18,    $24     # This wraps back the top.
                                # For scheduling, one might want
                                #   to collect the overflow(s) into
                                #   separate register(s).
                                # Collect the 16's within the register
                                #   at wrapup time.

                                # If we want to maximize the $18
                                #   latency, then, instead --
ADDQ    $24,    $18,    $25     # Pingpong between two accum's.
```

```
        CMPULT  $25,    $24,    $24     # One if overflow, else zero.
        ADDQ    $25,    $24,    $25     # Gather overflow (has to fit).
```

The four instructions in the better algorithm replace the following six instructions in the first algorithm:

```
        ZAPNOT  $19,    51,     $0      # U1 low zebra
        ZAP     $18,    51,     $1      # U0 hi zebra
        LDQ     $18,    ($16)           # L1 get next data
                                        # LDA result side 1
        SRL     $1,     16,     $1      # U0 hi=>lo
        ADDQ    $24,    $0,     $24     # U1 accum 0
        ADDQ    $25,    $1,     $25     # L0 accum 1
```

C

IEEE Floating-Point Conformance

The 21264/21364 supports the IEEE floating-point operations defined in the *Alpha System Reference Manual*, *Revision* 8 and therefore also from the *Alpha Architecture Reference Manual, 4th Edition*. Support for a complete implementation of the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754 1985) is provided by a combination of hardware and software.

The 21264/21364 provides the following hardware features to facilitate complete support of the IEEE standard:

- The 21264/21364 implements precise exception handling in hardware, as denoted by the AMASK instruction returning bit 9 set. TRAPB instructions are treated as NOPs and are not issued.
- The 21264/21364 accepts both Signaling and Quiet NaNs as input operands and propagates them as specified by the Alpha architecture. In addition, the 21264/21364 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 Reference Manual, 4th Edition*.
- The 21264/21364 accepts infinity operands and implements infinity arithmetic as defined by the IEEE standard and the *Alpha Architecture Reference Manual, 4th Edition*.
- The 21264/21364 implements SQRT for single (SQRTS) and double (SQRTT) precision in hardware.

Note: In addition, the 21264/21364 also implements the VAX SQRTF and SQRTG instructions.

- The 21264/21364 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.
- The 21264/21364 implements the following disable bits in the floating-point control register (FPCR):
  - Underflow disable (UNFD)
  - 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 21264/21364 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.

  The 21264/21364 will not produce a Denormal result for the underflow exception. Instead, a true zero (+0) is written to the destination register. In the 21264/21364, 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.

The 21264/21364 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 21264/21364 will record the condition in the EXC_SUM register and initiate an arithmetic trap.

The following items apply to Table C–1:

- The 21264/21364 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

  For IEEE instructions with /S, Table C–1 lists all exceptional input and output conditions recognized by the 21264/21364, along with the result and exception generated for each condition.

Table C–1 Exceptional Input and Output Conditions

Alpha Instructions	21264/21364 Hardware Supplied Result	Exception
ADDx SUBx INPUT		
Inf operand	±Inf	(none)
QNaN operand	QNaN	(none)

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

Alpha Instructions	21264/21364 Hardware Supplied Result	Exception
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)
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

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

Alpha Instructions	21264/21364 Hardware Supplied Result	Exception
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		

See Section 2.9 for information about the floating-point control register (FPCR).

Glossary

This glossary provides definitions for specific terms and acronyms associated with the Alpha 21264/21364 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 21264/21364 clocks. Systems must produce a bit time identical to 21264/21364'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 21264/21364 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.

core

That part of the pipeline that lies between the L1 Icache and the L1 Dcache.

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 21264/21364 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 21264/21364 or from the 21264/21364 to the system. The framing clock is a power-of-2 multiple of the 21264/21364 **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 21264/21364.

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 21264/21364) 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 21264/21364 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; than, 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

Privileged architecture library. *See also* PALcode. *Also* 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 to a positive voltage.

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 21264/21364 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 21264/21364 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 upper case.)

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 upper case.)

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 21264/21364 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 21264/21364 requests that an external logic unit process write data.

Index