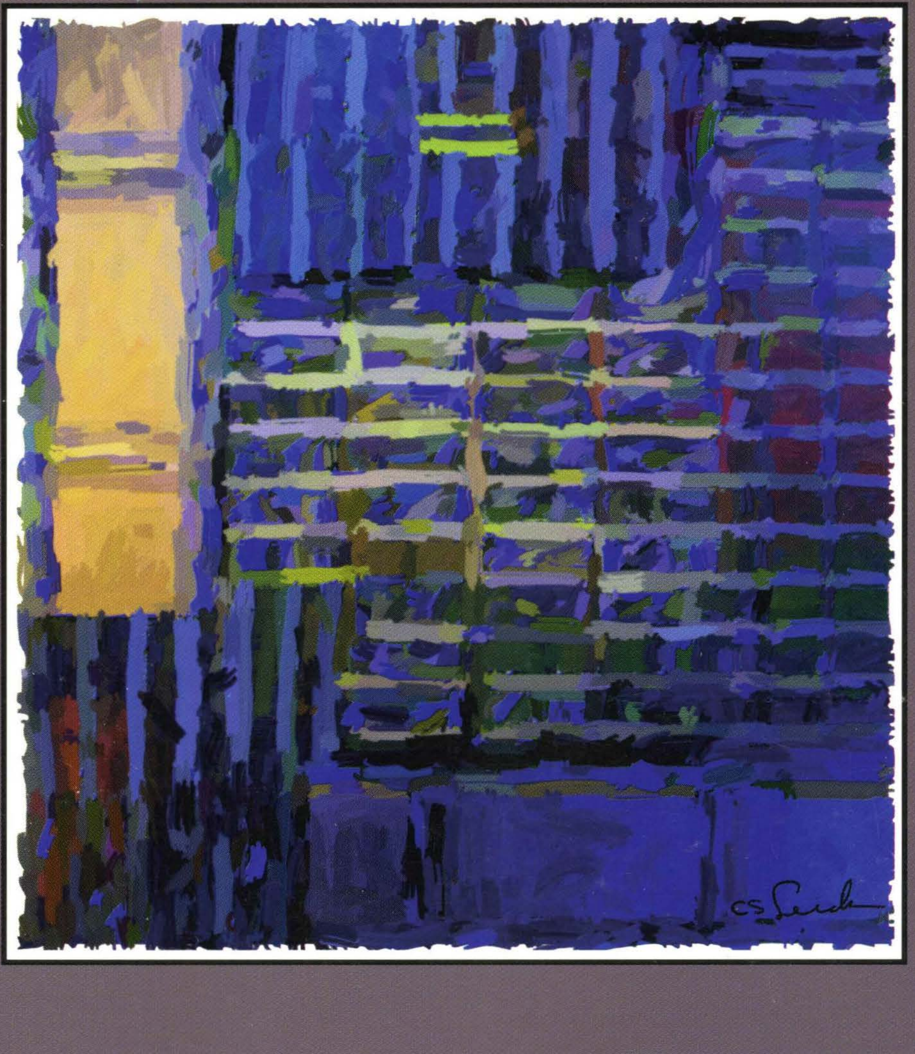
MOTOROLA

PowerPC™ 620 RISC Microprocessor User's Manual

MPC620UM/AD

PowerPC™ 620

RISC Microprocessor User's Manual



PowerPC™

MOTOROLA

MPC620UM/AD
8/97

PowerPC™ 620

RISC Microprocessor User's Manual

PowerPC

MOTOROLA

This document contains information on a new product under development. Motorola reserves the right to change or discontinue this product without notice. Information in this document is provided solely to enable system and software implementers to use PowerPC microprocessors. There are no express or implied copyright licenses granted hereunder to design or fabricate PowerPC integrated circuits or integrated circuits based on the information in this document.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.

Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

IBM is a registered trademark of International Business Machines Corporation. The PowerPC name, PowerPC logotype, PowerPC 601, PowerPC 602, PowerPC 603, PowerPC 603e, PowerPC 604, and PowerPC 620 are trademarks of International Business Machines Corporation used by Motorola under license from International Business Machines Corporation.

CHRP is a trademark of Apple Computer, Inc., International Business Machines Corporation, and Motorola, Inc. Power Macintosh is a trademark of Apple Computer, Inc.

CONTENTS

Paragraph Number	Title	Page Number

About This Book

Chapter 1 Overview

CONTENTS

Paragraph Number | Title | Page Number

Chapter 2
Programming Model

CONTENTS

Paragraph Number | Title | Page Number

CONTENTS

Paragraph Number | Title | Page Number

Chapter 3
Instruction and Data Cache Operation

CONTENTS

Chapter 4
Exceptions

CONTENTS

Paragraph Number	Title	Page Number

Chapter 5
Memory Management

CONTENTS

Paragraph Number | **Title** | **Page Number**

Chapter 6
Instruction Timing

CONTENTS

Paragraph Number **Title** **Page Number**

Chapter 7
Signal Descriptions

CONTENTS

Paragraph Number | Title | Page Number

CONTENTS

Chapter 8
System Interface Operation

CONTENTS

Paragraph Number	Title	Page Number

CONTENTS

Paragraph Number	Title	Page Number

CONTENTS

CONTENTS

CONTENTS

Paragraph Number | Title | Page Number

Chapter 9
Secondary Cache Interface

CONTENTS

Paragraph Number | Title | Page Number

Chapter 10
Performance Monitor

CONTENTS

Paragraph Number | Title | Page Number

Chapter 11
Power Management

CONTENTS

Paragraph Number | Title | Page Number

Appendix A
PowerPC Instruction Set Listings

Appendix B
Invalid Instruction Forms

Appendix C
Hardware Configuration

Appendix D
Bus Protocol Livelock Scenarios

Index

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

Figure Number	Title	Page Number

TABLES

TABLES

TABLES

TABLES

TABLES

TABLES

About This Book

The primary objective of this manual is to help hardware and software designers who are working with the PowerPC 620™ microprocessor. This book is intended as a companion to the *PowerPC™ Microprocessor Family: The Programming Environments, Rev. 1,* referred to as *The Programming Environments Manual.* Because the PowerPC architecture is designed to be flexible to support a broad range of processors, *The Programming Environments Manual* provides a general description of features that are common to PowerPC processors and indicates those features that are optional or that may be implemented differently in the design of each processor.

Note that *The Programming Environments Manual* does not attempt to replace the PowerPC architecture specification (documented in *The PowerPC Architecture: A Specification for a New Family of RISC Processors*), which defines the architecture from the perspective of the three programming environments and which remains the defining document for the PowerPC architecture.

The *PowerPC 620 RISC Microprocessor User's Manual* summarizes features of the 620 that are not defined by the architecture. This document and *The Programming Environments Manual* distinguishes between the three levels, or programming environments, of the PowerPC architecture, which are as follows:

- PowerPC user instruction set architecture (UISA)—The UISA defines the level of the architecture to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, memory conventions, and the memory and programming models seen by application programmers.
- PowerPC virtual environment architecture (VEA)—The VEA, which is the smallest component of the PowerPC architecture, defines additional user-level functionality that falls outside typical user-level software requirements. The VEA describes the memory model for an environment in which multiple processors or other devices can access external memory, defines aspects of the cache model and cache control instructions from a user-level perspective. The resources defined by the VEA are particularly useful for optimizing memory accesses and for managing resources in an environment in which other processors and other devices can access external memory.

- PowerPC operating environment architecture (OEA)—The OEA defines supervisor-level resources typically required by an operating system. The OEA defines the PowerPC memory management model, supervisor-level registers, and the exception model.

  Implementations that conform to the PowerPC OEA also conform to the PowerPC UISA and VEA.

It is important to note that some resources are defined more generally at one level in the architecture and more specifically at another. For example, conditions that can cause a floating-point exception are defined by the UISA, while the exception mechanism itself is defined by the OEA.

Because it is important to distinguish between the levels of the architecture in order to ensure compatibility across multiple platforms, those distinctions are shown clearly throughout this book.

For ease in reference, this book has arranged topics described by the architecture information into topics that build upon one another, beginning with a description and complete summary of 620-specific registers and progressing to more specialized topics such as 620-specific details regarding the cache, exception, and memory management models. As such, chapters may include information from multiple levels of the architecture. (For example, the discussion of the cache model uses information from both the VEA and the OEA.)

The information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation. For more information, contact your sales representative.

Audience

This manual is intended for system software and hardware developers and application programmers who want to develop products for the 620. It is assumed that the reader understands operating systems, microprocessor system design, the basic principles of RISC processing, and details of the PowerPC architecture.

Organization

Following is a summary and a brief description of the major sections of this manual:

- Chapter 1, "Overview," is useful for those who want a general understanding of the features and functions of the PowerPC architecture. This chapter describes the flexible nature of the PowerPC architecture definition, and provides an overview of how the PowerPC architecture defines the register set, operand conventions, addressing modes, instruction set, cache model, exception model, and memory management model.

- Chapter 2, "Programming Model," is useful for software engineers who need to understand the 620-specific registers, operand conventions, and details regarding how PowerPC instructions are implemented on the 620.
- Chapter 3, "Instruction and Data Cache Operation," provides a discussion of the cache and memory model as implemented on the 620.
- Chapter 4, "Exceptions," describes the exception model as implemented on the 620.
- Chapter 5, "Memory Management," provides descriptions of the PowerPC address translation and memory protection mechanism as implemented on the 620.
- Chapter 6, "Instruction Timing," describes instruction timing in the 620.
- Chapter 7, "Signal Descriptions," describes individual signals defined for the 620.
- Chapter 8, "System Interface Operation," describes interface operations on the 620.
- Chapter 9, "Secondary Cache Interface," provides information on the L2 cache interface operation, register set, and ECC errors, as well as providing timing diagrams.
- Chapter 10, "Performance Monitor," describes the operation of the performance monitor diagnostic tool incorporated in the 620.
- Chapter 11, "Power Management," describes the power saving mechanism implemented on the 620.
- Appendix A, "PowerPC Instruction Set Listings," lists all the PowerPC instructions. Instructions are grouped according to mnemonic, opcode, function, and form.
- This manual also includes a glossary and an index.

In this document, the terms "PowerPC 620 Microprocessor" and "620" are used to denote a microprocessor from the PowerPC architecture family. The PowerPC 620 microprocessors are available from Motorola as MPC620.

Suggested Reading

This section lists additional reading that provides background for the information in this manual as well as general information about the PowerPC architecture.

General Information

The following documentation provides useful information about the PowerPC architecture and computer architecture in general:

- The following books are available from the Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104; Tel. (800) 745-7323 (U.S.A.), (415) 392-2665 (International); internet address: mkp@mkp.com.
  - — *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.

Updates to the architecture specification are accessible via the world-wide web at http://www.austin.ibm.com/tech/ppc-chg.html.

— *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture*, by Apple Computer, Inc., International Business Machines, Inc., and Motorola, Inc.

— *Macintosh Technology in the Common Hardware Reference Platform*, by Apple Computer, Inc.

— *Computer Architecture: A Quantitative Approach*, Second Edition, by John L. Hennessy and David A. Patterson,

- *Inside Macintosh: PowerPC System Software,* Addison-Wesley Publishing Company, One Jacob Way, Reading, MA, 01867; Tel. (800) 282-2732 (U.S.A.), (800) 637-0029 (Canada), (716) 871-6555 (International).
- *PowerPC Programming for Intel Programmers,* by Kip McClanahan; IDG Books Worldwide, Inc., 919 East Hillsdale Boulevard, Suite 400, Foster City, CA, 94404; Tel. (800) 434-3422 (U.S.A.), (415) 655-3022 (International).

PowerPC Documentation

The PowerPC documentation is organized in the following types of documents:

- User's manuals—These books provide details about individual PowerPC implementations and are intended to be used in conjunction with *The Programming Environments Manual.* These include the following:

  — *PowerPC 601™ RISC Microprocessor User's Manual*:
  MPC601UM/AD (Motorola order #)

  — *PowerPC 602™ RISC Microprocessor User's Manual*:
  MPC602UM/AD (Motorola order #)

  — *PowerPC 603e™ RISC Microprocessor User's Manual with Supplement for PowerPC 603 Microprocessor*:
  MPC603EUM/AD (Motorola order #)

  — *PowerPC 604™ RISC Microprocessor User's Manual*:
  MPC604UM/AD (Motorola order #)

- *PowerPC Microprocessor Family: The Programming Environments*, Rev. 1 provides information about resources defined by the PowerPC architecture that are common to PowerPC processors. This document describes both the 64- and 32-bit portions of the architecture.
  MPCFPE/AD (Motorola order #)
- *Implementation Variances Relative to Rev. 1 of* The Programming Environments *Manual* is available via the world-wide web at http://www.mot.com/powerpc/.

- Addenda/errata to user's manuals—Because some processors have follow-on parts an addendum is provided that describes the additional features and changes to functionality of the follow-on part. These addenda are intended for use with the corresponding user's manuals. These include the following:
  - — *Addendum to PowerPC 603e RISC Microprocessor User's Manual: PowerPC 603e Microprocessor Supplement and User's Manual Errata:* MPC603EUMAD/AD (Motorola order #)
  - — *Addendum to PowerPC 604 RISC Microprocessor User's Manual*: *PowerPC 604e™ Microprocessor Supplement and User's Manual Errata*: MPC604UMAD/AD (Motorola order #)
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations for each PowerPC implementation. These include the following:
  - — *PowerPC 601 RISC Microprocessor Hardware Specifications*: MPC601EC/D (Motorola order #)
  - — *PowerPC 602 RISC Microprocessor Hardware Specifications*: MPC602EC/D (Motorola order #)
  - — *PowerPC 603 RISC Microprocessor Hardware Specifications*: MPC603EC/D (Motorola order #)
  - — *PowerPC 603e RISC Microprocessor Family: PID6-603e Hardware Specifications*: MPC603EEC/D (Motorola order #)
  - — *PowerPC 603e RISC Microprocessor Family: PID7V-603e Hardware Specifications*: MPC603E7VEC/D (Motorola order #)
  - — *PowerPC 604 RISC Microprocessor Hardware Specifications*: MPC604EC/D (Motorola order #)
  - — *PowerPC 604e RISC Microprocessor Family: PID9V-604e Hardware Specifications*: MPC604E9VEC/D (Motorola order #)
- Technical Summaries—Each PowerPC implementation has a technical summary that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's user's manual. Technical summaries are available for the 601, 602, 603, 603e, 604, and 604e as well as the following:
  - — *PowerPC 620™ RISC Microprocessor Technical Summary*: MPC620/D (Motorola order #)

- *PowerPC Microprocessor Family: The Bus Interface for 32-Bit Microprocessors*: MPCBUSIF/AD (Motorola order #) provides a detailed functional description of the 60x bus interface, as implemented on the 601, 603, and 604 family of PowerPC microprocessors. This document is intended to help system and chipset developers by providing a centralized reference source to identify the bus interface presented by the 60x family of PowerPC microprocessors.
- *PowerPC Microprocessor Family: The Programmer's Reference Guide*: MPCPRG/D (Motorola order #) is a concise reference that includes the register summary, memory control model, exception vectors, and the PowerPC instruction set.
- *PowerPC Microprocessor Family: The Programmer's Pocket Reference Guide*: MPCPRGREF/D (Motorola order #): This foldout card provides an overview of the PowerPC registers, instructions, and exceptions for 32-bit implementations.
- Application notes—These short documents contain useful information about specific design issues useful to programmers and engineers working with PowerPC processors.
- Documentation for support chips—These include the following:
  - — *MPC105 PCI Bridge/Memory Controller User's Manual*: MPC105UM/AD (Motorola order #)
  - — *MPC106 PCI Bridge/Memory Controller User's Manual*: MPC106UM/AD (Motorola order #)

Additional literature on PowerPC implementations is being released as new processors become available. For a current list of PowerPC documentation, refer to the world-wide web at http://www.mot.com/powerpc/.

Conventions

This document uses the following notational conventions:

ACTIVE_HIGH Names for signals that are active high are shown in uppercase text without an overbar.

$\overline{\text{ACTIVE\_LOW}}$ A bar over a signal name indicates that the signal is active low—for example, $\overline{\text{ARTRY}}$ (address retry) and $\overline{\text{TS}}$ (transfer start). Active-low signals are referred to as asserted (active) when they are low and negated when they are high. Signals that are not active low, such as APARITY0–APARITY3 (address bus parity signals) and TT0–TT4 (transfer type signals) are referred to as asserted when they are high and negated when they are low.

mnemonics Instruction mnemonics are shown in lowercase bold.

OPERATIONS Address-only bus operations that are named for the instructions that generate them are identified in uppercase letters, for example, ICBI, SYNC, TLBSYNC, and EIEIO operations.

italics Italics indicate variable command parameters, for example, **bcctr***x*

0x0 Prefix to denote hexadecimal number

0b0 Prefix to denote binary number

rA, **r**B Instruction syntax used to identify a source GPR

rA|0 The contents of a specified GPR or the value 0.

rD Instruction syntax used to identify a destination GPR

frA, **fr**B, **fr**C Instruction syntax used to identify a source FPR

frD Instruction syntax used to identify a destination FPR

REG[FIELD] Abbreviations or acronyms for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.

x In certain contexts, such as a signal encoding, this indicates a don't care.

n Used to express an undefined numerical value.

Acronyms and Abbreviations

The Table i contains acronyms and abbreviations that are used in this document. Note that the meanings for some acronyms (such as SDR1 and XER) are historical, and the words for which an acronym stands may not be intuitively obvious.

Table i. Acronyms and Abbreviated Terms

Term	Meaning
ALU	Arithmetic logic unit
ASR	Address space register
BAT	Block address translation
BIST	Built-in self test
BIU	Bus interface unit
BHT	Branch history table
BPU	Branch processing unit
BTAC	Branch target address cache
BUID	Bus unit ID
COP	Common on-chip processor
CR	Condition register
CTR	Count register
DABR	Data address breakpoint register

Table i. Acronyms and Abbreviated Terms (Continued)

Term	Meaning
DAR	Data address register
DBAT	Data BAT
DEC	Decrementer (register)
DEQ	Decode queue
DISQ	Dispatch queue
DSISR	Register used for determining the source of a DSI exception
DTLB	Data translation look-aside buffer
EA	Effective address
EAR	External access register
ECC	Error checking and correction
FIFO	First-in, first out
FLQ	Finish load queue
FPR	Floating-point register
FPSCR	Floating-point status and control register
FPU	Floating-point unit
GPR	General-purpose register
HID0	Hardware implementation dependent (register) 0
IABR	Instruction address breakpoint register
IBAT	Instruction BAT
IEEE	Institute of Electrical and Electronics Engineers
ITLB	Instruction translation look-aside buffer
IU	Integer unit
JTAG	Joint Test Action Group
L2	Secondary cache
LR	Link register
LRU	Least recently used
LSB	Least-significant byte
lsb	Least-significant bit
LSU	Load/store unit
MCIU	Multiple-cycle integer unit
MESI	Modified/exclusive/shared/invalid—cache coherency protocol
MMCR*n*	Monitor mode control register *n*

Table i. Acronyms and Abbreviated Terms (Continued)

Term	Meaning
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSR	Machine state register
NaN	Not a number
No-Op	No operation
OEA	Operating environment architecture
PID	Processor identification tag
PLL	Phase-locked loop
PMC*n*	Performance monitor control (register) *n*
PMI	Performance monitor interrupt
PTE	Page table entry
PTEG	Page table entry group
PVR	Processor version register
RISC	Reduced instruction set computing/computer
ROB	Reorder buffer
RTL	Register transfer language
RWITM	Read with intent to modify
SCIU	Single-cycle integer unit
SDA	Sampled data address (register)
SDR1	Register that specifies the page table base address for virtual-to-physical address translation
SIA	Sampled instruction address (register)
SIMM	Signed immediate value
SLB	Segment look-aside buffer
SPR	Special-purpose register
SPRG*n*	Registers available for general purposes
SR	Segment register
SRR0	(Machine status) save/restore register 0
SRR1	(Machine status) save/restore register 1
TB	Time base register
TLB	Translation lookaside buffer
UIMM	Unsigned immediate value

Table i. Acronyms and Abbreviated Terms (Continued)

Term	Meaning
UISA	User instruction set architecture
VEA	Virtual environment architecture
XATC	Extended address transfer code
XER	Register used for indicating conditions such as carries and overflows for integer operations

Terminology Conventions

Table ii lists certain terms used in this manual that differ from the architecture terminology conventions.

Table ii. Terminology Conventions

The Architecture Specification	This Manual
Data storage interrupt (DSI)	DSI exception
Extended mnemonics	Simplified mnemonics
Instruction storage interrupt (ISI)	ISI exception
Interrupt*	Exception
Privileged mode (or privileged state)	Supervisor-level privilege
Problem mode (or problem state)	User-level privilege
Real address	Physical address
Relocation	Translation
Storage (locations)	Memory
Storage (the act of)	Access

* For a detailed discussion of how the terms interrupt and exception are used in this document, see the introduction to Chapter 4, "Exceptions."

Table iii describes instruction field notation conventions used in this manual.

Table iii. Instruction Field Conventions

The Architecture Specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D (respectively)
BF, BFA	**crf**D, **crf**S (respectively)
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	**fr**A, **fr**B, **fr**C, **fr**D, **fr**S (respectively)

Table iii. Instruction Field Conventions (Continued)

The Architecture Specification	Equivalent to:
FXM	CRM
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S (respectively)
SI	SIMM
U	IMM
UI	UIMM
/, //, ///	0...0 (shaded)

Chapter 1
Overview

This chapter provides an overview of the PowerPC 620™ microprocessor. It includes the following:

- A summary of 620 features
- Details about the 620 hardware implementation. This includes descriptions of the 620's execution units, cache implementation, memory management units (MMUs), and system interface.
- A description of the 620 execution model. This section includes information about the programming model, instruction set, exception model, and instruction timing.
- A description of the performance monitor facility

1.1 Overview

This section describes the features of the 620, provides a block diagram showing the major functional units, and describes briefly how those units interact.

The 620 is an implementation of the PowerPC™ family of reduced instruction set computer (RISC) microprocessors. The 620 implements the PowerPC architecture as it is specified for 64-bit addressing, which provides 64-bit effective addresses, integer data types of 8, 16, 32, and 64 bits, and floating-point data types of 32 and 64 bits (single-precision and double-precision). The 620 is software compatible with the 32-bit versions of the PowerPC microprocessor family.

The 620 is a superscalar processor capable of issuing four instructions simultaneously. As many as four instructions can finish execution in parallel. The 620 has six execution units that can operate in parallel:

- Floating-point unit (FPU)
- Branch processing unit (BPU)
- Load/store unit (LSU)
- Three integer units (IUs):
  - — Two single-cycle integer units (SCIUs)
  - — One multiple-cycle integer unit (MCIU)

This parallel design, combined with the PowerPC architecture's specification of uniform instructions that allows for rapid execution times, yields high efficiency and throughput. The 620's rename buffers, reservation stations, dynamic branch prediction, and completion unit increase instruction throughput, guarantee in-order completion, and ensure a precise exception model. (Note that the PowerPC architecture specification refers to all exceptions as interrupts.)

The 620 has separate memory management units (MMUs) and separate 32-Kbyte on-chip caches for instructions and data. The 620 implements a 128-entry, two-way set-associative translation lookaside buffer (TLB) for instructions and data, and provides support for demand-paged virtual memory address translation and variable-sized block translation. The TLB and the cache use least-recently used (LRU) replacement algorithms.

The 620 has a 40-bit address bus, and can be configured with either a 64- or 128-bit data bus. The 620 interface protocol allows multiple masters to compete for system resources through a central external arbiter. Additionally, on-chip snooping logic maintains data cache coherency for multiprocessor applications. The 620 supports single-beat and burst data transfers for memory accesses and memory-mapped I/O accesses.

The 620 processor core uses an advanced, 2.5-V CMOS process technology, and is compatible with 3.3-V CMOS devices.

1.1.1 PowerPC 620 Microprocessor Features

This section summarizes features of the 620's implementation of the PowerPC architecture. Major features of the 620 are as follows:

- High-performance, superscalar microprocessor
  - — As many as four instructions can be issued per clock
  - — As many as six instructions can start executing per clock (including three integer instructions)
  - — Single clock cycle execution for most instructions
- Six independent execution units and two register files
  - — BPU featuring dynamic branch prediction
    - – Speculative execution through four branches
    - – 256-entry fully-associative branch target address cache (BTAC)
    - – 2048-entry branch history table (BHT) with two bits per entry indicating four levels of prediction—not-taken, strongly not-taken, taken, strongly taken
  - — Two single-cycle IUs (SCIUs) and one multiple-cycle IU (MCIU)
    - – Instructions that execute in the SCIU take one cycle to execute; most instructions that execute in the MCIU take multiple cycles to execute.
    - – Each SCIU has a two-entry reservation station to minimize stalls.

-
  -
    - The MCIU has a two-entry reservation station and provides early exit (three cycles) for 16 x 32-bit and overflow operations
    - Thirty-two GPRs for integer operands
    - Eight rename buffers for GPRs
  - Three-stage floating-point unit (FPU)
    - Fully IEEE 754-1985 compliant FPU for both single- and double-precision operations
    - Supports non-IEEE mode for time-critical operations
    - Fully pipelined, single-pass double-precision design
    - Hardware support for denormalized numbers
    - Two-entry reservation station to minimize stalls
    - Thirty-two 64-bit FPRs for single- or double-precision operands
    - Eight rename buffers for FPRs
  - Load/store unit (LSU)
    - Three-entry reservation station to minimize stalls
    - Single-cycle, pipelined cache access
    - Dedicated adder that performs EA calculations
    - Performs alignment and precision conversion for floating-point data
    - Performs alignment and sign extension for integer data
    - Five-entry pending load queue that provides load/store address collision detection
    - Five-entry finished store queue
    - Six-entry completed store queue
    - Supports both big- and little-endian modes
- Rename buffers
  - Eight GPR rename buffers
  - Eight FPR rename buffers
  - Sixteen condition register (CR) rename buffers

  The 620 rename buffers are described in Section 1.2.1.6, "Rename Buffers."
- Completion unit
  - Retires an instruction from the 16-entry reorder buffer when all instructions ahead of it have been completed and the instruction has finished execution
  - Guarantees sequential programming model (precise exception model)
  - Monitors all dispatched instructions and retires them in order

- — Tracks unresolved branches and removes speculatively executed, dispatched, and fetched instructions if branch is mispredicted
- — Retires as many as four instructions per clock

- Separate on-chip instruction and data caches (Harvard architecture)
  - — 32-Kbyte, eight-way set-associative instruction and data caches; data cache is 2-way interleaved.
  - — LRU replacement algorithm
  - — 64-byte (sixteen word) cache block size
  - — Physically indexed; physical tags
  - — Cache write-back or write-through operation programmable on a per page or per block basis
  - — Instruction cache can provide four instructions per clock; data cache can provide two words per clock.
  - — Caches can be disabled in software
  - — Parity checking performed on both caches
  - — Data cache coherency (MESI) maintained in hardware
  - — Interprocessor broadcast of cache control instructions
  - — Instruction cache coherency maintained in software
- On-chip L2 cache interface
  - — L2 cache is a unified instruction and data secondary cache with ECC.
  - — L2 cache is direct-mapped, physically-indexed, and physically-tagged.
  - — L2 data cache is inclusive of L1; L2 instruction cache is not inclusive of L1.
  - — L2 cache capacity is configurable from 1 Mbyte to 128 Mbyte.
  - — Independent user-configurable PLL provides L2 interface clock.
  - — L2 cache interface supports single-, double-, triple-, and quad-register synchronous SRAMs.
  - — L2 cache interface supports CMOS SRAMs.
  - — Supports direct connection of two SRAM banks
  - — Supports direct connection of coprocessor
- Separate memory management units (MMUs) for instructions and data
  - — Address translation facilities for 4-Kbyte page size, variable block size, and 256-Mbyte segment size
  - — Independent 64-entry fully-associative effective-to-physical address translation (EPAT) cache with invalid-first replacement algorithm for instructions and data
  - — Unified instruction and data translation lookaside buffer (TLB)
  - — TLB is 128-entry and two-way set-associative

  - 20-entry CAM segment lookaside buffer (SLB) with FIFO replacement algorithm
  - Sixteen segment registers that provide support for 32-bit memory management
  - SLB, TLB, and EPAT cache miss handling performed by 620 hardware
  - Hardware update of page frame table referenced and changed bits
  - Hardware broadcast of TLB and control instructions
  - Separate IBATs and DBATs (four each) also defined as SPRs
  - 64-bit effective addressing
  - 80-bit virtual addressing
  - 40-bit physical memory address for up to one terabyte
- Bus interface
  - Selectable processor-to-bus clock frequency ratios (2:1, 3:1, and 4:1)
  - A 64- and 128-bit split-transaction external data bus with burst transfers
  - Explicit address and data bus tagging
  - Pended (split) read protocol
  - Pipelined snoop response, fixed boot-time latency
  - 620 bus is crossbar compatible
  - Additional signals and signal redefinition for direct-store operations
- Multiprocessing support
  - Hardware-enforced, four-state cache coherency protocol (MESI) for data cache. Bits are provided in the instruction cache to indicate only whether a cache block is valid or invalid.
  - Data cache coherency for L1 and L2, and external L3 cache is fully supported by 620 hardware.
  - Snoop operations take priority over processor access to L1 and L2 cache.
  - Instruction cache coherency is software controlled.
  - Load/store with reservation instruction pair is provided for atomic memory references, semaphores, and other multiprocessor operations.
- Power requirements
  - Operating voltage is 2.5 V for the processor core, and 3.3 V for I/O drivers.
- Performance monitor can be used to help in debugging system designs and improving software efficiency, especially in multiprocessor systems.
- In-system testability and debugging features are provided through JTAG boundary-scan capability.

1.1.2 Block Diagram

Figure 1-1 provides a block diagram showing features of the 620. Note that this is a conceptual block diagram intended to show the basic features rather than an attempt to show how these features are physically implemented on the chip.

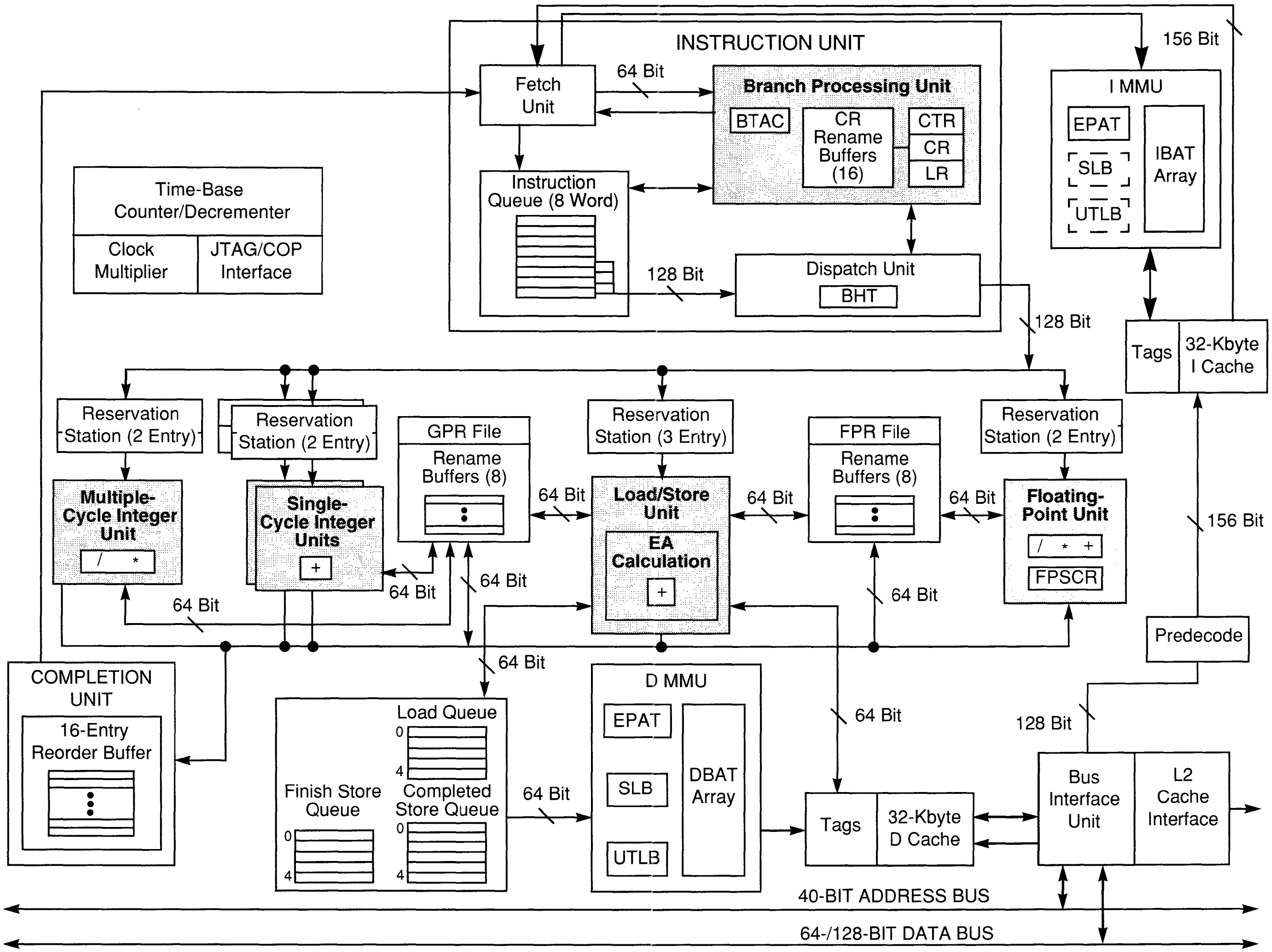


Figure 1-1. PowerPC 620 Microprocessor Block Diagram

1.2 PowerPC 620 Microprocessor Hardware Implementation

This section provides an overview of the 620's hardware implementation, including descriptions of the functional units, shown in Figure 1-2, the cache implementation, MMU, and the system interface.

Note that Figure 1-2 provides a more detailed block diagram than that presented in Figure 1-1—showing the additional data paths that contribute to the improved efficiency in instruction execution and more clearly indicating the relationships between execution units and their associated register files.

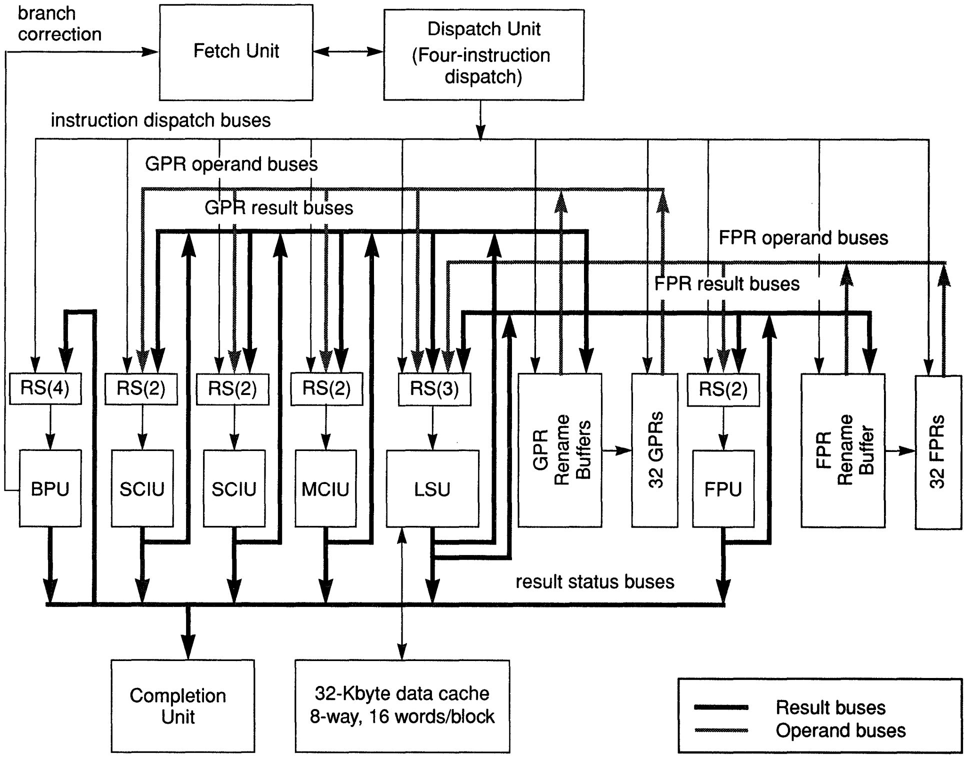


Figure 1-2. Block Diagram—Internal Data Paths

1.2.1 Instruction Flow

Several units on the 620 ensure the proper flow of instructions and operands and guarantee the correct update of the architectural machine state. These units include the following:

- Predecode unit—Provides logic to decode instructions and determine what resources are required for execution.
- Fetch unit—Using the next sequential address or the address supplied by the BPU when a branch is predicted or resolved, the fetch unit supplies instructions to the eight-word instruction queue.
- Dispatch unit—The dispatch unit dispatches instructions to the appropriate execution unit. During dispatch, operands are provided to the execution unit (or reservation station) from the register files, rename buffers, and result buses.
- Branch processing unit (BPU)—In addition to providing the fetcher with predicted target instructions when a branch is predicted (and a mispredict-recovery address if a branch is incorrectly predicted), the BPU executes all condition register logical and flow control instructions.
- Completion unit—The completion unit retires executed instructions in program order and controls the updating of the architectural machine state.

1.2.1.1 Predecode Unit

The instruction predecode unit provides the logic to decode the instructions and categorize the resources that will be used, source operands, destination registers, execution registers, and other resources required for execution. The instruction stream is predecoded on its way from the bus interface unit to the instruction cache.

1.2.1.2 Fetch Unit

The fetch unit provides instructions to the four-entry (8-instruction) instruction queue by accessing the on-chip instruction cache. Typically, the fetch unit continues fetching sequentially as many as four instructions at a time.

The address of the next instruction to be fetched is determined by several conditions, which are prioritized as follows:

1. Detection of an exception. Instruction fetching begins at the exception vector.
2. The BPU recovers from an incorrect prediction when a branch instruction is in the execute stage. Undispatched instructions are flushed and fetching begins at the correct target address.
3. The BPU recovers from an incorrect prediction when a branch instruction is in the dispatch stage. Undispatched instructions are flushed and fetching begins at the correct target address.
4. As a cache block is fetched, the branch target address cache (BTAC) and the branch history table (BHT) are searched with the fetch address. If it is found in the BTAC, the target address from the BTAC is the first candidate for being the next fetch address.

5. If none of the previous conditions exist, the instruction is fetched from the next sequential address.

1.2.1.3 Dispatch Unit

The dispatch unit provides the logic for dispatching the predecoded instructions to the appropriate execution unit. For many branch instructions, these decoded instructions along with the bits in the BHT, are used during the decode stage for branch correction. The dispatch logic also resolves unconditional branch instructions and predicts conditional branch instructions using the branch decode logic, BHT, and values in the count register (CTR).

The 2048-entry BHT provides two bits per entry, indicating four levels of dynamic prediction—strongly not-taken, not-taken, taken, and strongly taken. The history of a branch's direction is maintained in these two bits. For example, each time a branch is taken, the value is incremented (with a maximum value of three meaning strongly-taken); when it is not taken, the bit value is decremented (with a minimum value of zero meaning strongly not-taken). If the current value predicts taken and the next branch is taken again, the BHT entry then predicts strongly taken. If the following branch is not taken, the BHT then predicts taken.

The dispatch logic also allocates each instruction to the appropriate execution unit. A reorder buffer entry in the completion unit is allocated for each instruction, and data (or resource) dependency is checked between the instructions in the dispatch queue. The rename buffers are searched for the operands as the operands are fetched from the register file. Operands that are written by other instructions ahead of this one in the dispatch queue are given the tag of that instruction's rename buffer; otherwise, the rename buffer or register file supplies either the operand or a tag. As instructions are dispatched, the fetch unit is notified that the dispatch queue can be updated with more instructions.

1.2.1.4 Branch Processing Unit (BPU)

The BPU is used for branch instructions and condition register logical operations. All branches, including unconditional branches, are placed in reservation stations until conditions are resolved and they can be executed. At that point, branch instructions are executed in order—the completion unit is notified whether the prediction was correct.

The BPU also executes condition register logical instructions, which flow through the reservation station like the branch instructions.

1.2.1.5 Completion Unit

The completion unit retires executed instructions from the reorder buffer in the completion unit and updates register files and control registers. The completion unit recognizes exception conditions and discards any operations being performed on subsequent instructions in program order. The completion unit can quickly remove instructions from a mispredicted branch, and the dispatch unit begins dispatching from the correct path.

The instruction is retired from the reorder buffer when it has finished execution and all instructions ahead of it have been completed. The instruction's result is written into the appropriate register file and is removed from the rename buffers at or after completion. At completion, the 620 also updates any other resource affected by this instruction. Several instructions can complete simultaneously. Most exception conditions are recognized at completion time.

1.2.1.6 Rename Buffers

To avoid contention for a given register location, the 620 provides rename registers for storing instruction results before the completion unit commits them to the architected register. Eight rename registers are provided for the GPRs, eight for the FPRs, and sixteen for the condition register. GPRs, FPRs, and the condition register are described in Section 1.3.2, "Registers and Programming Model,"

When the dispatch unit dispatches an instruction to its execution unit, it allocates a rename register for the results of that instruction. The dispatch unit also provides a tag to the execution unit identifying the result that should be used as the operand. When the proper result is returned to the rename buffer, it is latched into the reservation station. When all operands are available in the reservation station, execution can begin.

The completion unit does not transfer instruction results from the rename registers to the registers until any speculative branch conditions preceding it in the completion queue are resolved and the instruction itself is retired from the completion queue without exceptions. If a speculatively executed branch is found to have been incorrectly predicted, the speculatively executed instructions following the branch are flushed from the completion queue and the results of those instructions are flushed from the rename registers.

1.2.2 Execution Units

The following sections describe the 620's arithmetic execution units— two single-cycle IUs, multiple-cycle IU, and FPU. When the reservation station sees the proper result being written back, it will grab it directly from one of the result buses. Once all operands are in the reservation station for an instruction, it is eligible to be executed. Reservation stations temporarily store dispatched instructions that cannot be executed until all of the source operands are valid.

1.2.2.1 Integer Units (IUs)

The two single-cycle IUs (SCIUs) and one multiple-cycle IU (MCIU) execute all integer instructions. These are shown in Figure 1-1 and Figure 1-2. The results generated by the IUs are put on the result buses that are connected to the appropriate reservation stations and rename buffers. Each IU has a two-entry reservation station to reduce stalls. The reservation station can receive instructions from the dispatch unit and operands from the GPRs, the rename buffers, or the result buses.

Each SCIU consists of three single-cycle subunits—a fast adder/comparator, a subunit for logical operations, and a subunit for performing rotates, shifts, and count-leading-zero operations. These subunits handle all one-cycle arithmetic instructions; only one subunit can execute an instruction at a time.

The MCIU consists of a 64-bit integer multiplier/divider. The MCIU executes **mfspr** and **mtspr** instructions, which are used to read and write special-purpose registers. The MCIU can execute an **mtspr** or **mfspr** instruction at the same time that it executes a multiply or divide instruction. These instructions are allowed to complete out of order.

1.2.2.2 Floating-Point Unit (FPU)

The FPU, shown in Figure 1-1 and Figure 1-2, is a single-pass, double-precision execution unit; that is, both single- and double-precision operations require only a single pass, with a latency of three cycles.

As the dispatch unit issues instructions to the FPU's two reservation stations, source operand data may be accessed from the FPRs, the floating-point rename buffers, or the result buses. Results in turn are written to the floating-point rename buffers and to the reservation stations and are made available to subsequent instructions. The three reservation stations provided by the FPU support out-of-order execution of floating-point instructions.

1.2.2.3 Load/Store Unit (LSU)

The LSU, shown in Figure 1-1 and Figure 1-2, transfers data between the data cache and the result buses, which route data to other execution units. The LSU supports the address generation and handles any alignment for transfers to and from system memory. The LSU also supports cache control instructions and load/store multiple/string instructions.

The LSU includes a 64-bit adder dedicated for EA calculation. Data alignment logic manipulates data to support aligned or misaligned transfers with the data cache. The LSU's load and store queues are used to buffer instructions that have been executed and are waiting to be completed. The queues are used to monitor data dependencies generated by data forwarding and out-of-order instruction execution ensuring a sequential model.

The LSU allows load instructions to precede store instructions in the reservation stations. Data dependencies resulting from the out-of-order execution of loads before stores to addresses with the same low-order 12 bits in the effective address are resolved when the store instruction is completed. If an out-of-order load operation is found to have an address that matches a previous store, the instruction pipeline is flushed, and the load instruction will be refetched and re-executed.

The LSU does not allow the following operations to be speculatively performed on unresolved branches:

- Store operations
- Loading of noncacheable data or cache miss operations
- Loading from direct-store segments

1.2.3 Memory Management Units (MMUs)

The primary functions of the MMUs are to translate logical (effective) addresses to physical addresses for memory accesses, I/O accesses (most I/O accesses are assumed to be memory-mapped), and direct-store accesses, and to provide access protection on blocks and pages of memory.

The PowerPC MMUs and exception model support demand-paged virtual memory. Virtual memory management permits execution of programs larger than the size of physical memory; demand-paged implies that individual pages are loaded into physical memory from system memory only when they are first accessed by an executing program.

The hashed page table is a variable-sized data structure that defines the mapping between virtual page numbers and physical page numbers. The page table size is a power of 2, and its starting address is a multiple of its size.

Address translations are enabled by setting bits in the MSR—MSR[IR] enables instruction address translations and MSR[DR] enables data address translations.

The 620's MMUs support up to one heptabyte (2^{80}) of virtual memory and one terabyte (2^{40}) of physical memory. The MMUs support block address translations, direct-store segments, and page translation of memory segments. Referenced and changed status are maintained by the processor for each page to assist implementation of a demand-paged virtual memory system.

Separate but identical translation logic is implemented for data accesses and for instruction accesses. The 620 implements a two-stage translation cache mechanism; the first stage consists of independent 64-entry content-addressable EPATs for instructions and data, and the second stage consists of a shared 128-entry, two-way set-associative translation lookaside buffer (TLB). If a TLB miss occurs during the second-stage address translation, memory segment lookup is assisted by a 20-entry content-addressable segment lookaside buffer (SLB). The operating environment architecture (OEA) defines an additional, optional bridge that allows 64-bit implementations to use a simpler memory management model to access 32-bit effective address space. For processors that implement the address translation portion of the bridge, segment descriptors take the form of the STEs defined for 64-bit MMUs; however, only 16 STEs are required to define the entire 4-Gbyte address space. Like 32-bit implementations, the effective address space is entirely defined by 16 contiguous 256-Mbyte segment descriptors. Rather than using the set of 16, 32-bit segment registers as is defined for the 32-bit MMU, the 16 STEs are implemented and are maintained in 16 SLB entries. For more information on the optional bridge, refer to, *PowerPC Microprocessor Family: The Programming Environments*, Rev. 1.

1.2.4 Cache Implementation

The PowerPC architecture does not define hardware aspects of cache implementations. For example, the 620 implements separate data and instruction caches (Harvard architecture), while other processors may use a unified cache, or no cache at all. The PowerPC architecture defines the unit of coherency as a cache block, which for the 620 is a 64-byte (sixteen-word) line.

PowerPC implementations can control the following memory access modes on a page or block basis:

- Write-back/write-through mode
- Caching-inhibited mode
- Memory coherency
- Guarded memory (prevents access for out-of-order execution)

1.2.4.1 Instruction Cache

The 620's 32-Kbyte, eight-way set-associative instruction cache is physically indexed. Within a single cycle, the instruction cache provides up to four instructions. Instruction cache coherency is not maintained by hardware.

The PowerPC architecture defines a special set of instructions for managing the instruction cache. The instruction cache can be invalidated entirely or on a cache-block basis. The instruction cache can be disabled and invalidated by setting the HID0[16] and HID0[20] bits, respectively.

1.2.4.2 Data Cache

The 620's data cache is a 32-Kbyte, eight-way set-associative cache. It is a physically-indexed, nonblocking, write-back cache with hardware support for reloading on cache misses. Within one cycle, the data cache provides double-word access to the LSU.

The data cache tags are dual-ported, so the process of snooping does not affect other transactions on the system interface. If a snoop hit occurs in the same cache set as a load or store access, the LSU is blocked internally for one cycle to allow the 16-word block of data to be copied to the write-back buffer.

The 620 data cache supports the four-state MESI (modified/exclusive/shared/invalid) protocol to ensure cache coherency.

These four states indicate the state of the cache block as follows:

- Modified (M)—The cache block is modified with respect to system memory; that is, data for this address is valid only in the cache and not in system memory.
- Exclusive (E)—This cache block holds valid data that is identical to the data at this address in system memory. No other cache has this data.
- Shared (S)—This cache block holds valid data that is identical to this address in system memory and at least one other caching device.
- Invalid (I)—This cache block does not hold valid data.

Like the instruction cache, the data cache can be invalidated all at once or on a per cache block basis. The data cache can be disabled and invalidated by setting the HID0[17] and HID0[21] bits, respectively.

Each cache line contains 16 contiguous words from memory that are loaded from a 16-word boundary (that is, bits A[58–63] of the logical addresses are zero); thus, a cache line never crosses a page boundary. Accesses that cross a page boundary can incur a performance penalty.

The organization of the cache is shown in Figure 1-3.

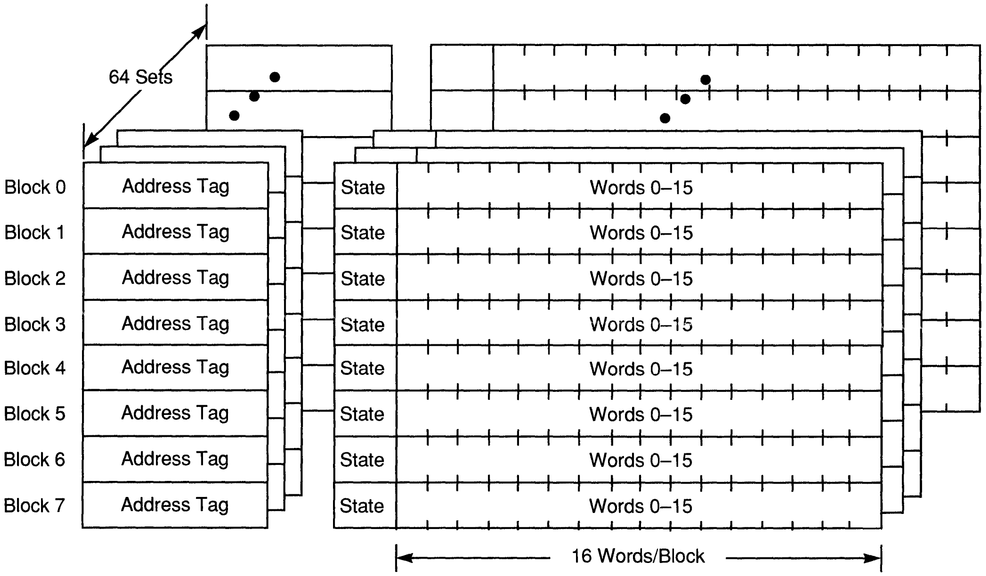


Figure 1-3. Cache Unit Organization

1.2.5 Level 2 (L2) Cache Interface

The 620 provides an integrated L2 cache controller that supports L2 configurations from 1 Mbyte to 128 Mbyte, using the same block size (64 bytes) as the internal L1 caches. The 620's L2 cache interface supports a direct-mapped, error-correction-code (ECC) protected, unified instruction and data secondary cache that uses single- and double-register synchronous static RAMs. The L2 cache interface supports a wide variety of static RAM access speeds by means of a boot-time configurable subsynchronous interface that is configurable for either CMOS or HSTL logic levels. An external coprocessor can also be connected to the 620 through the L2 cache interface.

The L2 cache interface generates 9 bits of ECC for the 128 bits of data in a cache block, and 6 bits of ECC for the tag and coherency state of the block. The ECC allows the correction of single-bit errors, and the detection of double-bit errors. Uncorrectable errors detected by the L2 cache interface generates a machine check exception. The ECC capability of the L2 cache interface can be configured in three modes—always-corrected mode, never-corrected mode, and automatic mode. In always-corrected mode, ECC is generated for write operations, and always corrected on read operations, resulting in constant L2 read access latency. In never-corrected mode, ECC generation, checking, and correction are disabled. In the automatic mode, ECC is generated during write operations, and read operations are corrected only when errors are detected, thereby increasing read latency only when correctable errors are detected.

1.2.6 System Interface/Bus Interface Unit (BIU)

The 620 provides a versatile bus interface that allows a wide variety of system design options. The interface includes a 144-bit data bus (128 bits of data and 16 bits of parity), a 43-bit address bus (40 bits of address and 3 bits of parity), and sufficient control signals to allow for a variety of system-level optimizations. The 620 uses one-beat, four-beat, and eight-beat data transactions (depending on whether the 620 is configured with a 64- or 128-bit data bus), although it is possible for other bus participants to perform longer data transfers. The 620 clocking structure supports processor-to-bus clock ratios of 2:1, 3:1, and 4:1 as described in Section 1.2.7, "Clocking."

The system interface is specific for each PowerPC processor implementation. The 620 system interface is shown in Figure 1-4.

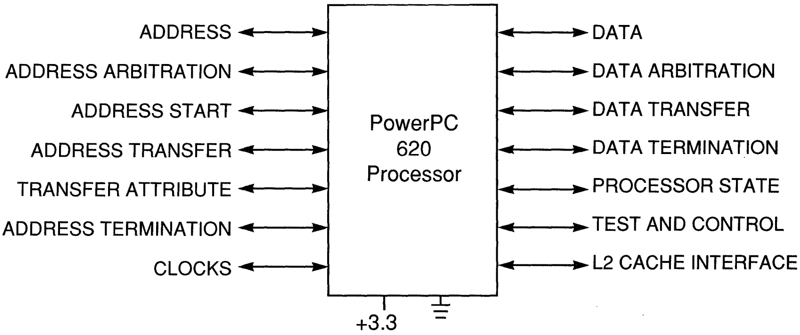


Figure 1-4. System Interface

Four-beat (or eight-beat, if in 64-bit data bus mode) burst-read memory operations that load a 16-word cache block into one of the on-chip caches are the most common bus transactions in typical systems, followed by burst-write memory operations, direct-store operations, and single-beat (noncacheable or write-through) memory read and write operations. Additionally, there can be address-only operations, data-only operations, variants of the burst and single-beat operations (global memory operations that are snooped and atomic memory operations, for example), and address retry activity.

Memory accesses can occur in single-beat or four-beat burst data transfers. The address and data buses are independent for memory accesses to support pipelining and split transactions, and all bus operations are explicitly tagged through the use of 8-bit tags for addresses and data. The 620 supports bus pipelining and out-of-order split-bus transactions.

Typically, memory accesses are weakly-ordered. Sequences of operations, including load and store string/multiple instructions, do not necessarily complete in the same order in which they began—maximizing the efficiency of the bus without sacrificing coherency of the data. The 620 allows load operations to precede store operations (except when a dependency exists). In addition, the 620 provides a separate queue for snoop push operations so these operations can access the bus ahead of previously queued operations. The 620 dynamically optimizes run-time ordering of load/store traffic to improve overall performance.

Access to the system interface is granted through an external arbitration mechanism that allows devices to compete for bus mastership. This arbitration mechanism is flexible, allowing the 620 to be integrated into systems that use various fairness and bus-parking procedures to avoid arbitration overhead. Additional multiprocessor support is provided through coherency mechanisms that provide snooping, external control of the on-chip caches and TLBs, and support for a secondary cache. The PowerPC architecture provides the load/store with reservation instruction pair (**lwarx/stwcx.**) for atomic memory references and other operations useful in multiprocessor implementations.

The following sections describe the 620 bus support for memory and direct-store operations. Note that some signals perform different functions depending upon the addressing protocol used.

1.2.6.1 Memory Accesses

Memory accesses allow transfer sizes of 8, 16, 24, 32, 64, or 128 bits in one bus clock cycle. Data transfers occur in either single-beat, four-beat, or eight-beat burst transactions. A single-beat transaction transfers as much as 128 bits. Single-beat transactions are caused by noncached accesses that access memory directly (that is, reads and writes when caching is disabled, caching-inhibited accesses, and stores in write-through mode). Burst transactions, which always transfer an entire cache block (64 bytes), are initiated when a block in the cache is read from or written to memory. Additionally, the 620 supports address-only transactions used to invalidate entries in other processors' TLBs and caches, and data-only transactions in which modified data is provided by a snooping device during a read operation to both the bus master and the memory system.

Typically I/O accesses are performed using the same protocol as memory accesses.

1.2.6.2 Signals

The 620's signals are grouped as follows:

- Address arbitration signals—The 620 uses these signals to arbitrate for address bus mastership.
- Address transfer start signals—These signals indicate that a bus master has begun a transaction on the address bus.
- Address transfer signals—These signals, which consist of the address bus, address parity, and address parity error signals, are used to transfer the address and to ensure the integrity of the transfer.
- Transfer attribute signals—These signals provide information about the type of transfer, such as the transfer size and whether the transaction is bursted, write-through, or caching-inhibited.
- Address transfer termination signals—These signals are used to acknowledge the end of the address phase of the transaction. They also indicate whether a condition exists that requires the address phase to be repeated.
- Data arbitration signals—The 620 uses these signals to arbitrate for data bus mastership.
- Data transfer signals—These signals, which consist of the data bus, data parity, and data parity error signals, are used to transfer the data and to ensure the integrity of the transfer.
- Data transfer termination signals—Data termination signals are required after each data beat in a data transfer. In a single-beat transaction, the data termination signals also indicate the end of the tenure, while in burst accesses, the data termination signals apply to individual beats and indicate the end of the tenure only after the final data beat. They also indicate whether a condition exists that requires the data phase to be repeated.
- System status signals—These signals include the interrupt signal, checkstop signals, and both soft reset and hard reset signals. These signals are used to interrupt and, under various conditions, to reset the processor.
- Processor state signal—This signal is used to indicate the state of the reservation coherency bit.
- Miscellaneous signals—These signals are used in conjunction with such resources as secondary caches and the time base facility.
- COP interface signals—The common on-chip processor (COP) unit is the master clock control unit and it provides a serial interface to the system for performing built-in self test (BIST) on all internal memory arrays.
- Clock signals—These signals determine the system clock frequency. These signals can also be used to synchronize multiprocessor systems.

NOTE

A bar over a signal name indicates that the signal is active low—for example, $\overline{\mathrm{DBG}}$ (data bus grant) and $\overline{\mathrm{EATS}}$ (early address transfer start). Active-low signals are referred to as asserted (active) when they are low and negated when they are high. Signals that are not active low, such as AP[0–2] (address bus parity signals) and DT[0–7] (data tag signals) are referred to as asserted when they are high and negated when they are low.

1.2.6.3 Signal Configuration

Figure 1-5 illustrates the logical pin configuration of the 620, showing how the signals are grouped.

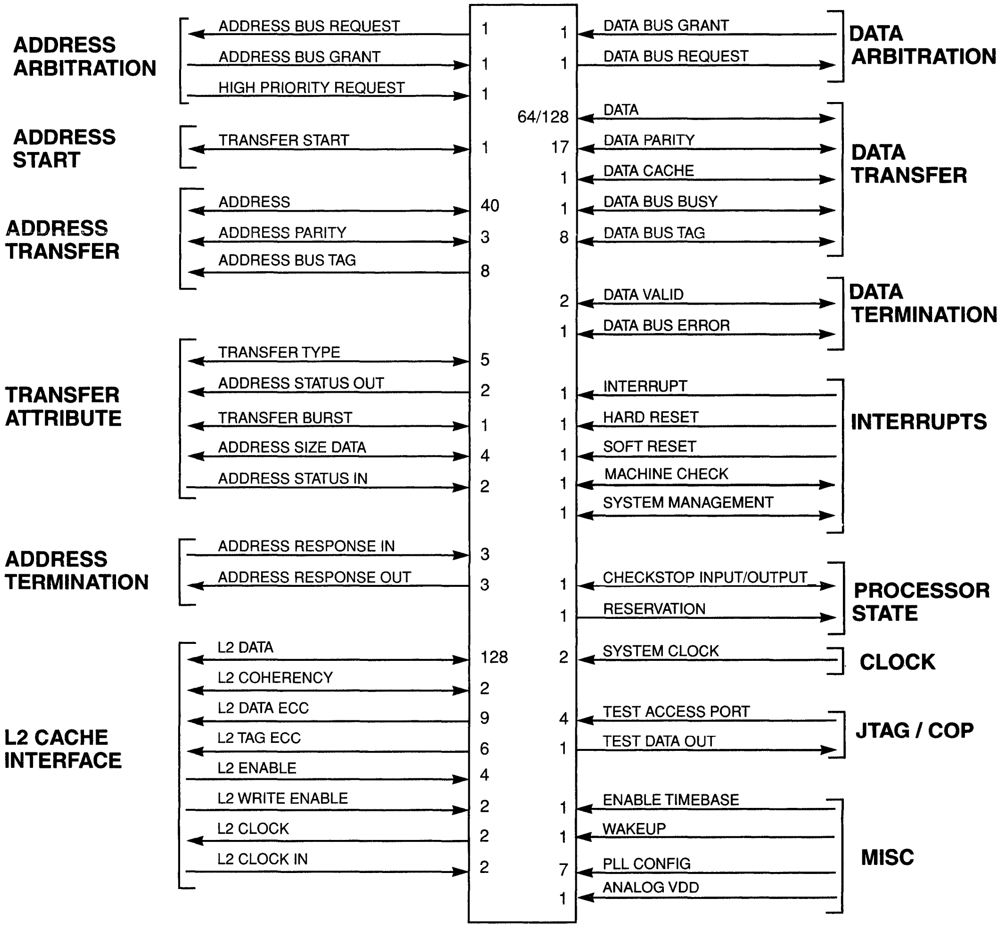


Figure 1-5. PowerPC 620 Microprocessor Signal Groups

1.2.7 Clocking

The 620 has a phase-locked loop (PLL) that generates the internal processor clock. The input, or reference signal, to the PLL is the bus clock. The feedback in the PLL guarantees that the processor clock is phase locked to the bus clock, regardless of process variations, temperature changes, or parasitic capacitances. The PLL also ensures a 50% duty cycle for the processor clock.

The 620 supports the following processor-to-bus clock frequency ratios—2:1, 3:1, and 4:1, although not all ratios are available for all frequencies. For more information about the configuration of the PLL, refer to the 620 hardware specifications.

1.3 PowerPC 620 Microprocessor Execution Model

This section describes the following characteristics of the 620's execution model:

- The PowerPC architecture
- The 620 register set and programming model
- The 620 instruction set
- The 620 exception model
- Instruction timing on the 620

1.3.1 Levels of the PowerPC Architecture

The PowerPC architecture is derived from the IBM POWER (Performance Optimized with Enhanced RISC) architecture. The PowerPC architecture shares the benefits of the POWER architecture optimized for single-chip implementations. The architecture design facilitates parallel instruction execution and is scalable to take advantage of future technological gains.

The PowerPC architecture consists of the following layers, and adherence to the PowerPC architecture can be measured in terms of which of the following levels of the architecture is implemented. For example, if a processor adheres to the virtual environment architecture, it is assumed that it meets the user instruction set architecture specification:

- PowerPC user instruction set architecture (UISA)—The UISA defines the level of the architecture to which user-level software must conform. The UISA defines the base user-level instruction set, user-level registers, data types, memory conventions, and the memory and programming models seen by application programmers. Note that the PowerPC architecture refers to user level as problem state.
- PowerPC virtual environment architecture (VEA)—The VEA, which is the smallest component of the PowerPC architecture, defines additional user-level functionality that falls outside typical user-level software requirements. The VEA describes the memory model for an environment in which multiple processors or other devices can access external memory, and defines aspects of the cache model and cache control

instructions from a user-level perspective. The resources defined by the VEA are particularly useful for managing resources in an environment in which other processors and other devices can access external memory.

Implementations that conform to the PowerPC VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

- PowerPC operating environment architecture (OEA)—The OEA defines supervisor-level resources typically required by an operating system. The OEA defines the PowerPC memory management model, supervisor-level registers, and the exception model. Note that the PowerPC architecture refers to the supervisor level as privileged state.

  Implementations that conform to the PowerPC OEA also conform to the PowerPC UISA and VEA.

The 620 complies with all three levels of the PowerPC architecture for 64-bit processors. In addition, the 620 implements the optional bridge; refer to *PowerPC Microprocessor Family: The Programming Environments*, Rev. 1 for more information. Note that the PowerPC architecture defines additional instructions for 64-bit data types. PowerPC processors are allowed to have implementation-specific features that fall outside, but do not conflict with, the PowerPC architecture specification. For example, the performance monitor is an implementation-specific feature of the 620.

The 620 is a high-performance, superscalar PowerPC implementation of the PowerPC architecture. Like other PowerPC processors, it adheres to the PowerPC architecture specifications but also has additional features not defined by the architecture. These features do not affect software compatibility. The PowerPC architecture allows optimizing compilers to schedule instructions to maximize performance through efficient use of the PowerPC instruction set and register model. The multiple, independent execution units in the 620 allow compilers to maximize parallelism and instruction throughput. Compilers that take advantage of the flexibility of the PowerPC architecture can additionally optimize instruction processing of the PowerPC processors.

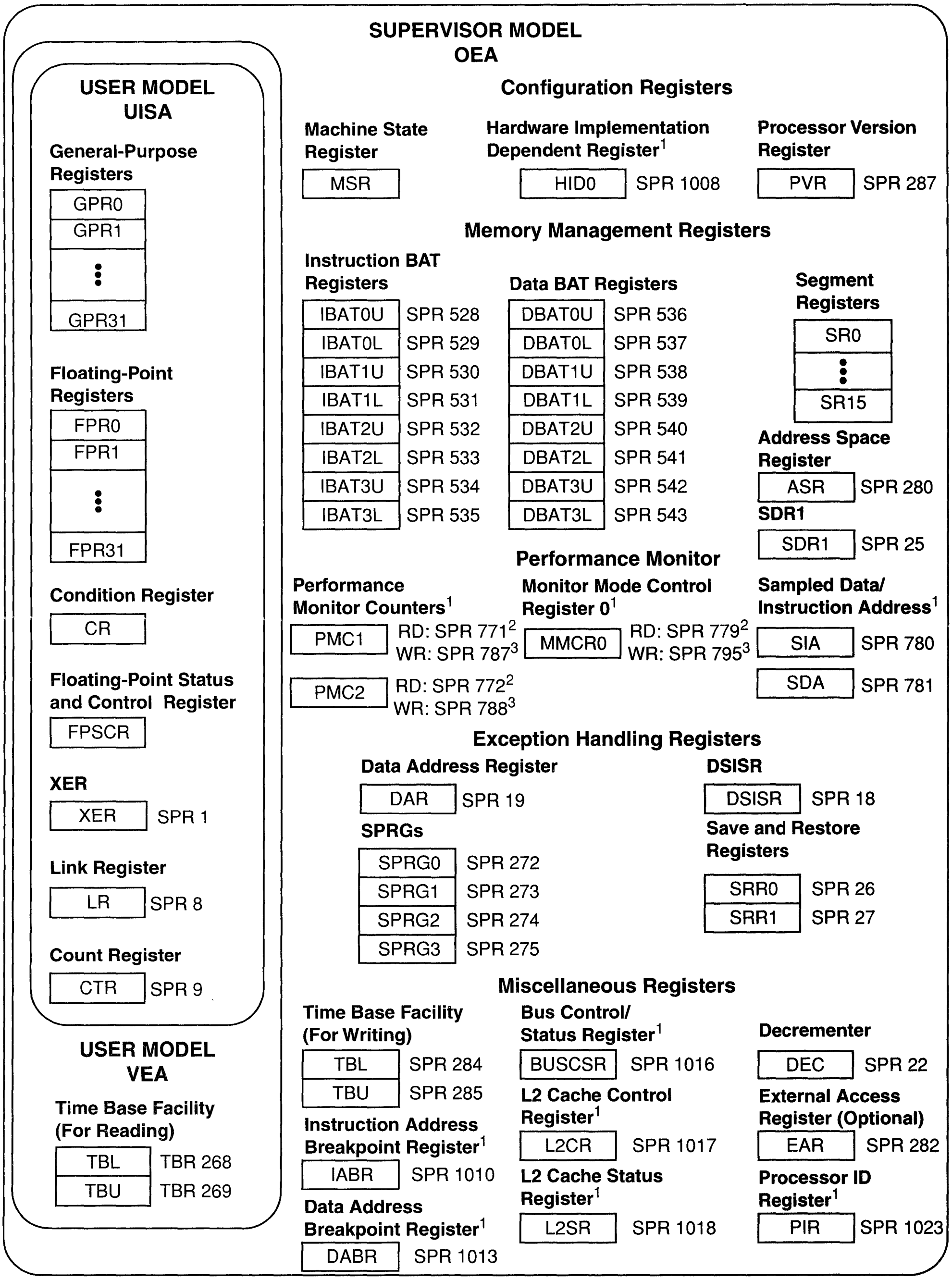
1.3.2 Registers and Programming Model

The PowerPC architecture defines register-to-register operations for most computational instructions. Source operands for these instructions are accessed from the registers or are provided as immediate values embedded in the instruction opcode. The three-register instruction format allows specification of a target register distinct from the two source operands. Load and store instructions transfer data between registers and memory.

During normal execution, a program can access the registers, shown in Figure 1-6, depending on the program's access privilege (supervisor or user, determined by the privilege-level (PR) bit in the machine state register (MSR)). Note that registers such as the general-purpose registers (GPRs) and floating-point registers (FPRs) are accessed through operands that are part of the instructions. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions) or implicitly as the part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

The numbers to the right of the SPRs indicate the number that is used in the syntax of the instruction operands to access the register.

Figure 1-6 shows the registers implemented in the 620, indicating those that are defined by the PowerPC architecture for 64-bit processors and those that are 620-specific.



1. 620-specific—not defined by the PowerPC architecture 2. RD: read only 3. WR: write only

Figure 1-6. Programming Model—PowerPC 620 Microprocessor Registers

PowerPC processors have two levels of privilege—supervisor mode of operation (typically used by the operating environment) and one that corresponds to the user mode of operation (used by application software). As shown in Figure 1-6, the programming model incorporates 32 GPRs, 32 FPRs, special-purpose registers (SPRs), and several miscellaneous registers. Note that each PowerPC implementation has its own unique set of implementation-dependent registers that are typically used for debugging, configuration, and other implementation-specific operations.

Some registers are accessible only by supervisor-level software. This division allows the operating system to control the application environment (providing virtual memory and protecting operating-system and critical machine resources). Instructions that control the state of the processor, the address translation mechanism, and supervisor registers can be executed only when the processor is in supervisor mode.

The PowerPC registers implemented in the 620 are summarized as follows:

- General-purpose registers (GPRs)—The PowerPC architecture defines 32 user-level, general-purpose registers (GPRs). These registers are 32 bits wide in 32-bit PowerPC implementations and 64 bits wide in 64-bit PowerPC implementations (such as the 620). The 620 also has eight GPR rename buffers, which provide a way to buffer data intended for the GPRs, reducing stalls when the results of one instruction are required by a subsequent instruction. The use of rename buffers is not defined by the PowerPC architecture, and they are transparent to the user with respect to the architecture. The GPRs and their associated rename buffers serve as the data source or destination for instructions executed in the IUs.
- Floating-point registers (FPRs)—The PowerPC architecture also defines 32 floating-point registers (FPRs). These 64-bit registers typically are used to provide source and target operands for user-level, floating-point instructions. The 620 has eight FPR rename buffers that provide a way to buffer data intended for the FPRs, reducing stalls when the results of one instruction are required by a subsequent instruction. The rename buffers are not defined by the PowerPC architecture and are transparent to the user. The FPRs and their associated rename buffers can contain data objects of either single- or double-precision floating-point formats.
- Condition register (CR)—The CR is a 32-bit user-level register that consists of eight 4-bit fields that reflect the results of certain operations, such as move, integer and floating-point compare, arithmetic, and logical instructions, and provide a mechanism for testing and branching. The 620 also has 16 CR rename buffers, which provide a way to buffer data intended for the CR. The rename buffers are not defined by the PowerPC architecture and are transparent to the user.
- Floating-point status and control register (FPSCR)—The floating-point status and control register (FPSCR) is a user-level register that contains all exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE-754 standard.

- Machine state register (MSR)—The machine state register (MSR) is a supervisor-level register that defines the state of the processor. The contents of this register are saved when an exception is taken and restored when the exception handling completes. The 620 implements the MSR as a 64-bit register that provides a superset of the 32-bit functionality.
- Segment registers (SRs)—For memory management, 32-bit PowerPC implementations use sixteen 32-bit segment registers (SRs). The 620 provides 16 segment registers for use when executing programs compiled for 32-bit PowerPC microprocessors (as part of the "optional 64-bit bridge" defined in the architecture).
- Special-purpose registers (SPRs)—The PowerPC operating environment architecture defines numerous special-purpose registers that serve a variety of functions, such as providing controls, indicating status, configuring the processor, and performing special operations. Some SPRs are accessed implicitly as part of executing certain instructions. All SPRs can be accessed by using the move to/from SPR instructions, **mtspr** and **mfspr**.
  - User-level SPRs—The following SPRs are accessible by user-level software:
    - Link register (LR)—The link register can be used to provide the branch target address and to hold the return address after branch and link instructions. The LR is 64 bits wide.
    - Count register (CTR)—The CTR is decremented and tested automatically as a result of branch and count instructions. The CTR is 64 bits wide.
    - XER—The 32-bit XER contains the integer carry and overflow bits.
    - Time base registers (TBL and TBU)—The TBL and TBU can be read by user-level software, but can be written to only by supervisor-level software.
  - Supervisor-level SPRs—The 620 also contains SPRs that can be accessed only by supervisor-level software. These registers consist of the following:
    - DSISR—32-bit data register that defines the cause of DSI and alignment exceptions.
    - Data address register (DAR)—A 64-bit register that holds the address of an access after an alignment or DSI exception.
    - Decrementer register (DEC)—A 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay. In the 620, the decrementer frequency is equal to the bus clock frequency (as is the time base frequency).
    - SDR1 register—The 64-bit register that specifies the page table format used in logical-to-physical address translation for pages.
    - Machine status save/restore register 0 (SRR0)—A 64-bit register that is used by the 620 for saving the address of the instruction that caused the exception, and the address to return to when a Return from Interrupt (**rfi**) instruction is executed.

- Machine status save/restore register 1 (SRR1)—A 64-bit register used to save machine status on exceptions and to restore machine status when an **rfi** instruction is executed.
- SPRG[0–3] registers—64-bit registers provided for operating system use.
- External access register (EAR)—A 32-bit register that controls access to the external control facility through the External Control In Word Indexed (**eciwx**) and External Control Out Word Indexed (**ecowx**) instructions.
- Processor version register (PVR—A 32-bit, read-only register that identifies the version (model) and revision level of the PowerPC processor.
- Time base registers (TBL and TBU)—Both upper and lower registers together, provide a 64-bit time base register. The registers are implemented as a 64-bit counter, with the least-significant bit being the most frequently incremented. The PowerPC architecture defines that the time base frequency be provided as a subdivision of the processor clock frequency. In the 620, the time base frequency is equal to the bus clock frequency (as is the decrementer frequency). Counting is enabled by the Time Base Enable ($\overline{\text{TBENABLE}}$) signal.
- Address space register (ASR)—A 64-bit register that holds the physical address of the segment table. The segment table defines the set of memory segments that can be addressed.
- Block address translation (BAT) registers—The PowerPC architecture defines 16 BAT registers, divided into four pairs of data BATs (DBATs) and four pairs of instruction BATs (IBATs).

The 620 includes the following registers not defined by the PowerPC architecture:

- Instruction address breakpoint register (IABR)—This register can be used to cause a breakpoint exception to occur if a specified instruction address is encountered.
- Data address breakpoint register (DABR)—This register can be used to cause a breakpoint exception to occur if a specified data address is encountered.
- Hardware implementation-dependent register 0 (HID0)—This register is used to control various functions within the 620, such as enabling checkstop conditions, and locking, enabling, and invalidating the instruction and data caches.
- Bus control and status register (BUSCSR)—This register controls the setting of various bus operational parameters, and provides read-only access to bus control values set at system reset.
- L2 cache control register (L2CR)—The L2 cache control register provides controls for the operation of the L2 cache interface, including the ECC mode desired, size of the L2 cache, and the selection of HSTL or CMOS interface logic.
- L2 cache status register (L2SR)—The L2 cache status register contains all ECC error information for the L2 cache interface.

- Processor identification register (PIR)—The PIR is a supervisor-level register that has a right-justified, 4-bit field that holds a processor identification tag used to identify a particular 620. This tag is used to identify the processor in multiple-master implementations.
- Performance monitor counter registers (PMC1 and PMC2)—The counters are used to record the number of times a certain event has occurred.
- Monitor mode control register 0 and 1 (MMCR0 and MMCR1)—These registers are used for enabling various performance monitoring interrupt conditions and establishing the function of the counters.
- Sampled instruction address and sampled data address registers (SIA and SDA)—These registers hold the addresses for instruction and data used by the performance monitoring interrupt.

Note that while it is not guaranteed that the implementation of HID registers is consistent among PowerPC processors, other processors may be implemented with similar or identical HID registers.

1.3.3 Instruction Set and Addressing Modes

The following subsections describe the PowerPC instruction set and addressing modes.

1.3.3.1 PowerPC Instruction Set and Addressing Modes

All PowerPC instructions are encoded as single-word (32-bit) opcodes. Instruction formats are consistent among all instruction types, permitting efficient decoding to occur in parallel with operand accesses. This fixed instruction length and consistent format greatly simplifies instruction pipelining.

1.3.3.1.1 Instruction Set

The 620 implements the entire PowerPC instruction set (for 64-bit implementations) and most optional PowerPC instructions. The PowerPC instructions can be loosely grouped into the following general categories:

- Integer instructions—These include computational and logical instructions.
  - Integer arithmetic instructions
  - Integer compare instructions
  - Logical instructions
  - Integer rotate and shift instructions
- Floating-point instructions—These include floating-point computational instructions, as well as instructions that affect the FPSCR. Floating-point instructions include the following:
  - Floating-point arithmetic instructions
  - Floating-point multiply/add instructions
  - Floating-point rounding and conversion instructions

- — Floating-point compare instructions
- — Floating-point move instructions
- — Floating-point status and control instructions
- — Optional floating-point instructions (listed with the optional instructions below)

The 620 supports all IEEE 754-1985 floating-point data types (normalized, denormalized, NaN, zero, and infinity) in hardware, eliminating the latency incurred by software exception routines.

The PowerPC architecture also supports a non-IEEE mode, controlled by a bit in the FPSCR. In this mode, denormalized numbers, NaNs, and some IEEE invalid operations are not required to conform to IEEE standards and can execute faster. Note that all single-precision arithmetic instructions are performed using a double-precision format. The floating-point pipeline is a single-pass implementation for double-precision products. A single-precision instruction using only single-precision operands in double-precision format performs the same as its double-precision equivalent.

- Load/store instructions—These include integer and floating-point load and store instructions.
  - — Integer load and store instructions
  - — Integer load and store multiple instructions
  - — Integer load and store string instructions
  - — Floating-point load and store
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow.
  - — Branch and trap instructions
  - — System call and **rfi** instructions
  - — Condition register logical instructions
- Synchronization instructions—The instructions are used for memory synchronizing, especially useful for multiprocessing.
  - — Load and store with reservation instructions—These UISA-defined instructions provide primitives for synchronization operations such as test and set, compare and swap, and compare memory.
  - — The Synchronize instruction (**sync**)—This UISA-defined instruction is useful for synchronizing load and store operations on a memory bus that is shared by multiple devices.

  - The Enforce In-Order Execution of I/O instruction (**eieio**)—The **eieio** instruction, defined by the VEA, can be used instead of the **sync** instruction when only memory references seen by I/O devices need to be ordered. The 620 implements **eieio** as a barrier for all storage accesses to the BIU, but not as a barrier for all instructions like the implementation of the **sync** instruction.
- Processor control instructions—These instructions are used for synchronizing memory accesses and managing caches, TLBs, segment registers and SLBs. These instructions include move to/from special-purpose register instructions (**mtspr** and **mfspr**).
- Memory/cache control instructions—These instructions provide control of caches, TLBs, segment registers, and SLBs.
  - User- and supervisor-level cache instructions
  - Segment lookaside buffer management instructions
  - Segment register manipulation instructions
  - Translation lookaside buffer management instructions
- Optional instructions—The 620 implements the following optional instructions:
  - The **eciwx/ecowx** instruction pair
  - TLB invalidate entry instruction (**tlbie**)
  - TLB synchronize instruction (**tlbsync**)
  - SLB invalidate entry instruction (**slbie**)
  - SLB invalidate all instruction (**slbia**)
  - Optional graphics instructions:
    - Store Floating-Point as Integer Word Indexed (**stfiwx**)
    - Floating Reciprocal Estimate Single (**fres**)
    - Floating Reciprocal Square Root Estimate (**frsqrte**)
    - Floating Square Root Single (**fsqrts**)
    - Floating Square Root Double (**fsqrt**)
    - Floating Select (**fsel**)

Note that this grouping of the instructions does not indicate which execution unit executes a particular instruction or group of instructions.

Integer instructions operate on byte, half-word, word, and double-word operands. Floating-point instructions operate on single-precision (one word) and double-precision (one double word) floating-point operands. The PowerPC architecture uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, word, and double-word operand loads and stores between memory and a set of 32 GPRs. It also provides for word and double-word operand loads and stores between memory and a set of 32 FPRs.

Computational instructions do not modify memory. To use a memory operand in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written back to the target location with specific store instructions.

PowerPC processors follow the program flow when they are in the normal execution state. However, the flow of instructions can be interrupted directly by the execution of an instruction or by an asynchronous event. Either kind of exception may cause one of several components of the system software to be invoked.

1.3.3.1.2 Calculating Effective Addresses

The effective address (EA) is the 64-bit address computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction.

The PowerPC architecture supports two simple memory addressing modes:

- EA = (**r**A|0) + offset (including offset = 0) (register indirect with immediate index)
- EA = (**r**A|0) + **r**B (register indirect with index)

These simple addressing modes allow efficient address generation for memory accesses. Calculation of the effective address for aligned transfers occurs in a single clock cycle.

For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the storage operand is considered to wrap around from the maximum effective address to effective address 0.

Effective address computations for both data and instruction accesses use 64-bit unsigned binary arithmetic. A carry from bit 0 is ignored.

1.3.4 Exception Model

The following subsections describe the PowerPC exception model and the 620 implementation, respectively.

The PowerPC exception mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, information about the state of the processor is saved to various registers and the processor begins execution at an address (exception vector) predetermined for each exception and the processor changes to supervisor mode.

Although multiple exception conditions can map to a single exception vector, a more specific condition may be determined by examining a register associated with the exception—for example, the DSISR and the FPSCR. Additionally, specific exception conditions can be explicitly enabled or disabled by software.

The PowerPC architecture requires that exceptions be handled in program order; therefore, although a particular PowerPC processor may recognize exception conditions out of order, exceptions are handled strictly in order. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the exception is taken. Any exceptions caused by those instructions must be handled first. Likewise, exceptions that are asynchronous and precise are recognized when they occur (unless they are masked) and the reorder buffer is drained. The address of the next instruction to be executed is saved in SRR0 so execution can resume at the proper place when the exception handler returns control to the interrupted process.

Unless a catastrophic condition causes a system reset or machine check exception, only one exception is handled at a time. If, for example, a single instruction encounters multiple exception conditions, those conditions are encountered sequentially. After the exception handler handles an exception, the instruction execution continues until the next exception condition is encountered. This method of recognizing and handling exception conditions sequentially guarantees that exceptions are recoverable.

Exception handlers should save the information stored in SRR0 and SRR1 early to prevent the program state from being lost due to a system reset or machine check exception or to an instruction-caused exception in the exception handler.

The PowerPC architecture supports four types of exceptions:

- Synchronous, precise—These are caused by instructions. All instruction-caused exceptions are handled precisely; that is, the machine state at the time the exception occurs is known and can be completely restored.
- Synchronous, imprecise—The PowerPC architecture defines two imprecise floating-point exception modes, recoverable and nonrecoverable. The 620 treats the imprecise, recoverable and imprecise, nonrecoverable modes as the precise mode.
- Asynchronous—The OEA portion of the PowerPC architecture defines two types of asynchronous exceptions:
    - Asynchronous, maskable—The PowerPC architecture defines the external interrupt and decrementer interrupt which are maskable and asynchronous exceptions. In the 620, and in many PowerPC processors, the external interrupt is generated by the assertion of the Interrupt ($\overline{\text{INT}}$) signal, which is not defined by the architecture. In addition, the 620 implements one additional interrupt, the system management interrupt, which performs similarly to the external interrupt, and is generated by the assertion of the System Management Interrupt ($\overline{\text{SMI}}$) signal.

      When these exceptions occur, their handling is postponed until all instructions in progress, and any exceptions associated with those instructions, complete execution.

— Asynchronous, nonmaskable—There are two nonmaskable asynchronous exceptions that are imprecise—system reset and machine check exceptions. Note that the OEA portion of the PowerPC architecture, which defines how these exceptions work, does not define the causes or the signals used to cause these exceptions. These exceptions may not be recoverable, or may provide a limited degree of recoverability for diagnostic purposes.

The PowerPC architecture defines two bits in the MSR—FE0 and FE1—that determine how floating-point exceptions are handled. There are four combinations of bit settings, of which the 620 implements two, which are as follows:

- Ignore exceptions mode—In this mode, the instruction dispatch logic feeds the FPU as fast as possible and the FPU uses an internal pipeline to allow overlapped execution of instructions. In this mode, floating-point exception conditions return a predefined value instead of causing an exception.
- Precise interrupt mode—This mode includes both the precise mode and imprecise recoverable and nonrecoverable modes defined in the PowerPC architecture. In this mode, a floating-point instruction that causes a floating-point exception brings the machine to a precise state. In doing so, the 620 takes floating-point exceptions as defined by the PowerPC architecture.

The 620 exception classes are shown in Table 1-1.

Table 1-1. Exception Classifications

Type	Exception
Asynchronous/nonmaskable	Machine check System reset
Asynchronous/maskable	External interrupt Decrementer System management interrupt (not defined by the PowerPC architecture)
Synchronous/precise	Instruction-caused exceptions
Synchronous/imprecise	Floating-point exceptions

The 620's exceptions, and a general description of conditions that cause them, are listed in Table 1-2.

Table 1-2. Overview of Exceptions and Conditions

Exception Type	Vector Offset (hex)	Causing Conditions
Reserved	00000	—
System reset	00100	A system reset is caused by the assertion of either the soft reset ($\overline{\text{SRESET}}$)or hard reset ($\overline{\text{HRESET}}$) signal.
Machine check	00200	A machine check exception is signaled by the assertion of a qualified $\overline{\text{DERR}}$ indication on the 620 bus, or the machine check interrupt ($\overline{\text{MCP}}$) signal. If MSR[ME] is cleared, the processor enters the checkstop state when one of these signals is asserted. Note that MSR[ME] is cleared when an exception is taken. The machine check exception is also caused by parity errors on the address or data bus, in the instruction or data caches, or L2 ECC errors. The assertion of the $\overline{\text{DERR}}$ signal is determined by load and store operations initiated by the processor; however, it is expected that the $\overline{\text{DERR}}$ signal would be used by a memory controller to indicate that a memory parity error or an uncorrectable memory ECC error has occurred. **Note that the machine check exception is imprecise with respect to the instruction that originated the bus operation.**
DSI	00300	The cause of a DSI exception can be determined by the bit settings in the DSISR, listed as follows: 0 Set if a load or store instruction results in a direct-store program exception; otherwise cleared. 1 Set if the translation of an attempted access is not found in the primary table entry group (PTEG), or in the secondary PTEG, or in the range of a BAT register; otherwise cleared. 4 Set if a memory access is not permitted by the page or BAT protection mechanism; otherwise cleared. 5 If SR[T] = 1, set by an **eciwx**, **ecowx**, **lwarx**, or **stwcx**. instruction; otherwise cleared. Set by an **eciwx** or **ecowx** instruction if the access is to an address that is marked as write-through. 6 Set for a store operation and cleared for a load operation. 9 Set if an EA matches the address in the DABR while in one of the three compare modes. 10 Set if the segment table search fails to find a translation for the effective address; otherwise cleared. 11 Set if **eciwx** or **ecowx** is used and EAR[E] is cleared.
ISI	00400	An ISI exception is caused when an instruction fetch cannot be performed for any of the following reasons: • The effective address cannot be translated. That is, there is a page fault for this portion of the translation, so an ISI exception must be taken to retrieve the translation from a storage device such as a hard disk drive. • The fetch access is to a direct-store segment. • The fetch access violates memory protection. If the key bits (Ks and Kp) in the segment register and the PP bits in the PTE or BAT are set to prohibit read access, instructions cannot be fetched from this location.
External interrupt	00500	An external interrupt occurs when the external exception signal, $\overline{\text{INT}}$, is asserted. This signal is expected to remain asserted until the exception handler begins execution. Once the signal is detected, the 620 stops dispatching instructions and waits for all dispatched instructions to complete. Any exceptions associated with dispatched instructions are taken before the interrupt is taken.

Table 1-2. Overview of Exceptions and Conditions (Continued)

Exception Type	Vector Offset (hex)	Causing Conditions
Alignment	00600	An alignment exception is caused when the processor cannot perform a memory access for the following reasons: • An integer load or store double word is not word aligned. • A floating-point load, store, **lmw, stmw, lwarx, stwcx.**, **eciwx**, or **ecowx** instruction is not word-aligned. • A **dcbz** instruction refers to a page that is marked either caching-inhibited or write-through. • A **dcbz** instruction has executed when the 620 data cache is locked or disabled. • An **lmw, stmw, lswi, lswx, stswi**, or **stswx** instruction is issued in little-endian mode. • A floating-point instruction access to a direct-store segment.
Program	00700	A program exception is caused by one of the following exception conditions, which correspond to bit settings in SRR1 and arise during execution of an instruction: • Floating-point exceptions—A floating-point enabled exception condition causes an exception when FPSCR[FEX] is set and depends on the values in MSR[FE0] and MSR[FE1]. FPSCR[FEX] is set by the execution of a floating-point instruction that causes an enabled exception or by the execution of a Move to FPSCR instruction that results in both an exception condition bit and its corresponding enable bit being set in the FPSCR. • Illegal instruction—An illegal instruction program exception is generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields or when execution of an optional instruction not provided in the specific implementation is attempted (these do not include those optional instructions that are treated as no-ops). • Privileged instruction—A privileged instruction type program exception is generated when the execution of a privileged instruction is attempted and the MSR user privilege bit, MSR[PR], is set. This exception is also generated for **mtspr** or **mfspr** with an invalid SPR field if SPR[0] = 1 and MSR[PR] = 1. • Trap—A trap type program exception is generated when any of the conditions specified in a trap instruction is met.
Floating-point unavailable	00800	A floating-point unavailable exception is caused by an attempt to execute a floating-point instruction (including floating-point load, store, and move instructions) when the floating-point available bit is disabled (MSR[FP] = 0).
Decrementer	00900	The decrementer exception occurs when the most significant bit of the decrementer (DEC) register transitions from 0 to 1.
Reserved	00A00–00BFF	—
System call	00C00	A system call exception occurs when a System Call (**sc**) instruction is executed.
Trace	00D00	Either the MSR[SE] = 1 and any instruction (except **rfi**) successfully completed or MSR[BE] = 1 and a branch instruction is completed.
Floating-point assist	00E00	Defined by the PowerPC architecture, but does not occur in the 620.
Reserved	00E10–00EFF	—

Table 1-2. Overview of Exceptions and Conditions (Continued)

Exception Type	Vector Offset (hex)	Causing Conditions
Performance monitoring interrupt	00F00	The performance monitoring interrupt is a 620-specific exception and is used with the 620 performance monitor, described in Section 1.4, "Performance Monitor." The performance monitoring facility can be enabled to signal an exception when the value in one of the performance monitor counter registers (PMC1 or PMC2) goes negative. The conditions that can cause this exception can be enabled or disabled in the monitor mode control register 0 (MMCR0). Although the exception condition may occur when the MSR[EE] bit is cleared, the actual interrupt is masked by the EE bit and cannot be taken until the EE bit is set.
Reserved	01000–012FF	—
Instruction address breakpoint	01300	An instruction address breakpoint exception occurs when the address (bits 0 to 29) in the IABR matches the next instruction to complete in the completion unit, and the IABR enable bit IABR[30] is set.
System management interrupt	01400	A system management interrupt is caused when MSR[EE] = 1 and the $\overline{\text{SMI}}$ input signal is asserted.
Reserved	01500–02FFF	Reserved, implementation-specific exceptions. These are not implemented in the 620.

1.3.5 Instruction Timing

As shown in Figure 1-7, the common pipeline of the 620 has five stages through which all instructions must pass. Some instructions occupy multiple stages simultaneously and some individual execution units have additional stages. For example, the floating-point pipeline consists of three stages through which all floating-point instructions must pass.

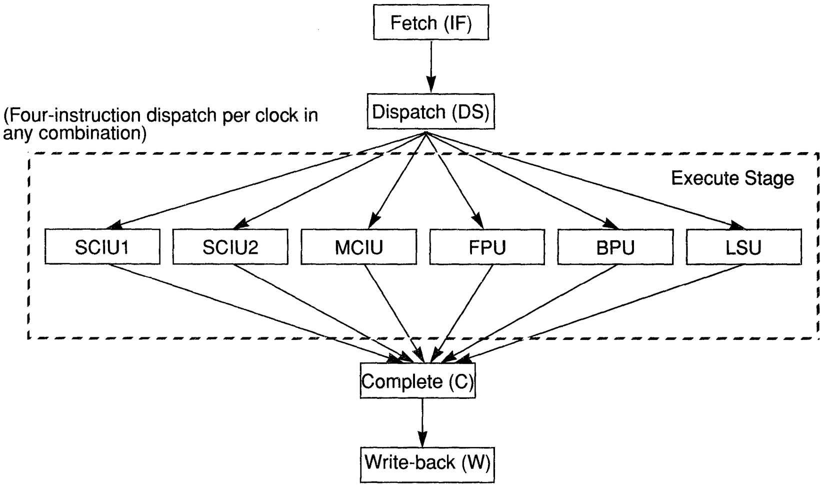


Figure 1-7. Pipeline Diagram

The common pipeline stages are as follows:

- Instruction fetch (IF)—During the IF stage, the fetch unit loads the decode queue (DEQ) with instructions from the instruction cache and determines from what address the next instruction should be fetched.
- Instruction dispatch (DS)—During the dispatch stage, the decoding that is not time-critical is performed on the instructions provided by the previous IF stage. Logic associated with this stage determines when an instruction can be dispatched to the appropriate execution unit. At the end of the DS stage, instructions and their operands are latched into the execution input latches or into the unit's reservation station. Logic in this stage allocates resources such as the rename registers and reorder buffer entries.
- Execute (E)—While the execution stage is viewed as a common stage in the 620 instruction pipeline, the instruction flow is split among the six execution units, some of which consist of multiple pipelines. An instruction may enter the execute stage from either the dispatch stage or the execution unit's dedicated reservation station.

  At the end of the execute stage, the execution unit writes the results into the appropriate rename buffer entry and notifies the completion stage that the instruction has finished execution.

  The execution unit reports any internal exceptions to the completion stage and continues execution, regardless of the exception. Under some circumstances, results can be written directly to the target registers, bypassing the rename buffers.
- Complete (C)—The completion stage ensures that the correct machine state is maintained by monitoring instructions in the completion buffer and the status of instruction in the execute stage.

  When instructions complete, they are removed from the reorder buffer. Results may be written back from the rename buffers to the register as early as the complete stage. If the completion logic detects an instruction containing exception status or if a branch has been mispredicted, all subsequent instructions are cancelled, any results in rename buffers are discarded, and instructions are fetched from the correct instruction stream.
- Write-back (W)—The write-back stage is used to write back any information from the rename buffers that was not written back during the complete stage. The CR, CTR, and LR are updated during the write-back stage.

All instructions are fully pipelined except for divide operations and some integer multiply operations. The integer multiplier is a three-stage pipeline. SPR and divide operations can execute in the MCIU in parallel with multiply operations.

The floating-point pipeline has three stages. All floating-point instructions are fully pipelined except for divide and square root operations.

1.4 Performance Monitor

The 620 incorporates a performance monitor facility that system designers can use to help bring up, debug, and optimize software performance, especially in multiprocessing systems. The performance monitor is a software-accessible mechanism that provides detailed information concerning the dispatch, execution, completion, and memory access of PowerPC instructions.

The monitor mode control registers (MMCR0 and MMCR1) can be used to specify the conditions for which a performance monitoring interrupt is taken. For example, one such condition is associated with one of the performance monitor counter registers (PMC1–PMC8) incrementing until the most significant bit indicates a negative value. Additionally, the sampled instruction address and sampled data address registers (SIA and SDA) are used to hold addresses for instruction and data related to the performance monitoring interrupt.

Chapter 2 Programming Model

This chapter describes the PowerPC programming model with respect to the PowerPC 620. It consists of three major sections, which describe the following:

- Registers implemented in the 620
- Operand conventions
- The 620 instruction set

2.1 The PowerPC 620 Processor Register Set

This section describes the registers in the 620 and includes an overview of the registers defined by the PowerPC architecture and a more detailed description of 620-specific registers and differences in how the registers defined by the PowerPC architecture are implemented in the 620. Full descriptions of the basic register set defined by the PowerPC architecture are provided in Chapter 2, "PowerPC Register Set," in *The Programming Environments Manual.*

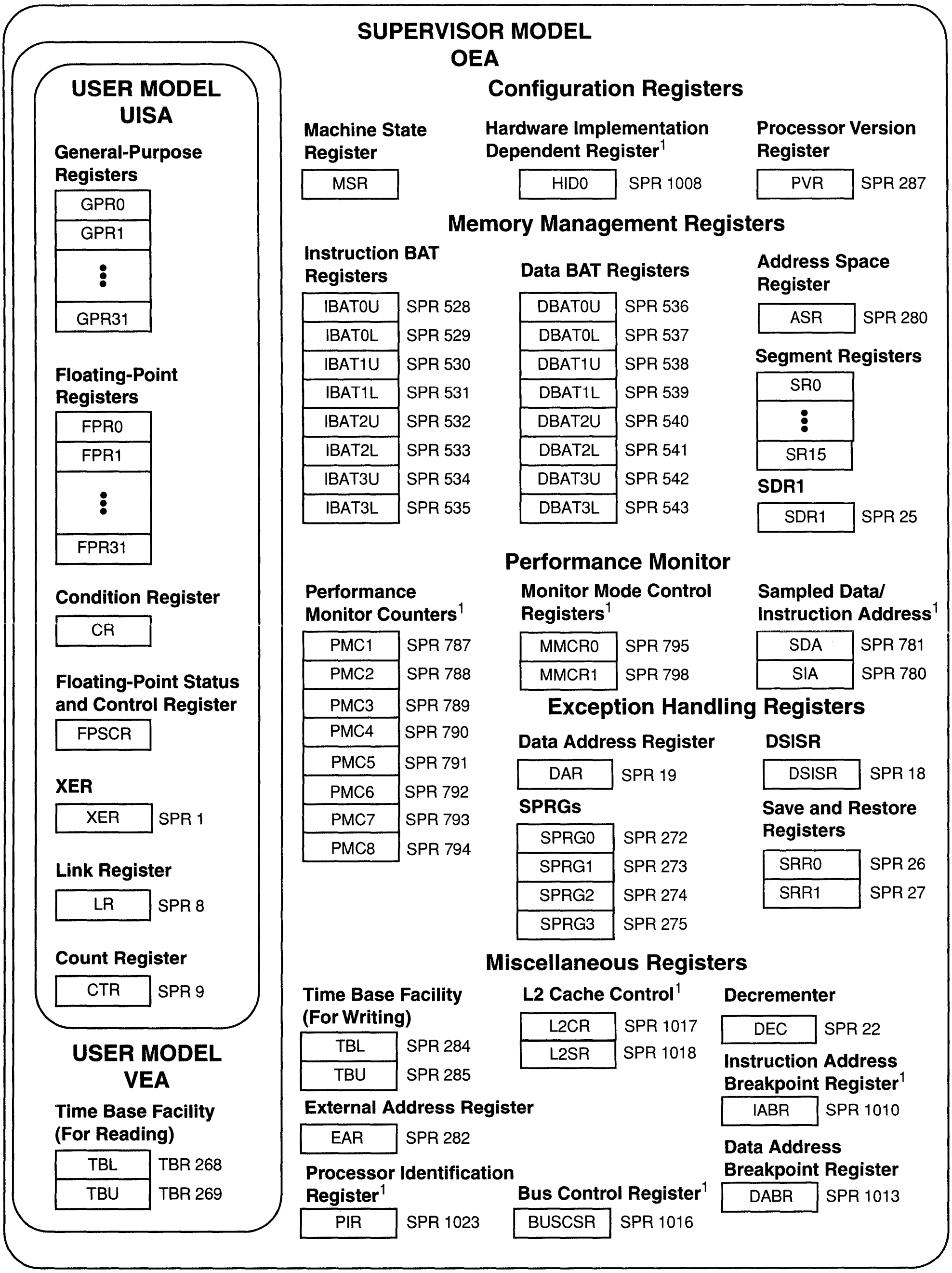
Note that registers are defined at all three levels of the PowerPC architecture—user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA). The PowerPC architecture defines register-to-register operations for all computational instructions. Source data for these instructions are accessed from the on-chip registers or are provided as immediate values embedded in the opcode. The three-register instruction format allows specification of a target register distinct from the two source registers, thus preserving the original data for use by other instructions and reducing the number of instructions required for certain operations. Data is transferred between memory and registers with explicit load and store instructions only.

2.1.1 Register Set

The PowerPC UISA registers, shown in Figure 2-1, are user-level. The general-purpose registers (GPRs) and floating-point registers (FPRs) are accessed through instruction operands. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions) or implicit as part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

The number to the right of the special-purpose registers (SPRs) indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the XER is SPR 1). These registers can be accessed using the **mtspr** and **mfspr** instructions.

Implementation Note—The 620 fully decodes the SPR field of the instruction. If the SPR specified is undefined, the illegal instruction program exception occurs.



[1] 620-specific—not defined by the PowerPC architecture

Figure 2-1. Programming Model—PowerPC 620 Microprocessor Registers

The PowerPC's user-level registers are described as follows:

- **User-level registers** (UISA)—The user-level registers can be accessed by all software with either user or supervisor privileges. The user-level register set includes the following:
  - — General-purpose registers (GPRs)—The PowerPC general-purpose register file consists of thirty-two GPRs designated as GPR0–GPR31. The GPRs serve as data source or destination registers for all integer instructions and provide data for generating addresses. See "General Purpose Registers (GPRs)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.
  - — Floating-point registers (FPRs)—The floating-point register file consists of thirty-two FPRs designated as FPR0–FPR31, which serves as the data source or destination for all floating-point instructions. These registers can contain data objects of either single- or double-precision floating-point format. For more information, see "Floating-Point Registers (FPRs)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual.*
  - — Condition register (CR)—The CR is a 32-bit register, divided into eight 4-bit fields, CR0–CR7, that reflects the results of certain arithmetic operations and provides a mechanism for testing and branching. For more information, see "Condition Register (CR)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual.*

    Implementation Note: The PowerPC architecture indicates that in some implementations the Move to Condition Register Fields (**mtcrf**) instruction may perform more slowly when only a portion of the fields are updated as opposed to all of the fields. The condition register access latency for the 620 is the same in both cases. In the 620, an **mtcrf** instruction that sets only a single field performs significantly faster than one that sets either no fields or multiple fields. For more information regarding the most efficient use of the **mtcrf** instruction, see Section 6.3, "Instruction Scheduling Guidelines."
  - — Floating-point status and control register (FPSCR)—The FPSCR contains all floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard. For more information, see "Floating-Point Status and Control Register (FPSCR)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual.*

    Implementation Note: The PowerPC architecture states that in some implementations, the Move to FPSCR Fields (**mtfsf**) instruction may perform more slowly when only a portion of the fields are updated as opposed to all of the fields. In the 620 implementation, there is no degradation of performance.

The remaining user-level registers are SPRs. Note that the PowerPC architecture provides a separate mechanism for accessing SPRs (the **mtspr** and **mfspr** instructions). These instructions are commonly used to explicitly access certain registers, while other SPRs may be more typically accessed as the side effect of executing other instructions.

- XER—The XER indicates overflow and carries for integer operations. It is set implicitly by many instructions. See "XER Register (XER)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.
- Link register (LR)—The LR provides the branch target address for the Branch Conditional to Link Register (**bclr***x*) instruction, and can optionally be used to hold the logical address of the instruction that follows a branch and link instruction, typically used for linking to subroutines. For more information, see "Link Register (LR)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual.*
- Count register (CTR)—The CTR holds a loop count that can be decremented during execution of appropriately coded branch instructions. The CTR can also provide the branch target address for the Branch Conditional to Count Register (**bcctr***x*) instruction. For more information, see "Count Register (CTR)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual.*

- **User-level registers** (VEA)—The PowerPC VEA introduces the time base facility (TB), a 64-bit structure that maintains and operates an interval timer. The TB is read as a 64-bit register, and written to as two 32-bit registers—time base upper (TBU) and time base lower (TBL). Note that the time base register can be accessed by both user- and supervisor-level instructions. In the context of the VEA, user-level applications are permitted read-only access to the TB. The OEA defines supervisor-level access to the TB for writing values to the TB. For more information, see "PowerPC VEA Register Set—Time Base," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual.*
- **Supervisor-level registers** (OEA)—The OEA defines the registers that are used typically by an operating system for such operations as memory management, configuration, and exception handling. The supervisor-level registers defined by the PowerPC architecture for 64-bit implementations are describes as follows:
  - Configuration registers
    - Machine state register (MSR)—The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsr**), Move to Machine State Register Double Word (**mtmsrd**), System Call (**sc**), Return from Exception (**rfi**), and Return from Exception Double Word (**rfid**) instructions. It can be read by the Move from Machine State Register (**mfmsr**) instruction. See "Machine State Register (MSR)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.

Implementation Note—Note that the 620 defines MSR[61] as the performance monitor marked mode bit (PMM). This additional bit is described in Table 2-1.

Table 2-1. MSR[PMM] Bit

Bit	Name	Description
61	PMM	Performance monitor marked mode. Used to mark specific processes. In conjunction with the MMCR0[3–4], FCM0, and FCM1, provides control for the processes in which the performance monitor is enabled or disabled. 0 Process is not a marked process. 1 Process is a marked process. This bit is specific to the 620, and is defined as reserved by the PowerPC architecture. For more information about the performance monitor, see Chapter 10, "Performance Monitor."

  - Processor version register (PVR)—This register is a read-only register that identifies the version (model) and revision level of the PowerPC processor. For more information, see "Processor Version Register (PVR)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual.*

    Implementation Note: The processor version number is 0x0014 for the 620. The processor revision level starts at 0x0000 and is different for each revision of the chip. The revision level is updated for each silicon revision.

- Memory management registers
  - Block-address translation (BAT) registers—The PowerPC OEA includes eight block-address translation registers (BATs), consisting of four pairs of instruction BATs (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and four pairs of data BATs (DBAT0U–DBAT3U and DBAT0L–DBAT3L). See Figure 2-1 for a list of the SPR numbers for the BAT registers. For more information, see "BAT Registers," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual.* Because BAT upper and lower words are loaded separately, software must ensure that BAT translations are correct during the time that both BAT entries are being loaded.
  - SDR1—The SDR1 register specifies the page table base address used in virtual-to-physical address translation. For more information, see "SDR1," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.
  - Address space register (ASR)—The ASR is a 64-bit register that holds bits 0–51 of the segment table's physical address. For more information, see "Address Space Register (ASR)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.
  - Segment registers (SR)—The PowerPC OEA defines sixteen 32-bit segment registers (SR0–SR15). Note that the SRs are implemented on 32-bit implementations only. The fields in the segment register are interpreted differently depending on the value of bit 0. See "Segment Registers," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.

- Exception handling registers
  - Data address register (DAR)—After a DSI or an alignment exception, DAR is set to the effective address generated by the faulting instruction. See "Data Address Register (DAR)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.
  - SPRG0–SPRG3—The SPRG0–SPRG3 registers are provided for operating system use. See "SPRG0–SPRG3," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.
  - DSISR—The DSISR register defines the cause of DSI and alignment exceptions. See "DSISR," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.
  - Machine status save/restore register 0 (SRR0)—The SRR0 register is used to save machine status on exceptions and to restore machine status when an **rfi** or **rfid** instruction is executed. See "Machine Status Save/Restore Register 0 (SRR0)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.
  - Machine status save/restore register 1 (SRR1)—The SRR1 register is used to save machine status on exceptions and to restore machine status when an **rfi** or **rfid** instruction is executed. See "Machine Status Save/Restore Register 1 (SRR1)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.
- Miscellaneous registers
  - Time Base (TB)—The TB is a 64-bit structure that maintains the time of day and operates interval timers. The TB is read as a 64-bit register, and written to as two 32-bit registers—time base upper (TBU) and time base lower (TBL). Note that the time base registers can be accessed by both user- and supervisor-level instructions. See "Time Base Facility (TB)—OEA," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.
  - Decrementer register (DEC)—This register is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay; the frequency is a subdivision of the processor clock. See "Decrementer Register (DEC)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.

    Implementation Note: In the 620, the time base and decrementer registers are decremented once per bus clock cycle.
  - Data address breakpoint register (DABR)—This optional register can be used to cause a breakpoint exception to occur if a specified data address is encountered. See "Data Address Breakpoint Register (DABR)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.

  - External access register (EAR)—This optional register is used in conjunction with the **eciwx** and **ecowx** instructions. Note that the EAR register and the **eciwx** and **ecowx** instructions are optional in the PowerPC architecture and may not be supported in all PowerPC processors that implement the OEA. See "External Access Register (EAR)," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for more information.

- **Hardware implementation registers**—The PowerPC architecture allows implementations to include SPRs not defined by the PowerPC architecture. Those incorporated in the 620 are described as follows. Note that in the 620, these registers are all supervisor-level registers.
  - Instruction address breakpoint register (IABR)—This register can be used to cause a breakpoint exception to occur if a specified instruction address is encountered.
  - Hardware implementation-dependent register 0 (HID0)—This register is used to control various functions within the 620, such as enabling checkstop conditions, and locking, enabling, and invalidating the instruction and data caches.
  - Processor identification register (PIR)—The PIR is a supervisor-level register that has a right-justified, four-bit field that holds a processor identification tag used to identify a particular 620. This tag is used to identify the processor in multiple-master implementations. Note that although the SPR number is defined by the OEA, the register definition is implementation-specific.
  - Bus control and status register (BUSCR)—This register is used to set various parameters for the processor interface, and provides status and error information for bus operations.
  - L2 cache control register (L2CR)—The L2CR contains all the control parameters for the L2 cache interface. The L2CR also contains all the settings required to set up ECC for the L2 interface.
  - L2 cache status register (L2SR)—The L2SR provides all ECC error information for the L2 cache interface.
  - Performance monitor counter registers (PMC1–PMC8)—These counters are used to record the number of times a certain event has occurred.
  - Monitor mode control registers 0 and 1 (MMCR0 and MMCR1)—These registers are used for enabling various performance monitoring interrupt conditions and establishes the function of the counters.
  - Sampled instruction address and sampled data address registers (SIA and SDA)—These registers hold the addresses for instruction and data used by the performance monitoring interrupt.

Note that while it is not guaranteed that the implementation of HID registers is consistent among PowerPC processors, other processors may be implemented with similar or identical HID registers.

2.1.2 Implementation-Specific Registers

This section describes registers that are defined for the 620 but are not included in the PowerPC architecture. This section also includes a description of the PIR, which is assigned an SPR number by the architecture but is not defined by it. Note that the 620-specific register descriptions do not define all the bits in each register, and that all undefined bits should be considered reserved and should be cleared to 0. Also note that all of the 620-specific registers are supervisor-level registers.

2.1.2.1 Instruction Address Breakpoint Register (IABR)

The 620 also implements an instruction address breakpoint register (IABR). When enabled, instruction fetch addresses will be compared with an effective address that is stored in the IABR. The granularity of these compares will be a word. If the word specified by the IABR is fetched, the instruction breakpoint handler will be invoked. The instruction which triggers the breakpoint will not be executed before the handler is invoked.

The IABR is shown in Figure 2-2.

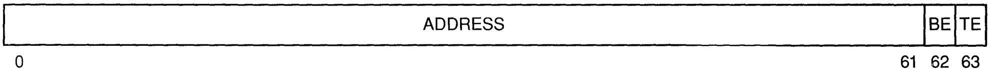


Figure 2-2. Instruction Address Breakpoint Register

The instruction address breakpoint register is used in conjunction with the instruction address breakpoint exception, which occurs when an attempt is made to execute an instruction at an address specified in the IABR. The bits in the IABR are defined as shown in Table 2-2.

Table 2-2. Instruction Address Breakpoint Register Bit Settings

Bit	Description
0–61	Word address to be compared
62	Breakpoint enabled. Setting this bit indicates that breakpoint checking is to be done.
63	Translation enabled. This bit is compared with the MSR[IR] bit. An IABR match is signaled only if these bits also match.

The instruction that triggers the instruction address breakpoint exception is executed before the exception handler is invoked. For more information about the IABR exception, see Section 4.6.14, "Instruction Address Breakpoint Exception (0x01300)."

The IABR can be accessed with the **mtspr** and **mfspr** instructions using the SPR 1010.

2.1.2.2 Processor Identification Register (PIR)

The processor identification register (PIR), shown in Figure 2-3, is a 32-bit register that holds a processor identification tag in the four least significant bits (PIR[28–31]). This tag is useful for processor differentiation in multiprocessor system designs. In addition, this tag is used for several direct-store bus operations in the form of a 'bus transaction from' tag.

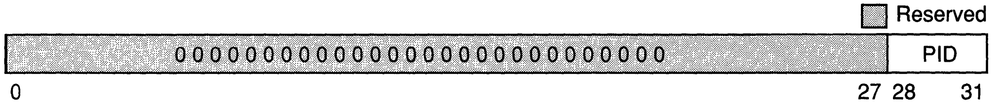


Figure 2-3. Processor Identification Register

The PIR can be accessed with the **mtspr** and **mfspr** instructions using SPR 1023. Note that although this number is defined by the OEA, the register structure is defined by each implementation that implements this optional register.

2.1.2.3 Hardware Implementation-Dependent Register 0 (HID0)

The HID0 register (SPR 1008) controls the state of several functions within the 620. 2.1.2.3 provides bit setting information for the HID0 register.

Table 2-3. HID0 Bit Settings

Bit	Description
0	Enable machine check input 0 The assertion of the $\overline{\text{MCP}}$ does not cause a machine check exception. 1 Enables the entry into a machine check exception based on assertion of the $\overline{\text{MCP}}$ input, detection of a cache parity error, detection of an address parity error, or detection of a data parity error. Note that the machine check exception is further affected by the MSR[ME] bit, which specifies whether the processor checkstops or continues processing.
1	Enable cache parity checking 0 The detection of a cache parity error does not cause a machine check exception. 1 Enables the entry into a machine check exception based on the detection of a cache parity error. Note that the machine check exception is further affected by the MSR[ME] bit, which specifies whether the processor checkstops or continues processing.
2	Enable machine check on address bus parity error 0 The detection of an address bus parity error does not cause a machine check exception. 1 Enables the entry into a machine check exception based on the detection of an address parity error. Note that the machine check exception is further affected by the MSR[ME] bit, which specifies whether the processor checkstops or continues processing.

Table 2-3. HID0 Bit Settings (Continued)

Bit	Description
3	Enable machine check on data bus parity error 0 The detection of a data bus parity error does not cause a machine check exception. 1 Enables the entry into a machine check exception based on the detection of a data bus parity error. Note that the machine check exception is further affected by the MSR[ME] bit, which specifies whether the processor checkstops or continues processing.
14	Processor internal watchdog timer disable 0 Processor internal watchdog timer enabled. The 620 is forced into the checkstop state if it does not complete any valid instructions during the period of time required for the decrementer to pass through 0 twice. 1 Processor internal watchdog timer is disabled
15	Not hard reset 0 A hard reset occurred if software had previously set this bit 1 A hard reset has not occurred.
16	Instruction cache enable 0 The instruction cache is neither accessed nor updated. All pages are accessed as if they were marked cache-inhibited (WIM = X1X). All potential cache accesses from the bus (snoop and cache operations) are ignored. 1 The instruction cache is enabled
17	Data cache enable 0 The data cache is neither accessed nor updated. All pages are accessed as if they were marked cache-inhibited (WIM = X1X). All potential cache accesses from the bus (snoop, cache ops) are ignored. 1 The data cache is enabled.
20	Instruction cache invalidate all 0 The instruction cache is not invalidated. 1 When set, an invalidate operation is issued that marks the state of each clock in the instruction cache as invalid without writing back any modified lines to memory. Access to the cache is blocked during this time. Accesses to the cache from the bus are signaled as a miss while the Invalidate-all operation is in progress. The bit is cleared when the invalidation operation begins (usually the cycle immediately following the write operation to the register). Note that the instruction cache must be enabled for the invalidation to occur.
21	Data cache invalidate all 0 The data cache is not invalidated. 1 When set, an invalidate operation is issued that marks the state of each clock in the data cache as invalid without writing back any modified lines to memory. Access to the cache is blocked during this time. Accesses to the cache from the bus are signaled as a miss while the Invalidate-all operation is in progress. The bit is cleared when the invalidation operation begins (usually the cycle immediately following the Write operation to the register). Note that the data cache must be enabled for the invalidation to occur.
24	Serial instruction execution disable 0 The 620 executes one instruction at a time. The 620 does not post a trace exception after each instruction completes, as it would if MSR[SE] or MSR[BE] were set. 1 Instruction execution is not serialized.
25–26	Branch prediction mode 00 Static branch prediction without update of branch history table (BHT) 01 Dynamic branch prediction 10 Static branch prediction with BHT updates 11 Static branch prediction when branch instruction y-bit = 1 (no BHT update); dynamic branch prediction when y-bit = 0.

Table 2-3. HID0 Bit Settings (Continued)

Bit	Description
27–28	Instruction fetch modes (address translation enabled) 00 No speculative fetch from memory 01 Reserved 10 Reserved 11 Allow unrestricted speculative fetching from memory
30	Branch target address cache disable 0 The branch target address cache is enabled 1 The branch target address cache is disabled

2.1.2.4 Bus Status and Control Register (BUSCSR)

The bus status and control register (BUSCSR) is a 64-bit register (accessed as SPR 1016) that provides the means for setting operational parameters for the 620's system interface and provides status information related to bus operations. Note that some register bits are marked read/write, some are read-only, and others are cleared by a write operation (W=0). Read-only bits are not affected by write operations, and reserved bits are undefined for a read operation, and no operation occurs for a write. The fields of the register are defined in Table 2-4.

Table 2-4. BUSCSR Bit Settings

Bit	Description	Read/ Write
40–41	BUSRATIO configuration (BUSRATIO[0–1]) These bits reflect the setting of the BUSRATIO signals present at the system interface.	Read only
42	Bus clock logic (BUSCLKGTL) This bit reflects the configuration of the BUSCLKGTL input signal. The configuration of the BUSCLKGTL signal determines whether the BUSCLK input signals are to be driven by GTL or CMOS logic. If the BUSCLKGTL bit is set to 1, the BUSCLK input signals are configured for GTL logic; if the BUSCLKGTL bit is cleared to 0, the BUSCLK input signals are configured for CMOS logic.	Read only
44–45	AStat and AResp tenure (BUSRESPTEN[0–1]) These bits reflects the configuration of the BUSRESPTEN input signals, which determine the latency from the assertion of ASTATOUT and ARESPOUT to the sampling of ASTATIN and ARESPIN signals.	Read only
47	64-bit data bus mode (BUSDX) This bit reflects the configuration of the BUSDX input signal. If this bit is cleared to 0, the data bus is configured for 128-bit operation. If this bit is set to 1, the bus is configured for 64-bit operation, and data is transferred via the DH[0–63] signals.	Read only
48–50	Address to AResp latency (BUSTLAR[0–2]) These bits determine the address sampled to response driven latency. 000 8 bus clock cycles 010-111 2–7 bus clock cycles 001 Reserved	R/W
51	Bus data error enable (BUSDERREN) This bit, when set to 1, enables a machine check exception when the $\overline{\text{DERR}}$ signal is asserted.	R/W

Table 2-4. BUSCSR Bit Settings (Continued)

Bit	Description	Read/ Write
52	Bus positive acknowledge error enable (BUSPOSACKEN) Setting this bit to 1 enables the positive acknowledge error condition to cause a machine check exception.	R/W
54	Bus intervention enable (BUSINTVEN) Setting this bit to 1 enables intervention for the bus operations initiated by the processor.	R/W
55	Bus snoop enable (BUSSNPEN) Setting this bit to 1 enables the snooping of bus operations, and the assertion of the M-bit (memory-coherency enforced) for bus operations initiated by the 620. If this bit is cleared to 0 the 620 will not snoop bus operations, and the M bit will not be asserted for bus operations initiated by the 620.	R/W
58	Bus response error (BUSRESPERR) This bit indicates that a reserved response code was detected for an address operation, resulting in a machine check exception and the early termination of the bus operation. This bit is cleared by a write operation to this bit position.	R/W=0
59	Bus positive acknowledge error (BUSPOSACKERR) This bit indicates that an expected positive acknowledge was not received for an address bus operation. This bit is set regardless of the configuration of the BUSCSR[52] bit, and is cleared by a write operation to this bit position.	R/W=0
60	Bus data error (BUSDERR) This bit indicates that the $\overline{\text{DERR}}$ signal was asserted for read data other than PIO Load Last. In the case of PIO Load Last operations, this bit is set only if $\overline{\text{DERR}}$ is asserted for a read data operation and a PIO Reply is received with the error bit set. This bit is set regardless of the configuration of the BUSCSR[51], and is cleared by a write operation to this bit position.	R/W=0
61–63	Bus parity error (BUSPARERR[0–2]) These bits indicate that a bus parity error has occurred, and are set regardless of the configuration of the HID0[EBA] and HID0[EBD] bits. These bits are cleared to 0 by a write operation. 000 Address bus parity error 001 Data bus tag parity error 010 Data bus data parity error 011–111 Reserved	

2.1.2.5 L2 Cache Control Register (L2CR)

The L2 cache control register (L2CR) is a 64-bit register (accessed as SPR 1017) that provides the means for setting operational parameters for the 620's L2 interface, and provides status information related to L2 operations. The fields of the register are defined in Table 2-5.

Table 2-5. L2CR Bit Settings

Bit	Description	Read/ Write
36	L2CLOCKIN and $\overline{\text{L2CLOCKIN}}$ PECL enable (L2CLKPECL) When this bit is set to 1 the L2CLKIN and $\overline{\text{L2CLKIN}}$ signals are placed in PECL mode regardless of the configuration of L2CR[57]. If this bit is cleared to 0, the logic levels for the L2CLKIN and $\overline{\text{L2CLKIN}}$ signals are determined by the configuration of L2CR[57].	R/W
37	Remove dead cycles between read and write operations enable (L2NORWDEAD) When this bit is set to 1 dead cycles are not inserted between read and write cycles on the L2 cache interface.	R/W
40–43	Cache capacity and organization (L2SIZE) These bits determine the L2 cache capacity. For additional information on cache size configuration, refer to Section 9.3.1.3.1, "The L2TAGADD Signal." 0000 1 MB 0001 2 MB 0010 4 MB 0011 8 MB 0100 16 MB 0101 32 MB 0110 64 MB 0111 128 MB 1000–1111 Reserved	R/W
44	L2 drive power (L2DPWR) This bit is used to define the drive power for the L2 interface point-to-point signals. 0 50 ohm drive power 1 Reserved	R/W
46	ECC error enable (L2ECCERREN) When this bit is set to 1 a multi-bit ECC error will cause a machine check exception.	R/W
47–48	Multi-level cache configuration (L2CLC[0–1]) These bits, in conjunction with the HID0[16–17] bits, select one of seven multi-level cache configurations. For more information about the configuration of these bits, refer to Section 9.3.1.8, "L2CLC[0–1] Bits".	R/W
49	L2 PLL enable (L2PLLEN) When this bit is set to 1 the L2 PLL is enabled to lock.	R/W
50	L2 cache initialize enable (L2INIT) When this bit is cleared to 0 it indicates that the L2 SRAMs are being initialized. When in this configuration, L2 read operations are returned a MESI cache state of invalid, and ECC is forced to pass. When this bit is set to 1 the L2 SRAM initialization mode is disabled.	R/W
51	L2 late/normal write select (L2LATEWRITE) When this bit is cleared to 0 write data is driven on the bus data signals the same cycle as the control and address signals are driven to the SRAMs. If this bit is set to 1 the write data is driven on the bus data signals a cycle after the control and address signals are driven to the SRAMs, thereby saving a dead cycle on the bus data signals when performing a read operation after a write operation.	R/W

Table 2-5. L2CR Bit Settings (Continued)

Bit	Description	Read/ Write
52–53	ECC mode select (L2MODE[0–1]) These bits define the L2 cache ECC configuration. 00 Never correct (ECC disabled) 01 Always correct 10 Automatic switch correct 11 Reserved	R/W
54	L2 SRAM register depth select (L2SINGSYNC) This bit is used in conjunction with L2CR[38] to determine the number of pipeline registers present in the SRAMs used to implement the L2 cache. For more information about the configuration of this bit, refer to Section 9.3.1.10, "L2SINGSYNC Bit".	R/W
56	L2 double-bank enable (L2B2ENABLE) When this bit is set to 1 the internal decode for a dual SRAM bank L2 cache is enabled. For more information about the configuration of this bit, refer to Section 9.3.1.7, "L2B2ENABLE Bit".	R/W
60–61	L2 SRAM clock frequency (L2RATIOSR[0–1]) These bits set the ratio of the L2 clock to the processor clock. If the L2 is disabled, these bits must be set to 0b01. 00 Reserved 01 1:1 ratio 10 2:1 ratio 11 3:1 ratio For more information about the configuration of these bits, refer to Section 9.3.1.4, "L2RATIOSR Bit".	R/W

2.1.2.6 L2 Cache Status Register (L2SR)

The L2 status register (L2SR) provides all information related to ECC errors occurring on the L2 cache interface. The L2SR is accessed as SPR 1018. Note that the status provided by the L2SR is valid only for the first ECC error, and subsequent ECC errors before the first ECC error is cleared are lost. An ECC error that occurs during a read operation to the L2SR is lost. Any write operation to the L2SR register results in clearing the register bits to 0 (W=0).

Table 2-6. L2SR Bit Settings

Bit	Description	Read/ Write
16	L2 ECC error detected (L2ECC) This bit is set to 1 when the first single-or double-bit ECC error is detected.	R/W=0
17–25	L2 data syndrome (L2DATASYN[0–8]) These bits reflect the ECC syndrome for the first L2 data ECC error.	R/W=0
26–31	L2 tag syndrome (L2TAGSYN[0–5]) These bits reflect the ECC syndrome for the first L2 tag ECC error.	R/W=0
35–59	L2 ECC address ((L2ECCADDR[0–24]) these bits contain the address of the L2 quadword address of the first detected ECC error. These bits map to bits 11 to 35 of the 40 bit address driven on the address bus. The bits can be identified as either a tag or data address by examining the tag or data syndromes for a non-zero value.	R/W=0

2.1.2.7 Performance Monitor Registers

The remaining twelve registers defined for use with the 620 are used by the performance monitor. For more information about the performance monitor, see Chapter 10, “Performance Monitor.”

2.1.2.7.1 Monitor Mode Control Register 0 (MMCR0)

The monitor mode control register 0 (MMCR0) is a 32-bit SPR (SPR 795) whose bits are partitioned into bit fields that determine the events to be counted and recorded. The selection of allowable combinations of events causes the counters to operate concurrently.

The MMCR0 can be written to or read only in supervisor mode. The MMCR0 includes controls such as counter enable control, counter overflow interrupt control, counter event selection and counter freeze control.

This register must be cleared at power up. Reading this register does not change its contents. The fields of the register are defined in Table 2-7.

Table 2-7. MMCR0 Bit Settings

Bit	Name	Description
0	FC	Freeze counters 0 The values of the PMC*n* counters can be changed by hardware. 1 The values of the PMC*n* counters cannot be changed by hardware.
1	FCS	Freeze counting while in supervisor mode 0 The PMC*n* counters can be changed by hardware. 1 If the processor is in supervisor mode (MSR[PR] is cleared), the counters are not changed by hardware.
2	FCP	Freeze counting while in user mode 0 The PMC*n* counters can be changed by hardware. 1 If the processor is in user mode (MSR[PR] is set), the PMC counters are not changed by hardware.
3	FCM1	Freeze counting while MSR[PM] is set 0 The PMC*n* counters can be changed by hardware. 1 If MSR[PMM] is set, the PMC*n* counters are not changed by hardware.
4	FCM0	Freeze counting while MSR(PMM) is zero. 0 The PMC*n* counters can be changed by hardware. 1 If MSR[PMM] is cleared, the PMC*n* counters are not changed by hardware.
5	PMXE	Performance monitor exception request enable. 0 Interrupt signaling is disabled. 1 Interrupt signaling is enabled. This bit is cleared by hardware when a performance monitor interrupt is signaled. To reenable these interrupt signals, software must set this bit after servicing the performance monitor interrupt. The IPL ROM code clears this bit before passing control to the operating system.
6	FCEX	Freeze counting of PMCs when a performance monitor interrupt is signaled (that is, ((PMC*n*INTCONTROL = 1) & (PMC*n*[0] = 1) & (ENINT = 1)) or the occurrence of an enabled time base transition with ((TBXE =1) & (ENINT = 1)). Setting of this bit can be overridden by configuration of MMCR0[18]. 0 The signaling of a performance monitoring interrupt has no effect on the counting status of PMCs. 1 The signaling of a performance monitoring interrupt prevents the changing of the PMC counters until condition is reset by software. Because a time base signal could have occurred along with an enabled counter negative condition, software should always reset TBXE to0, if the value in TBXE was a 1.
7–8	TBSEL	64-bit time base, bit selection enable 00 Pick bit 63 to count 01 Pick bit 55 to count 10 Pick bit 51 to count 11 Pick bit 47 to count
9	TBXE	Cause interrupt signalling on bit transition (identified in TBSEL) from off to on 0 Do not allow interrupt signal if chosen bit transitions. 1 Signal interrupt if chosen bit transitions. Software is responsible for setting and clearing TBXE.
10–15	THRESHOLD	Threshold value. This number is multiplied by eight and the result is the number of processor cycles to which the threshold value will be set.

Table 2-7. MMCR0 Bit Settings (Continued)

Bit	Name	Description
16	PMC1XE	Enable exception signaling due to PMC1 counter negative. 0 Disable PMC1 exception signaling due to PMC1 counter negative 1 Enable PMC1 exception signaling due to PMC1 counter negative
17	PMC*n*XE	Enable exception signalling due to PMC*n* (where *n*>1) counter negative. This signal overrides the setting of FCEX. 0 Disable PMC*n* exception signaling due to PMC*n* counter negative 1 Enable PMC*n* exception signaling due to PMC*n* counter negative
18	TRIGGER	May be used to trigger counting of PMC*n* (where *n*>1) after PMC1 has become negative or after a performance monitoring interrupt is signaled. 0 Enable PMC*n* counting 1 Disable PMC*n* counting until PMC1 bit 0 is set or until a performance monitor interrupt is signaled This signal can be used to trigger counting of PMC*n* after PMC1 has become negative. This provides a triggering mechanism for counting after a certain condition occurs or after a preset time has elapsed. It can be used to support getting the count associated with a specific event.
19–25	PMC1SEL	PMC1 input selector, see Table 2-9 for events selectable.
26–31	PMC2SEL	PMC2 input selector, see Table 2-10 for events selectable.

2.1.2.7.2 Monitor Mode Control Register 1 (MMCR1)

The monitor mode control register 1 (MMCR1) is a 32-bit SPR (SPR 798) whose bits are partitioned into bit fields that determine the events to be counted and recorded. The selection of allowable combinations of events causes the counters to operate concurrently.

The MMCR1 can be written to or read only in supervisor mode. The MMCR1 includes controls such as counter enable control, counter overflow interrupt control, counter event selection and counter freeze control.

This register must be cleared at power up. Reading this register does not change its contents. The fields of the register are defined in Table 2-8

Table 2-8. MMCR1 Bit Settings

Bit	Name	Description
0–4	PMC3SEL	PMC3 input selector, see Table 2-11 for events selectable.
5–9	PMC4SEL	PMC4 input selector, see Table 2-12 for events selectable.
10–14	PMC5SEL	PMC5 input selector, see Table 2-13 for events selectable.
15–19	PMC6SEL	PMC6 input selector, see Table 2-14 for events selectable.
20–24	PMC7SEL	PMC7 input selector, see Table 2-15 for events selectable.
25–28	PMC8SEL	PMC8 input selector, see Table 2-16 for events selectable.

Table 2-8. MMCR1 Bit Settings (Continued)

Bit	Name	Description
29	FCUIABR	Freeze Counters until IABR Match. After a monitored IABR match is detected this bit is reset to zero by the hardware. An IABR match is said to be monitored if it occurs when PMC updates are permitted by the configuration of MMCR0[0–4], MSR[PR] and MSR[PMM]. 0 = The PMCs are conditionally updated. 1 = The PMCs are not updated until a "monitored" IABR match occurs.
30	PMC1HIST	PMC1 History Mode 0 = PMC1 is conditionally incremented. 1 = PMC1 is in History mode.
31	PMC*n*HIST	PMC*n, n*>1, History mode 0 = PMC*n, n*>1, are conditionally incremented. 1 = PMC*n, n*>1, are in History mode.

2.1.2.7.3 Performance Monitor Counter Registers (PMC1–PMC8)

PMC1 through PMC8 are 32-bit counters that can be programmed to generate interrupt signals when they are negative. Counters are considered to be negative when the high-order bit (the sign bit) becomes set; that is, they reach the value 2147483648 (0x8000_0000). However, an interrupt is not signaled unless both PCM*n*[XE] and MMCR0[PMXE] are also set.

Note that the interrupts can be masked by clearing MSR[EE]; the interrupt signal condition may occur with MSR[EE] cleared, but the interrupt is not taken until the EE bit is set. Setting MMCR0[FCEX] forces the counters to stop counting when a counter interrupt occurs.

PMC1 through PMC8 are SPRs 787 through 794, respectively, and can be read and written to by using the **mfspr** and **mtspr** instructions. Software is expected to use the **mtspr** instruction to explicitly set the PMC register to non-negative values. If software sets a negative value, an erroneous interrupt may occur. For example, if both PCM*n*[XE] and MMCR0[PMXE] are set and the **mtspr** instruction is used to set a negative value, an interrupt signal condition may be generated prior to the completion of the **mtspr** and the values of the SIA and SDA may not have any relationship to the type of instruction being counted.

The event that is to be monitored can be chosen by setting the appropriate bits in the MMCR0[19–31]. The number of occurrences of these selected events is counted from the time the MMCR0 was set either until a new value is introduced into the MMCR0 register or until a performance monitor interrupt is generated. Table 2-9 lists the selectable events with their appropriate MMCR0 encodings.

Table 2-9. Selectable Events—PMC1

MMCR0[19–25] Encoding	Description
0x00	Processor cycles.
0x01	Number of instructions completed.
0x02	Time base selected bit transition from 0 to 1.
0x03	Number of instructions dispatched.
0x04	Number of load instructions completed.
0x05	L1 instruction cache miss.
0x06	A load miss occurred in L1.
0x07	Threshold exceeded (loads with no L2 intervention)
0x08	Data cache EPAT miss.
0x09	Threshold exceeded (stores with no L2 intervention)
0x0A	A Read-Burst missed the L2 and another bus device has modified data.
0x0B	L1 instruction cache IERAT miss.
0x0C	Brought/wrote a line into the ICACHE and used it.
0x0D	Data cache detected an offset hit.
0x0E	Number of instructions deleted due to global cancel.
0x0F	Chaining the counters in history mode. (PMC1 to PMC8)
0x12	A master-generated store operation is retried.
0x14	MSR external interrupt enable bit, MSR[EE], is off
0x15	Branch unit idle.
0x16	A single instruction serialization class instruction is in execution (Counts the total number of cycles this condition is detected.)
0x17	The FPU status and control register instructions.
0x18	One store buffer is in use.
0x19	A snooped operation cleaned data from the L2.
0x1A	Number of stores in the completion buffer.
0x1B	The link register stack is full.
0x1C	A conditional branch was resolved at dispatch.
0x1D	Number of loads in the completion buffer.
0x1E	Number of entries in the completion buffer.

Table 2-9. Selectable Events—PMC1 (Continued)

MMCR0[19–25] Encoding	Description
0x1F	The finished store queue (FSQ) is full.
0x51	Data cache and instruction cache SLB miss occurred.
0x53	Data cache and instruction cache TLB miss.
0x56	A single instruction serialization class instruction is in execution (Counts the number of times this condition is detected.)

Bits MMCR0[26–31] are used for selecting events associated with PMC2. These settings are shown in Table 2-10.

Table 2-10. Selectable Events—PMC2

MMCR0[26–31] Select Encoding	Description
0x00	Number of instructions completed.
0x01	Processor cycles.
0x02	Time base selected bit transition from zero to one.
0x03	Number of instructions dispatched.
0x05	Data cache store address lookup.
0x06	A sampled Read-Burst generated an L2 miss.
0x08	A conditional branch was predicted.
0x09	Store miss occurred in L1.
0x0A	Threshold exceeded (loads with L2 intervention)
0x0B	A Read-with-Intent-to-Modify (RWITM) generated an L2 access.
0x0C	Threshold exceeded (stores with L2 intervention)
0x0D	A store conditional instruction failed to execute successfully
0x0E	A master-generated non-burst store operation is stalled waiting for a store buffer.
0x0F	Chaining the counters in history mode. (PMC2 to PMC1)
0x10	A sampled Read-Burst missed the L2 and another bus device has modified data.
0x11	The complex integer unit does not have a valid instruction to execute.
0x12	A system call interrupt was taken.
0x14	Two store buffers are in use.
0x15	A master-generated load operation is not retried.
0x16	A **lwarx** instruction has finished execution.
0x18	A sample store instruction was scheduled for execution.

Table 2-10. Selectable Events—PMC2 (Continued)

MMCR0[26–31] Select Encoding	Description
0x19	The instruction buffer is empty this cycle.
0x1C	A snooped operation generated a push or an intervention.
0x1D	A master-generated store operation is loaded into the store buffer.
0x33	Data cache and instruction cache SLB miss occurred.

Bits MMCR1[0–4] are used for selecting events associated with PMC3. These settings are shown in Table 2-11.

Table 2-11. Selectable Events—PMC3

MMCR1[0–4] Select Encoding	Description
0x00	Brought/wrote a line into the ICACHE and used it.
0x01	Processor cycles.
0x02	Number of instructions completed.
0x03	Time base selected bit transition from zero to one.
0x04	Number of instructions dispatched.
0x05	A load miss occurred in L1.
0x06	A sampled Read-with-Intent-to-Modify (RWITM) generated an L2 miss.
0x07	The branch queue is full.
0x08	A sampled Read-with-Intent-to-Modify (RWITM) missed the L2 and another bus device has modified data.
0x09	A store instruction was completed.
0x0A	A sampled store was completed.
0x0B	A load instruction is the next instruction to complete.
0x0C	A Read-with-Intent-to-Modify (RWITM) generated an L2 miss.
0x0D	A sampled Read-Burst generated an L2 access.
0x0E	A master-generated Non-Burst Store operation is stalled waiting for a store buffer.
0x0F	Chaining the counters in History mode. (PMC3 to PMC2)
0x10	A double word unaligned store was scheduled
0x11	A master-generated store operation is loaded into the store buffer.
0x13	Three store buffers are in use.

Table 2-11. Selectable Events—PMC3 (Continued)

MMCR1[0–4] Select Encoding	Description
0x14	A master-generated store conditional (STCX) is cancelled.
0x15	A snooped operation generated a transition in the L2 from Exclusive or Shared to Invalid.
0x16	The FPU divide instructions.
0x18	I/O interrupts detected.

Bits MMCR1[5–9] are used for selecting events associated with PMC4. These settings are shown in Table 2-12.

Table 2-12. Selectable Events—PMC4

MMCR1[5–9] Select Encoding	Description
0x00	L1 instruction cache IERAT miss.
0x01	Processor cycles.
0x02	Number of instructions completed.
0x03	Time base selected bit transition from zero to one.
0x04	Number of instructions dispatched.
0x05	Number of load instructions completed.
0x07	The load/store scheduled a sampled load instruction
0x08	A sampled Read-with-Intent-to-Modify (RWITM) generated an L2 access.
0x09	The load/store received data from the data cache.
0x0A	A Read-with-Intent-to-Modify (RWITM) missed the L2 and another bus device has modified data.
0x0B	Data cache **sync** request was made to the BIU.
0x0C	Global cancel due to a load or store instruction address conflict.
0x0D	The multi-cycle integer unit pipeline is busy with a valid instruction.
0x0E	A master-generated store operation is not retried.
0x0F	Chaining the counters in History mode. (PMC4 to PMC3)
0x10	Data cache detected an aliased hit.
0x11	The simple integer unit 1 does not have a valid instruction to execute.
0x12	A double word unaligned load was scheduled.
0x13	Completion stalled on a load operation.
0x14	A master-generated bus operation received an ARESPIN Retry.
0x16	Branch completed.

Table 2-12. Selectable Events—PMC4 (Continued)

MMCR1[5–9] Select Encoding	Description
0x17	The dispatch buffer is empty this cycle.
0x18	Link register stack error.
0x19	The condition register logical unit produced a result.
0x1B	A snooped operation cleaned data from the L1.

Bits MMCR1[10–14] are used for selecting events associated with PMC5. These settings are shown in Table 2-13.

Table 2-13. Selectable Events—PMC5

MMCR1[10–14] Select Encoding	Description
0x00	Data cache EPAT miss.
0x01	The instruction cache was accessed and a fetch block was fetched.
0x02	No instructions completed.
0x04	A Read-Burst generated an L2 access.
0x05	The FPU finished the execution of an instruction.
0x06	The load/store reservation stations are empty.
0x07	BTAC hit.
0x08	Completed store queue (CSQ) is full.
0x09	A master-generated store operation is stalled waiting for a store buffer.
0x0A	A snooped operation generated a transition in the L2 from Modified to Invalid.
0x0B	The FPU convert and round instructions.
0x0C	Processor cycles.
0x0D	A master-generated Bus operation received an ASTATIN Retry.
0x0F	Chaining the counters in History mode. (PMC5 to PMC4)

Bits MMCR1[15–19] are used for selecting events associated with PMC6. These settings are shown in Table 2-14.

Table 2-14. Selectable Events—PMC6

MMCR1[15–19] Select Encoding	Description
0x01	Store hit occurred in L1.
0x02	The multi-cycle integer unit finished the execution of an instruction.
0x03	A BTAC miss was detected.
0x04	An instruction fetch generated an L2 miss.
0x05	A conditional branch was dispatched.
0x06	The load queue is full.
0x08	A snooped operation generated a push or an intervention.
0x09	The MSR[EE] bit is off and an external interrupt is pending.
0x0A	A master-generated load operation is retried.
0x0B	The FPU move instructions and the select instruction.
0x0C	Processor cycles.
0x0D	A snooped operation accessed the L2.
0x0E	A snooped operation generated a transition in the L2 from Exclusive to Shared.
0x0F	Chaining the counters in History mode. (PMC6 to PMC5)

Bits MMCR1[20–24] are used for selecting events associated with PMC7. These settings are shown in Table 2-15.

Table 2-15. Selectable Events—PMC7

MMCR1[20–24] Select Encoding	Description
0x00	L1 instruction cache miss.
0x01	The simple integer unit 0 finished the execution of an instruction.
0x02	A branch was dispatched (any).
0x03	Global cancel due to a branch guessed wrong.
0x04	A bus operation was snooped.
0x06	No instructions dispatched.
0x07	The simple integer unit 0 does not have a valid instruction to execute.
0x0A	A store instruction was dispatched.
0x0B	Processor cycles.
0x0F	Chaining the counters in History mode. (PMC7 to PMC6)

Bits MMCR1[25–28] are used for selecting events associated with PMC8. These settings are shown in Table 2-16.

Table 2-16. Selectable Events—PMC8

MMCR1[25–28] Select Encoding	Description
0x1	A snooped operation hit the L2.
0x2	A Read-Burst generated an L2 miss.
0x3	A store conditional instruction executed successfully.
0x4	The simple integer unit 1 finished the execution of an instruction.
0x5	A bus operation was ASTATOUT Retried.
0x7	Prefetch bad.
0x8	Completion stalled on a store operation.
0xA	A load instruction was dispatched.
0xB	Misaligned data interrupt
0xC	Processor cycles.
0xF	Chaining the counters in History mode. (PMC8 to PMC7)

2.1.2.7.4 Sampled Instruction Address Register (SIA)

The SIA contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. If the performance monitor interrupt was triggered by a threshold event, the SIA contains the exact instruction that caused the counter to become negative. The instruction whose effective address is put in the SIA is called the sampled instruction. For more information on threshold-related interrupts, see Section •, “Performance Monitor Events.”

If the performance monitor interrupt was caused by something besides a threshold event, the SIA contains the address of the last instruction completed during that cycle. The SIA is a supervisor-level SPR.

The SIA (SPR 780) can be read by using the **mfspr** instruction and written to by using the **mtspr** instruction.

2.1.2.7.5 Sampled Data Address Register (SDA)

The SDA contains the effective address of an operand of the last sampled instruction executed by the load/store unit at or around the time that the processor signals the performance monitor interrupt condition. If the performance monitor interrupt was triggered by a threshold event, the SDA contains the effective address of the operand of the SIA.

If the performance monitor interrupt was caused by something other than a threshold event, the SIA contains the address of the last instruction completed during that cycle. The SDA contains an effective address that is not guaranteed to match the instruction in the SIA. The SDA is a supervisor-level SPR.

The SDA (SPR 781) can be read by using the **mfspr** instruction and written to by using the **mtspr** instruction.

2.2 Operand Conventions

This section describes the operand conventions as they are represented in two levels of the PowerPC architecture—UISA and VEA. Detailed descriptions are provided of conventions used for storing values in registers and memory, accessing PowerPC registers, and representation of data in these registers.

2.2.1 Floating-Point Execution Models

The IEEE 754 standard defines conventions for 64- and 32-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands.

- Double-precision arithmetic instructions may have single-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision and always produce single-precision results.

For arithmetic instructions, conversion from double- to single-precision must be done explicitly by software, while conversion from single- to double-precision is done implicitly by the processor.

All PowerPC implementations provide the equivalent of the following execution models to ensure that identical results are obtained. The definition of the arithmetic instructions for infinities, denormalized numbers, and NaNs follow conventions described in the following sections.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following examples when the corresponding exception enable bit is one:

- Underflow during multiplication using a denormalized operand
- Overflow during division using a denormalized divisor

2.2.2 Data Organization in Memory and Data Transfers

Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Memory operands may be bytes, half words, words, or double words, or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

2.2.3 Alignment and Misaligned Accesses

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the natural address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned.

Operands for single-register memory access instructions have the characteristics shown in Table 2-18. (Although not permitted as memory operands, quad words are shown because quad-word alignment is desirable for certain memory operands).

Table 2-17. Supported Data Sizes and Alignments

Size (Bytes)	Definition	Ordinary Segment (T=0)	Direct Store Segment (T=1)	Notes
1	Byte	Supported	Supported	All byte alignments are supported.
2	Half-Word	Supported	Supported	All alignments that do not cross a double word boundary are supported.
3	3-Byte	Supported	Supported	
4	Word	Supported	Supported	
8	Double-Word	Supported	Supported	Only double word alignments are supported.
16	Quad-Word	Supported	Unsupported	Only quad word aligned quad word is supported.
5–7, 9–15	—	Unsupported	Unsupported	Not Supported

The concept of alignment is also applied more generally to data in memory. For example, a 12-byte data item is said to be word-aligned if its address is a multiple of four.

Some instructions require their memory operands to have certain alignment. In addition, alignment may affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

Instructions are 32 bits (one word) long and must be word-aligned.

2.2.4 Floating-Point Operand

The 620 provides hardware support for all single- and double-precision floating-point operations for most value representations and all rounding modes. This architecture provides for hardware to implement a floating-point system as defined in ANSI/IEEE standard 754-1985, *IEEE Standard for Binary Floating Point Arithmetic.* Detailed information about the floating-point execution model can be found in Chapter 3, "Operand Conventions," in *The Programming Environments Manual.*

The 620 supports non-IEEE mode whenever FPSCR[29] is set. In this mode, denormalized numbers, NaNs, and some IEEE invalid operations are treated in a non-IEEE conforming manner. This is accomplished by delivering results that approximate the values required by the IEEE standard. Table 2-18 summarizes the conditions and mode behavior for operands.

Table 2-18. Floating-Point Operand Data Type Behavior

Operand A Data Type	Operand B Data Type	Operand C Data Type	IEEE Mode (NI = 0)	Non-IEEE Mode (NI = 1)
Single denormalized Double denormalized	Single denormalized Double denormalized	Single denormalized Double denormalized	Normalize all three	Zero all three
Single denormalized Double denormalized	Single denormalized Double denormalized	Normalized or zero	Normalize A and B	Zero A and B
Normalized or zero	Single denormalized Double denormalized	Single denormalized Double denormalized	Normalize B and C	Zero B and C
Single denormalized Double denormalized	Normalized or zero	Single denormalized Double denormalized	Normalize A and C	Zero A and C
Single denormalized Double denormalized	Normalized or zero	Normalized or zero	Normalize A	Zero A
Normalized or zero	Single denormalized Double denormalized	Normalized or zero	Normalize B	Zero B
Normalized or zero	Normalized or zero	Single denormalized Double denormalized	Normalize C	Zero C
Single QNaN Single SNaN Double QNaN Double SNaN	Don't care	Don't care	QNaN[1]	QNaN[1]
Don't care	Single QNaN Single SNaN Double QNaN Double SNaN	Don't care	QNaN[1]	QNaN[1]
Don't care	Don't care	Single QNaN Single SNaN Double QNaN Double SNaN	QNaN[1]	QNaN[1]

Table 2-18. Floating-Point Operand Data Type Behavior (Continued)

Operand A Data Type	Operand B Data Type	Operand C Data Type	IEEE Mode (NI = 0)	Non-IEEE Mode (NI = 1)
Single normalized Single infinity Single zero Double normalized Double infinity Double zero	Single normalized Single infinity Single zero Double normalized Double infinity Double zero	Single normalized Single infinity Single zero Double normalized Double infinity Double zero	Do the operation	Do the operation

[1] Prioritize according to Chapter 3, "Operand Conventions," in *The Programming Environments Manual.*

Table 2-19 summarizes the mode behavior for results.

Table 2-19. Floating-Point Result Data Type Behavior

Precision	Data Type	IEEE Mode (NI = 0)	Non-IEEE Mode (NI = 1)
Single	Denormalized	Return single-precision denormalized number with trailing zeros.	Return zero.
Single	Normalized Infinity Zero	Return the result.	Return the result.
Single	QNaN SNaN	Return QNaN.	Return QNaN.
Single	INT	Place integer into low word of FPR.	If (Invalid Operation) then Place (0x8000) into FPR[32–63] else Place integer into FPR[32–63].
Double	Denormalized	Return double precision denormalized number.	Return zero.
Double	Normalized Infinity Zero	Return the result.	Return the result.
Double	QNaN SNaN	Return QNaN.	Return QNaN.
Double	INT	Not supported by 620	Not supported by 620

2.2.5 Effect of Operand Placement on Performance

The PowerPC VEA states that the placement (location and alignment) of operands in memory may affect the relative performance of memory accesses. The best performance is guaranteed if memory operands are aligned on natural boundaries. To obtain the best performance across the widest range of PowerPC processor implementations, the programmer should assume the performance model described in Chapter 3, "Operand Conventions," in *The Programming Environments Manual.*

2.3 Instruction Set Summary

This section describes instructions and addressing modes defined for the 620. These instructions are divided into the following functional categories:

- Integer instructions—These include arithmetic and logical instructions. For more information, see Section 2.3.4.1, "Integer Instructions."
- Floating-point instructions—These include floating-point arithmetic instructions, as well as instructions that affect the floating-point status and control register (FPSCR). For more information, see Section 2.3.4.2, "Floating-Point Instructions."
- Load and store instructions—These include integer and floating-point load and store instructions. For more information, see Section 2.3.4.3, "Load and Store Instructions."
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow. For more information, see Section 2.3.4.4, "Branch and Flow Control Instructions."
- Processor control instructions—These instructions are used for synchronizing memory accesses and managing caches, TLBs, and segment registers. For more information, see Section 2.3.4.6, "Processor Control Instructions," Section 2.3.5.1, "Processor Control Instructions," and Section 2.3.6.2, "Processor Control Instructions."
- Memory synchronization instructions—These instructions are used for memory synchronizing. See Section 2.3.4.7, "Memory Synchronization Instructions," Section 2.3.5.2, "Memory Synchronization Instructions," for more information.
- Memory control instructions—These instructions provide control of caches, TLBs, and segment registers. For more information, see Section 2.3.5.3, "Memory Control Instructions," and Section 2.3.6.3, "Memory Control Instructions."
- External control instructions—These include instructions for use with special input/output devices. For more information, see Section 2.3.5.4, "Optional External Control Instructions."

Note that this grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions. This information, which is useful in taking full advantage of the 620's superscalar parallel instruction execution, is provided in Chapter 6, "Instruction Timing."

Integer instructions operate on word operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The PowerPC architecture uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, word, double word, multiple word, and string operand loads and stores between memory and a set of 32 general-purpose registers (GPRs). It also provides for word and double word operand loads and stores between memory and a set of 32 floating-point registers (FPRs).

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

The description of each instruction includes the mnemonic and a formatted list of operands. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the frequently-used instructions; see Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual* for a complete list of simplified mnemonics. Note that the architecture specification refers to simplified mnemonics as extended mnemonics. Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in that document.

2.3.1 Classes of Instructions

The 620 instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

Note that while the definitions of these terms are consistent among the PowerPC processors, the assignment of these classifications is not. For example, a PowerPC instruction defined for 64-bit implementations are treated as illegal by 32-bit implementations such as the PowerPC 604™.

The class is determined by examining the primary opcode and the extended opcode, if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction or of a reserved instruction, the instruction is illegal.

Instruction encodings that are now illegal may become assigned to instructions in the architecture or may be reserved by being assigned to processor-specific instructions.

2.3.1.1 Definition of Boundedly Undefined

If instructions are encoded with incorrectly set bits in reserved fields, the results on execution can be said to be boundedly undefined. If a user-level program executes the incorrectly coded instruction, the resulting undefined results are bounded in that a spurious change from user to supervisor state is not allowed, and the level of privilege exercised by the program in relation to memory access and other system resources cannot be exceeded. Boundedly undefined results for a given instruction may vary between implementations, and between execution attempts in the same implementation.

2.3.1.2 Defined Instruction Class

Defined instructions are guaranteed to be supported in all PowerPC implementations, except as stated in the instruction descriptions in Chapter 8, "Instruction Set," in *The*

Programming Environments Manual. The 620 provides hardware support for all instructions defined for 64-bit implementations.

A PowerPC processor invokes the illegal instruction error handler (part of the program exception) when the unimplemented PowerPC instructions are encountered so they may be emulated in software, as required. Note that the architecture specification refers to exceptions as interrupts.

The 620 provides hardware support for all instructions defined for 64-bit implementations. The 620 supports the optional **fsqrt** and **fsqrts** instructions, and does not support the **tlbie** and **tlbia** instructions.

A defined instruction can have invalid forms. The 620 provides limited support for instructions that are represented in an invalid form. Appendix B, "Invalid Instruction Forms," lists all invalid instruction forms and specifies the operation of the 620 upon detecting each.

2.3.1.3 Illegal Instruction Class

Illegal instructions can be grouped into the following categories:

- Instructions not defined in the PowerPC architecture.The following primary opcodes are defined as illegal but may be used in future extensions to the architecture:

  1, 4, 5, 6, 9, 22, 56, 57, 60, 61

  Future versions of the PowerPC architecture may define any of these instructions to perform new functions.

- All unused extended opcodes are illegal. The unused extended opcodes can be determined from information in Section A.2, "Instructions Sorted by Opcode," and Section 2.3.1.4, "Reserved Instruction Class." Notice that extended opcodes for instructions defined only for 64-bit implementations are illegal in 32-bit implementations, and vice versa. The following primary opcodes have unused extended opcodes.

  17, 19, 31, 59, 63 (Primary opcodes 30 and 62 are illegal for all 32-bit implementations, but as 64-bit opcodes they have some unused extended opcodes.)

- An instruction consisting of only zeros is guaranteed to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized memory invokes the system illegal instruction error handler (a program exception). Note that if only the primary opcode consists of all zeros. The instruction is considered a reserved instruction, as described in Section 2.3.1.4, "Reserved Instruction Class."

The 620 invokes the system illegal instruction error handler (a program exception) when it detects any instruction from this class.

See Section 4.6.7, "Program Exception (0x00700)," for additional information about illegal and invalid instruction exceptions. With the exception of the instruction consisting entirely of binary zeros, the illegal instructions are available for further additions to the PowerPC architecture.

2.3.1.4 Reserved Instruction Class

Reserved instructions are allocated to specific implementation-dependent purposes not defined by the PowerPC architecture. An attempt to execute an unimplemented reserved instruction invokes the illegal instruction error handler (a program exception). See "Program Exception (0x00700)," in Chapter 6, "Exceptions," in *The Programming Environments Manual* for additional information about illegal and invalid instruction exceptions.

The PowerPC architecture defines four types of reserved instructions:

- Instructions in the POWER architecture not part of the PowerPC UISA

  POWER architecture incompatibilities and how they are handled by PowerPC processors are listed in Appendix B, "POWER Architecture Cross Reference," in *The Programming Environments Manual.*
- Implementation-specific instructions required to conform to the PowerPC architecture
- Architecturally-allowed extended opcodes
- Implementation-specific instructions

2.3.2 Addressing Modes

This section provides an overview of conventions for addressing memory and for calculating effective addresses as defined by the PowerPC architecture for 64-bit implementations. For more detailed information, see "Conventions," in Chapter 4, "Addressing Modes and Instruction Set Summary," of *The Programming Environments Manual.*

2.3.2.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a memory access or branch instruction or when it fetches the next sequential instruction.

Bytes in memory are numbered consecutively starting with zero. Each number is the address of the corresponding byte.

2.3.2.2 Memory Operands

Memory operands may be bytes, half words, words, or double words, or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction. The PowerPC architecture supports both big-endian and little-endian byte ordering. The default byte and bit ordering is big-endian.

See "Byte Ordering," in Chapter 3, "Operand Conventions," of *The Programming Environments Manual* for more information about big- and little-endian byte ordering.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the natural address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned. For a detailed discussion about memory operands, see Chapter 3, "Operand Conventions," of *The Programming Environments Manual.*

2.3.2.3 Effective Address Calculation

An effective address (EA) is the 64-bit sum computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction. For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the memory operand is considered to wrap around from the maximum effective address through effective address 0, as described in the following paragraphs.

Effective address computations for both data and instruction accesses use 64-bit unsigned binary arithmetic. A carry from bit 0 is ignored.

Load and store operations have three categories of effective address generation:

- Register indirect with immediate index mode
- Register indirect with index mode
- Register indirect mode

Refer to Section 2.3.4.3.2, "Integer Load and Store Address Generation," for a detailed description of effective address generation for load and store operations.

Branch instructions have three categories of effective address generation:

- Immediate
- Link register indirect
- Count register indirect

2.3.2.4 Synchronization

The synchronization described in this section refers to the state of the processor that is performing the synchronization.

2.3.2.4.1 Context Synchronization

The System Call (**sc**), Return from Interrupt Double Word (**rfid**), and Return from Interrupt (**rfi**) instructions perform context synchronization by allowing previously issued instructions to complete before performing a change in context. Execution of one of these instructions ensures the following:

- No higher priority exception exists (**sc**).
- All previous instructions have completed to a point where they can no longer cause an exception. If a prior memory access instruction causes direct-store error exceptions, the results are guaranteed to be determined before this instruction is executed.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The instructions following the **sc**, **rfi,** and **rfid** instructions execute in the context established by these instructions.

2.3.2.4.2 Execution Synchronization

An instruction is execution synchronizing if all previously initiated instructions appear to have completed before the instruction is initiated or, in the case of **sync** and **isync**, before the instruction completes. For example, the Move to Machine State Register (**mtmsr**) and Move to Machine State Register Double Word (**mtmsrd**) instructions are execution synchronizing. It ensures that all preceding instructions have completed execution and will not cause an exception before the instruction executes, but does not ensure subsequent instructions execute in the newly established environment. For example, if **mtmsr** or **mtmsrd** sets the MSR[PR] bit, unless an **isync** immediately follows the **mtmsr** or **mtmsrd** instruction, a privileged instruction could be executed or privileged access could be performed without causing an exception even though the MSR[PR] bit indicates user mode.

2.3.2.4.3 Instruction-Related Exceptions

There are two kinds of exceptions in the 620—those caused directly by the execution of an instruction and those caused by an asynchronous event (or interrupts). Either may cause components of the system software to be invoked.

Exceptions can be caused directly by the execution of an instruction as follows:

- An attempt to execute an illegal instruction causes the illegal instruction (program exception) handler to be invoked. An attempt by a user-level program to execute the supervisor-level instructions listed below causes the privileged instruction (program exception) handler to be invoked. The 620 provides the following supervisor-level instructions—**dcbi**, **mfmsr**, **mfspr**, **mfsr**, **mfsrin**, **mtmsrd**, **mtmsr**, **mtspr**, **mtsrd**, **mtsrdin**, **mtsr**, **mtsrin**, **rfid**, **rfi**, **slbia**, **slbie**, **tlbie**, and **tlbsync**. Note that the privilege level of the **mfspr** and **mtspr** instructions depends on the SPR encoding.
- An attempt to access memory that is not available (page or segment fault) causes the ISI exception handler to be invoked.

- An attempt to access memory with an effective address alignment that is invalid for the instruction causes the alignment exception handler to be invoked.
- The execution of an **sc** instruction invokes the system call exception handler that permits a program to request the system to perform a service.
- The execution of a trap instruction invokes the program exception trap handler.
- The execution of a floating-point instruction when floating-point instructions are disabled invokes the floating-point unavailable handler.
- The execution of an instruction that causes a floating-point exception while exceptions are enabled in the MSR invokes the program exception handler.

Exceptions caused by asynchronous events are described in Chapter 4, "Exceptions."

2.3.3 Instruction Set Overview

This section provides a brief overview of the PowerPC instructions implemented in the 620 and highlights any special information with respect to how the 620 implements a particular instruction. Note that the categories used in this section correspond to those used in Chapter 4, "Addressing Modes and Instruction Set Summary," in *The Programming Environments Manual*. These categorizations are somewhat arbitrary and are provided for the convenience of the programmer and do not necessarily reflect the PowerPC architecture specification.

Note that some instructions have the following optional features:

- CR Update—The dot (**.**) suffix on the mnemonic enables the update of the CR.
- Overflow option—The **o** suffix indicates that the overflow bit in the XER is enabled.

2.3.4 PowerPC UISA Instructions

The PowerPC UISA includes the base user-level instruction set (excluding a few user-level cache control, synchronization, and time base instructions), user-level registers, programming model, data types, and addressing modes. This section discusses the instructions defined in the UISA.

2.3.4.1 Integer Instructions

This section describes the integer instructions. These consist of the following:

- Integer arithmetic instructions
- Integer compare instructions
- Integer logical instructions
- Integer rotate and shift instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs, into the XER, and into condition register (CR) fields.

2.3.4.1.1 Integer Arithmetic Instructions

Table 2-20 lists the integer arithmetic instructions for the PowerPC processors.

Table 2-20. Integer Arithmetic Instructions

Name	Mnemonic	Operand Syntax
Add Immediate	**addi**	rD,rA,SIMM
Add Immediate Shifted	**addis**	rD,rA,SIMM
Add	**add** (**add. addo addo.**)	rD,rA,rB
Subtract From	**subf** (**subf. subfo subfo.**)	rD,rA,rB
Add Immediate Carrying	**addic**	rD,rA,SIMM
Add Immediate Carrying and Record	**addic.**	rD,rA,SIMM
Subtract from Immediate Carrying	**subfic**	rD,rA,SIMM
Add Carrying	**addc** (**addc. addco addco.**)	rD,rA,rB
Subtract from Carrying	**subfc** (**subfc. subfco subfco.**)	rD,rA,rB
Add Extended	**adde** (**adde. addeo addeo.**)	rD,rA,rB
Subtract from Extended	**subfe** (**subfe. subfeo subfeo.**)	rD,rA,rB
Add to Minus One Extended	**addme** (**addme. addmeo addmeo.**)	rD,rA
Subtract from Minus One Extended	**subfme** (**subfme. subfmeo subfmeo.**)	rD,rA
Add to Zero Extended	**addze** (**addze. addzeo addzeo.**)	rD,rA
Subtract from Zero Extended	**subfze** (**subfze. subfzeo subfzeo.**)	rD,rA
Negate	**neg** (**neg. nego nego.**)	rD,rA
Multiply Low Immediate	**mulli**	rD,rA,SIMM
Multiply Low Word	**mullw** (**mullw. mullwo mullwo.**)	rD,rA,rB
Multiply Low Double Word	**mulld** (**mulld. mulldo mulldo.**)	rD,rA,rB
Multiply High Word	**mulhw** (**mulhw.**)	rD,rA,rB
Multiply High Double Word	**mulhd** (**mulhd.**)	rD,rA,rB
Multiply High Word Unsigned	**mulhwu** (**mulhwu.**)	rD,rA,rB
Multiply High Double Word Unsigned	**mulhdu** (**mulhdu.**)	rD,rA,rB
Divide Word	**divw** (**divw. divwo divwo.**)	rD,rA,rB
Divide Word Unsigned	**divwu** (**divwu. divwuo divwuo.**)	rD,rA,rB
Divide Double Word	**divd** (**divd. divdo divdo.**)	rD,rA,rB
Divide Double Word Unsigned	**divdu divdu. divduo divduo.**	rD,rA,rB

Although there is no Subtract Immediate instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. The **subf** instructions subtract the second operand (**r**A) from the third operand (**rB**). Simplified mnemonics are provided in which the third operand is subtracted from the second operand. See Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual* for examples.

The UISA states that for some implementations that execute instructions that set the overflow bit (OE) or the carry bit (CA) it may either execute these instructions slowly or it may prevent the execution of the subsequent instruction until the operation is complete. The 620 arithmetic instructions may suffer this penalty. The summary overflow bit (SO) and overflow bit (OV) in the XER are set to reflect an overflow condition of a 32-bit (executing **mullw**, **divw**, **divwu**) or 64-bit (executing **mulld**, **divd**, **divdu**) result. This may only occur when the overflow enable bit is set (OE = 1).

2.3.4.1.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of register **r**A with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of register **r**B. The comparison is signed for the **cmpi** and **cmp** instructions, and unsigned for the **cmpli** and **cmpl** instructions. Table 2-21 summarizes the integer compare instructions.

Table 2-21. Integer Compare Instructions

Name	Mnemonic	Operand Syntax
Compare Immediate	**cmpi**	**crf**D,L,**r**A,SIMM
Compare	**cmp**	**crf**D,L,**r**A,**r**B
Compare Logical Immediate	**cmpli**	**crf**D,L,**r**A,UIMM
Compare Logical	**cmpl**	**crf**D,L,**r**A,**r**B

The **crf**D operand can be omitted if the result of the comparison is to be placed in CR0. Otherwise the target CR field must be specified in the instruction **crf**D field, using an explicit field number.

For information on simplified mnemonics for the integer compare instructions see Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual.*

2.3.4.1.3 Integer Logical Instructions

The logical instructions shown in Table 2-22 perform bit-parallel operations on the specified operands. Logical instructions with the CR updating enabled (uses dot suffix) and instructions **andi.** and **andis.** set CR field CR0 to characterize the result of the logical operation. Logical instructions do not affect the XER[SO], XER[OV], and XER[CA] bits.

See Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual* for simplified mnemonic examples for integer logical operations.

Table 2-22. Integer Logical Instructions

Name	Mnemonic	Operand Syntax
AND Immediate	**andi.**	rA,rS,UIMM
AND Immediate Shifted	**andis.**	rA,rS,UIMM
OR Immediate	**ori**	rA,rS,UIMM
OR Immediate Shifted	**oris**	rA,rS,UIMM
XOR Immediate	**xori**	rA,rS,UIMM
XOR Immediate Shifted	**xoris**	rA,rS,UIMM
AND	**and** (**and.**)	rA,rS,rB
OR	**or** (**or.**)	rA,rS,rB
XOR	**xor** (**xor.**)	rA,rS,rB
NAND	**nand** (**nand.**)	rA,rS,rB
NOR	**nor** (**nor.**)	rA,rS,rB
Equivalent	**eqv** (**eqv.**)	rA,rS,rB
AND with Complement	**andc** (**andc.**)	rA,rS,rB
OR with Complement	**orc** (**orc.**)	rA,rS,rB
Extend Sign Byte	**extsb** (**extsb.**)	rA,rS
Extend Sign Half Word	**extsh** (**extsh.**)	rA,rS
Extend Sign Word	**extsw** (**extsw.**)	rA,rS
Count Leading Zeros Word	**cntlzw** (**cntlzw.**)	rA,rS
Count Leading Zeros Double Word	**cntlzd** (**cntlzd.**)	rA,rS

2.3.4.1.4 Integer Rotate and Shift Instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. See Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual* for a complete list of simplified mnemonics that allows simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and simple rotates and shifts.

Integer rotate instructions rotate the contents of a register. The result of the rotation is either inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register is unchanged), or ANDed with a mask before being placed into the target register.

The integer rotate instructions are summarized in Table 2-23.

Table 2-23. Integer Rotate Instructions

Name	Mnemonic	Operand Syntax
Rotate Left Double Word then Clear Left	**rldcl** (**rldcl.**)	rA,rS,rB,MB
Rotate Left Double Word then Clear Right	**rldcr** (**rldcr.**)	rA,rS,rB,ME
Rotate Left Double Word Immediate then Clear	**rldic** (**rldic.**)	rA,rS,SH,MB
Rotate Left Double Word Immediate then Clear Left	**rldicl** (**rldicl.**)	rA,rS,SH,MB
Rotate Left Double Word Immediate then Clear Right	**rldicr** (**rldicr.**)	rA,rS,SH,ME
Rotate Left Double Word Immediate then Mask Insert	**rldimi** (**rldimi.**)	rA,rS,SH,MB
Rotate Left Word Immediate then AND with Mask	**rlwinm** (**rlwinm.**)	rA,rS,SH,MB,ME
Rotate Left Word then AND with Mask	**rlwnm** (**rlwnm.**)	rA,rS,rB,MB,ME
Rotate Left Word Immediate then Mask Insert	**rlwimi** (**rlwimi.**)	rA,rS,SH,MB,ME

The integer shift instructions perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics (shown in Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual*) are provided to make coding of such shifts simpler and easier to understand.

Multiple-precision shifts can be programmed as shown in Appendix C, "Multiple-Precision Shifts," in *The Programming Environments Manual.* The integer shift instructions are summarized in Table 2-24.

Table 2-24. Integer Shift Instructions

Name	Mnemonic	Operand Syntax
Shift Left Double Word	**sld** (**sld.**)	rA,rS,rB
Shift Left Word	**slw** (**slw.**)	rA,rS,rB
Shift Right Algebraic Double Word	**srad** (**srad.**)	rA,rS,rB
Shift Right Algebraic Double Word Immediate	**sradi** (**sradi.**)	rA,rS,SH
Shift Right Double Word	**srd** (**srd.**)	rA,rS,rB
Shift Right Word	**srw** (**srw.**)	rA,rS,rB
Shift Right Algebraic Word Immediate	**srawi** (**srawi.**)	rA,rS,SH
Shift Right Algebraic Word	**sraw** (**sraw.**)	rA,rS,rB

2.3.4.2 Floating-Point Instructions

This section describes the floating-point instructions, which include the following:

- Floating-point arithmetic instructions
- Floating-point multiply-add instructions
- Floating-point rounding and conversion instructions
- Floating-point compare instructions
- Floating-point status and control register instructions
- Floating-point move instructions

See Section 2.3.4.3, "Load and Store Instructions," for information about floating-point loads and stores.

The PowerPC architecture supports a floating-point system as defined in the IEEE 754 standard, but requires software support to conform with that standard. All floating-point operations conform to the IEEE 754 standard, except if software sets the non-IEEE mode bit (NI) in the FPSCR.

2.3.4.2.1 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are summarized in Table 2-25.

Table 2-25. Floating-Point Arithmetic Instructions

Name	Mnemonic	Operand Syntax
Floating Add (Double-Precision)	**fadd** (**fadd.**)	**frD**,**frA**,**frB**
Floating Add Single	**fadds** (**fadds.**)	**frD**,**frA**,**frB**
Floating Subtract (Double-Precision)	**fsub** (**fsub.**)	**frD**,**frA**,**frB**
Floating Subtract Single	**fsubs** (**fsubs.**)	**frD**,**frA**,**frB**
Floating Multiply (Double-Precision)	**fmul** (**fmul.**)	**frD**,**frA**,**frC**
Floating Multiply Single	**fmuls** (**fmuls.**)	**frD**,**frA**,**frC**
Floating Divide (Double-Precision)	**fdiv** (**fdiv.**)	**frD**,**frA**,**frB**
Floating Divide Single	**fdivs** (**fdivs.**)	**frD**,**frA**,**frB**
Floating Square Root (Double-Precision)	**fsqrt** (**fsqrt.**)	**frD**,**frB**
Floating Square Root Single	**fsqrts** (**fsqrts.**)	**frD**,**frB**
Floating Reciprocal Estimate Single	**fres** (**fres.**)	**frD**,**frB**
Floating Reciprocal Square Root Estimate	**frsqrte** (**frsqrte.**)	**frD**,**frB**
Floating Select	**fsel**	**frD**,**frA**,**frC**,**frB**

All single-precision arithmetic instructions are performed using a double-precision format. The floating-point architecture is a single-pass implementation for double-precision products. In most cases, a single-precision instruction using only single-precision operands, in double-precision format, has the same latency as its double-precision equivalent.

2.3.4.2.2 Floating-Point Multiply-Add Instructions

These instructions combine multiply and add operations without an intermediate rounding operation. The floating-point multiply-add instructions are summarized in Table 2-26.

Table 2-26. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Operand Syntax
Floating Multiply-Add (Double-Precision)	**fmadd** (**fmadd.**)	frD,frA,frC,frB
Floating Multiply-Add Single	**fmadds** (**fmadds.**)	frD,frA,frC,frB
Floating Multiply-Subtract (Double-Precision)	**fmsub** (**fmsub.**)	frD,frA,frC,frB
Floating Multiply-Subtract Single	**fmsubs** (**fmsubs.**)	frD,frA,frC,frB
Floating Negative Multiply-Add (Double-Precision)	**fnmadd** (**fnmadd.**)	frD,frA,frC,frB
Floating Negative Multiply-Add Single	**fnmadds** (**fnmadds.**)	frD,frA,frC,frB
Floating Negative Multiply-Subtract (Double-Precision)	**fnmsub** (**fnmsub.**)	frD,frA,frC,frB
Floating Negative Multiply-Subtract Single	**fnmsubs** (**fnmsubs.**)	frD,frA,frC,frB

2.3.4.2.3 Floating-Point Rounding and Conversion Instructions

The Floating Round to Single-Precision (**frsp**) instruction is used to truncate a 64-bit double-precision number to a 32-bit single-precision floating-point number. The floating-point convert instructions convert a 64-bit double-precision floating-point number to a 32-bit signed integer number.

Examples of uses of these instructions to perform various conversions can be found in Appendix D, “Floating-Point Models,” in *The Programming Environments Manual.*

Table 2-27. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Operand Syntax
Floating Convert from Integer Double Word	**fcfid** (**fcfid.**)	frD,frB
Floating Convert to Integer Double Word	**fctid** (**fctid.**)	frD,frB
Floating Convert to Integer Double Word with Round toward Zero	**fctidz** (**fctidz.**)	frD,frB
Floating Round to Single	**frsp** (**frsp.**)	frD,frB
Floating Convert to Integer Word	**fctiw** (**fctiw.**)	frD,frB
Floating Convert to Integer Word with Round toward Zero	**fctiwz** (**fctiwz.**)	frD,frB

2.3.4.2.4 Floating-Point Compare Instructions

Floating-point compare instructions compare the contents of two floating-point registers. The comparison ignores the sign of zero (that is +0 = –0). The floating-point compare instructions are summarized in Table 2-28.

Table 2-28. Floating-Point Compare Instructions

Name	Mnemonic	Operand Syntax
Floating Compare Unordered	**fcmpu**	**crf**D,**fr**A,**fr**B
Floating Compare Ordered	**fcmpo**	**crf**D,**fr**A,**fr**B

Within the PowerPC architecture, an **fcmpu** or **fcmpo** instruction with the Rc bit set can cause an illegal instruction program exception or produce a boundedly undefined result. In the 620, **crf**D should be treated as undefined.

2.3.4.2.5 Floating-Point Status and Control Register Instructions

Every FPSCR instruction appears to synchronize the effects of all floating-point instructions executed by a given processor. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the given processor appear to have completed before the FPSCR instruction is initiated and that no subsequent floating-point instructions appear to be initiated by the given processor until the FPSCR instruction has completed. The FPSCR instructions are summarized in Table 2-29.

Table 2-29. Floating-Point Status and Control Register Instructions

Name	Mnemonic	Operand Syntax
Move from FPSCR	**mffs** (**mffs.**)	**fr**D
Move to Condition Register from FPSCR	**mcrfs**	**crf**D,**crf**S
Move to FPSCR Field Immediate	**mtfsfi** (**mtfsfi.**)	**crf**D,IMM
Move to FPSCR Fields	**mtfsf** (**mtfsf.**)	FM,**fr**B
Move to FPSCR Bit 0	**mtfsb0** (**mtfsb0.**)	**crb**D
Move to FPSCR Bit 1	**mtfsb1** (**mtfsb1.**)	**crb**D

2.3.4.2.6 Floating-Point Move Instructions

Floating-point move instructions copy data from one FPR to another. The floating-point move instructions do not modify the FPSCR. The CR update option in these instructions controls the placing of result status into CR1. Table 2-30 summarizes the floating-point move instructions.

Table 2-30. Floating-Point Move Instructions

Name	Mnemonic	Operand Syntax
Floating Move Register	**fmr** (**fmr.**)	**fr**D,**fr**B

Table 2-30. Floating-Point Move Instructions (Continued)

Floating Negate	**fneg** (**fneg.**)	**fr**D,**fr**B
Floating Absolute Value	**fabs** (**fabs.**)	**fr**D,**fr**B
Floating Negative Absolute Value	**fnabs** (**fnabs.**)	**fr**D,**fr**B

2.3.4.3 Load and Store Instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering. This section describes the load and store instructions, which consist of the following:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte reverse instructions
- Integer load and store multiple instructions
- Integer load and store string instructions
- Floating-point load instructions
- Floating-point store instructions

Implementation Notes—The following describes how the 620 handles misalignment:

- If an unaligned memory access crosses a 4-Kbyte page boundary, within a normal segment, an exception may occur when the boundary is crossed (that is, a protection violation occurs on the new page). In these cases, the 620 triggers a DSI exception and the instruction may have partially completed.
- Some misaligned memory accesses suffer performance degradation as compared to an aligned access of the same type. Memory accesses that cross a word boundary are broken into multiple discrete accesses by the load/store unit, except floating-point doubles aligned on a double-word boundary. Any noncacheable access that crosses a double-word boundary is broken into multiple external bus tenures.
- Any operation that crosses a word boundary (double word for floating-point doubles aligned on a double-word boundary) is broken into two accesses. Each of these accesses is translated. If either translation results in a data memory violation, a DSI exception is signaled. If two translations cross from T = 1 into T = 0 space (a programming error), the 620 completes all of the accesses for the operation, the segment information from the T = 1 space is presented on the bus for every access of the operation, and the 620 requires a direct-store protocol "Reply" from the device. Similarly, if two translations cross from T = 0 into T = 1 space, a DSI exception is not signaled.
- In the PowerPC architecture, the Rc bit must be zero for almost all load and store instructions. If the Rc bit is one, the instruction form is invalid. These include the integer load indexed instructions (**lbzx**, **lbzux**, **ldux**, **ldx**, **lhzx**, **lhzux**, **lhax**, **lhaux**, **lwaux**, **lwax**, **lwzx**, **lwzux**), the integer store indexed instructions (**stbx**, **stbux**,

stdux, stdx, sthx, sthux, stwx, stwux), the load and store with byte-reversal instructions (**lhbrx, lwbrx, sthbrx, stwbrx**), the string instructions (**lswi, lswx, stswi, stswx**), and the synchronization instructions (**sync, lwarx**). In the 620, executing one of these invalid instruction forms causes CR0 to be set to an undefined value. The floating-point load and store indexed instructions (**lfsx, lfsux, lfdx, lfdux, stfsx, stfsux, stfdx, stfdux**) are also invalid when the Rc bit is one. In the 620, executing one of these invalid instruction forms causes CR0 to be set to an undefined value.

2.3.4.3.1 Self-Modifying Code

When a processor modifies a memory location that may be contained in the instruction cache, software must ensure that memory updates are visible to the instruction fetching mechanism. This can be achieved by the following instruction sequence:

```
dcbst   |update memory
sync    |wait for update
icbi    |remove (invalidate) copy in instruction cache
sync    |wait for icbi to be globally performed
isync   |remove copy in own instruction buffer
```

These operations are required because the data cache is a write-back cache. Since instruction fetching bypasses the data cache, changes to items in the data cache may not be reflected in memory until the fetch operations complete.

Special care must be taken to avoid coherency paradoxes in systems that implement unified secondary caches, and designers should carefully follow the guidelines for maintaining cache coherency that are provided in the VEA, and discussed in Chapter 5, "Cache Model and Memory Coherency," in *The Programming Environments Manual.* Because the 620 does not broadcast the M bit for instruction fetches, external caches are subject to coherency paradoxes.

2.3.4.3.2 Integer Load and Store Address Generation

Integer load and store operations generate effective addresses using register indirect with immediate index mode, register indirect with index mode, or register indirect mode. See Section 2.3.2.3, "Effective Address Calculation," for information about calculating effective addresses. Note that in some implementations, operations that are not naturally aligned may suffer performance degradation. Refer to Section 4.6.6, "Alignment Exception (0x00600)," for additional information about load and store address alignment exceptions.

2.3.4.3.3 Register Indirect Integer Load Instructions

For integer load instructions, the byte, half word, word, or double word addressed by the EA (effective address) is loaded into **r**D. Many integer load instructions have an update form, in which **r**A is updated with the generated effective address. For these forms, if **r**A ≠ 0 and **r**A ≠ **r**D (otherwise invalid), the EA is placed into **r**A and the memory element (byte, half word, word, or double word) addressed by the EA is loaded into **r**D. Note that the PowerPC architecture defines load with update instructions with operand **r**A = 0 or **r**A = **r**D as invalid forms.

Implementation Notes—The following notes describe the 620 implementation of integer load instructions:

- In the PowerPC architecture, the Rc bit must be zero for almost all load and store instructions. If the Rc bit is one, the instruction form is invalid. These include the integer load indexed instructions (**lbzx**, **lbzux**, **ldux**, **ldx**, **lhzx**, **lhzux**, **lhax**, **lhaux**, **lwaux**, **lwax**, **lwzx**, and **lwzux**). In the 620, executing one of these invalid instruction forms causes CR0 to be set to an undefined value.
- For load with update instructions (**lbzu**, **lbzux**, **ldu**, **ldux**, **lhzu**, **lhzux**, **lhau**, **lhaux**, **lwzu**, **lwzux**, **lfsu**, **lfsux**, **lfdu**, **lfdux**), when **r**A = 0 or **r**A = **r**D the instruction form is considered invalid. If **r**A = 0, the 620 sets GPR0 to an undefined value. If **r**A = **r**D, the 620 sets **r**D to an undefined value.
- The PowerPC architecture cautions programmers that some implementations of the architecture may execute the Load Half Algebraic (**lha**, **lhax**) instructions with greater latency than other types of load instructions. This is not the case for the 620.

2

Table 2-31 summarizes the integer load instructions.

Table 2-31. Integer Load Instructions

Name	Mnemonic	Operand Syntax
Load Byte and Zero	**lbz**	rD,d(rA)
Load Byte and Zero Indexed	**lbzx**	rD,rA,rB
Load Byte and Zero with Update	**lbzu**	rD,d(rA)
Load Byte and Zero with Update Indexed	**lbzux**	rD,rA,rB
Load Double Word	**ld**	rD,d(rA)
Load Double Word with Update	**ldu**	rD,d(rA)
Load Double Word with Update Indexed	**ldux**	rD,rA,rB
Load Double Word Indexed	**ldx**	rD,rA,rB
Load Half Word and Zero	**lhz**	rD,d(rA)
Load Half Word and Zero Indexed	**lhzx**	rD,rA,rB
Load Half Word and Zero with Update	**lhzu**	rD,d(rA)
Load Half Word and Zero with Update Indexed	**lhzux**	rD,rA,rB
Load Half Word Algebraic	**lha**	rD,d(rA)
Load Half Word Algebraic Indexed	**lhax**	rD,rA,rB
Load Half Word Algebraic with Update	**lhau**	rD,d(rA)
Load Half Word Algebraic with Update Indexed	**lhaux**	rD,rA,rB
Load Word Algebraic	**lwa**	rD,d(rA)
Load Word Algebraic with Update Indexed	**lwaux**	rD,rA,rB
Load Word Algebraic with Indexed	**lwax**	rD,rA,rB

Table 2-31. Integer Load Instructions (Continued)

Name	Mnemonic	Operand Syntax
Load Word and Zero	**lwz**	rD,d(rA)
Load Word and Zero Indexed	**lwzx**	rD,rA,rB
Load Word and Zero with Update	**lwzu**	rD,d(rA)
Load Word and Zero with Update Indexed	**lwzux**	rD,rA,rB

2.3.4.3.4 Integer Store Instructions

For integer store instructions, the contents of **r**S are stored into the byte, half word, word or double word in memory addressed by the EA (effective address). Many store instructions have an update form, in which **r**A is updated with the EA. For these forms, the following rules apply:

- If **r**A ≠ 0, the effective address is placed into **r**A.
- If **r**S = **r**A, the contents of register **r**S are copied to the target memory element, then the generated EA is placed into **r**A (**r**S).

The PowerPC architecture defines store with update instructions with **r**A = 0 as an invalid form. In addition, it defines integer store instructions with the CR update option enabled (Rc field, bit 31, in the instruction encoding = 1) to be an invalid form. Table 2-32 summarizes the integer store instructions.

Table 2-32. Integer Store Instructions

Name	Mnemonic	Operand Syntax
Store Byte	**stb**	rS,**d**(rA)
Store Byte Indexed	**stbx**	rS,rA,rB
Store Byte with Update	**stbu**	rS,**d**(rA)
Store Byte with Update Indexed	**stbux**	rS,rA,rB
Store Double Word	**std**	rS,**d**(rA)
Store Double Word with Update	**stdu**	rS,**d**(rA)
Store Double Word with Update Indexed	**stdux**	rS,rA,rB
Store Double Word Indexed	**stdx**	rS,rA,rB
Store Half Word	**sth**	rS,**d**(rA)
Store Half Word Indexed	**sthx**	rS,rA,rB
Store Half Word with Update	**sthu**	rS,**d**(rA)
Store Half Word with Update Indexed	**sthux**	rS,rA,rB
Store Word	**stw**	rS,**d**(rA)
Store Word Indexed	**stwx**	rS,rA,rB

Table 2-32. Integer Store Instructions (Continued)

Name	Mnemonic	Operand Syntax
Store Word with Update	**stwu**	rS,d(rA)
Store Word with Update Indexed	**stwux**	rS,rA,rB

Implementation Notes—The following notes describe the 620 implementation of integer store instructions:

- In the PowerPC architecture, the Rc bit must be zero for almost all load and store instructions. If the Rc bit is one, the instruction form is invalid. These include the integer store indexed instructions (**stbx**, **stbux**, **stdux, stdx, sthx**, **sthux**, **stwx**, **stwux**). In the 620, executing one of these invalid instruction forms causes CR0 to be set to an undefined value.
- For the store with update instructions (**stbu**, **stbux**, **stdu, stdux, sthu**, **sthux**, **stwu**, **stwux**, **stfsu**, **stfsux**, **stfdu**, **stfdux**), when **r**A = 0, the instruction form is considered invalid. In this case, the 620 sets GPR0 to an undefined value.

2.3.4.3.5 Integer Load and Store with Byte Reverse Instructions

Table 2-33 describes integer load and store with byte reverse instructions. When used in a PowerPC system operating with the default big-endian byte order, these instructions have the effect of loading and storing data in little-endian order. Likewise, when used in a PowerPC system operating with little-endian byte order, these instructions have the effect of loading and storing data in big-endian order. For more information about big-endian and little-endian byte ordering, see Section 3.2.2, "Byte Ordering," in *The Programming Environments Manual.*

Implementation Note—In the PowerPC architecture, the Rc bit must be zero for almost all load and store instructions. If the Rc bit is one, the instruction form is invalid. These include the load and store with byte-reverse instructions (**lhbrx**, **lwbrx**, **sthbrx**, **stwbrx**). In the 620, executing one of these invalid instruction forms causes CR0 to be set to an undefined value.

Table 2-33. Integer Load and Store with Byte Reverse Instructions

Name	Mnemonic	Operand Syntax
Load Half Word Byte-Reverse Indexed	**lhbrx**	rD,rA,rB
Load Word Byte-Reverse Indexed	**lwbrx**	rD,rA,rB
Store Half Word Byte-Reverse Indexed	**sthbrx**	rS,rA,rB
Store Word Byte-Reverse Indexed	**stwbrx**	rS,rA,rB

2.3.4.3.6 Integer Load and Store Multiple Instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs. The load multiple and store multiple instructions may have operands that require memory accesses crossing a 4-Kbyte page boundary. As a result, these instructions may be interrupted by a DSI exception associated with the address translation of the second page.

Implementation Notes—The following describes the 620 implementation of the load/store multiple instruction:

- The PowerPC architecture requires that memory operands for Load Multiple and Store Multiple instructions (**lmw** and **stmw**) be word-aligned. If the operands to these instructions are not word-aligned, an alignment exception occurs. The 620 provides hardware support for **lmw** and **stmw** instructions that cross a page boundary. However, a DSI exception may occur when the boundary is crossed (for example, if a protection violation occurs on the new page).
- Executing an **lmw** instruction in which **r**A is in the range of registers to be loaded or in which **r**A = **r**D = 0 is invalid in the architecture. In the 620, all registers loaded are set to undefined values. Any exceptions resulting from a memory access cause the system error handler normally associated with the exception to be invoked.
- The 620's implementation of the **lmw** instruction allows one double word of data to be transferred to the GPRs per internal clock cycle (that is, one register is filled per clock) whenever the data is found in the cache. For the **stmw** instruction, data is transferred from the GPRs to the cache at a rate of one double word (GPR) per clock cycle.
- When an **lmw** or **stmw** access is to noncacheable memory, data is transferred on the external bus at a rate of one double word per external bus tenure.
- The load multiple instruction can be interrupted after the instruction has partially completed. If **r**A has been modified and the instruction is restarted, the instruction begins loading from the addresses specified by the new value of **r**A, which might be anywhere in memory; therefore, the system error handler may be invoked.

The PowerPC architecture defines the load multiple word (**lmw**) instruction with **r**A in the range of registers to be loaded as an invalid form.

Table 2-34. Integer Load and Store Multiple Instructions

Name	Mnemonic	Operand Syntax
Load Multiple Word	**lmw**	rD,d(rA)
Store Multiple Word	**stmw**	rS,d(rA)

2.3.4.3.7 Integer Load and Store String Instructions

The integer load and store string instructions allow movement of data from memory to registers or from registers to memory without concern for alignment. These instructions can be used for a short move between arbitrary memory locations or to initiate a long move between misaligned memory fields. However, in some implementations, these instructions are likely to have greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results. Table 2-35 summarizes the integer load and store string instructions.

In other PowerPC implementations operating with little-endian byte order, execution of a load or string instruction causes the system alignment error handler to be invoked; see Section 3.2.2, "Byte Ordering," in *The Programming Environments Manual* for more information.

Table 2-35. Integer Load and Store String Instructions

Name	Mnemonic	Operand Syntax
Load String Word Immediate	**lswi**	rD,rA,NB
Load String Word Indexed	**lswx**	rD,rA,rB
Store String Word Immediate	**stswi**	rS,rA,NB
Store String Word Indexed	**stswx**	rS,rA,rB

Load string and store string instructions may involve operands that are not word-aligned.

As described in Section 4.6.6, "Alignment Exception (0x00600)," a misaligned string operation suffers a performance penalty compared to an aligned operation of the same type. A non–word-aligned string operation that crosses a 4-Kbyte boundary, or a double-word-aligned string operation that crosses a 256-Mbyte boundary always causes an alignment exception. A non–word-aligned string operation that crosses a double-word boundary is also slower than a word-aligned string operation.

Implementation Note—The following describes the 620 implementation of the load/store string instruction:

- The 620 provides hardware support for **lswi**, **lswx**, **stswi**, and **stswx** instructions to cross a page boundary. However, a DSI exception may occur when the boundary is crossed (for example, if a protection violation occurs on the new page).
- An **lswi** or **lswx** instruction in which **r**A or **r**B is in the range of registers potentially to be loaded or in which **r**A = **r**D = 0 is an invalid instruction form. In the 620, all registers loaded are set to undefined values. Any exceptions resulting from a memory access cause the system error handler normally associated with the exception to be invoked.
- The load multiple and load string instructions can be interrupted after the instruction has partially completed. If **r**A has been modified and the instruction is restarted, the instruction begins loading from the addresses specified by the new value of **r**A, which might be anywhere in memory; therefore, the system error handler may be invoked.
- The load string instructions can be interrupted after the instruction has partially completed. If **r**A has been modified and the instruction is restarted, the instruction begins loading from the addresses specified by the new value of **r**A, which might be anywhere in memory; therefore, the system error handler may be invoked.

2.3.4.3.8 Floating-Point Load and Store Address Generation

Floating-point load and store operations generate effective addresses using the register indirect with immediate index addressing mode and register indirect with index addressing mode. Floating-point loads and stores are not supported for direct-store accesses. The use of floating-point loads and stores for direct-store access results in an alignment exception.

There are two forms of the floating-point load instruction—single-precision and double-precision operand formats. Because the FPRs support only the floating-point double-precision format, single-precision floating-point load instructions convert single-precision data to double-precision format before loading the operands into the target FPR.

Note that the PowerPC architecture defines load with update instructions with **r**A = 0 as an invalid form.

Table 2-36. Floating-Point Load Instructions

Name	Mnemonic	Operand Syntax
Load Floating-Point Single	**lfs**	**fr**D,**d**(**r**A)
Load Floating-Point Single Indexed	**lfsx**	**fr**D,**r**A,**r**B
Load Floating-Point Single with Update	**lfsu**	**fr**D,**d**(**r**A)
Load Floating-Point Single with Update Indexed	**lfsux**	**fr**D,**r**A,**r**B
Load Floating-Point Double	**lfd**	**fr**D,**d**(**r**A)
Load Floating-Point Double Indexed	**lfdx**	**fr**D,**r**A,**r**B
Load Floating-Point Double with Update	**lfdu**	**fr**D,**d**(**r**A)
Load Floating-Point Double with Update Indexed	**lfdux**	**fr**D,**r**A,**r**B

2.3.4.3.9 Floating-Point Store Instructions

This section describes floating-point store instructions. There are three basic forms of the store instruction—single-precision, double-precision, and integer. The integer form is supported by the optional **stfiwx** instruction. Because the FPRs support only floating-point, double-precision format for floating-point data, single-precision floating-point store instructions convert double-precision data to single-precision format before storing the operands. Table 2-37 summarizes the floating-point store instructions.

Table 2-37. Floating-Point Store Instructions

Name	Mnemonic	Operand Syntax
Store Floating-Point Single	**stfs**	**frS,d(r**A)
Store Floating-Point Single Indexed	**stfsx**	**frS,r** B
Store Floating-Point Single with Update	**stfsu**	**frS,d(r**A)
Store Floating-Point Single with Update Indexed	**stfsux**	**frS,r** B
Store Floating-Point Double	**stfd**	**frS,d(rA)**
Store Floating-Point Double Indexed	**stfdx**	**frS,rB**
Store Floating-Point Double with Update	**stfdu**	**frS,d(r**A)
Store Floating-Point Double with Update Indexed	**stfdux**	**frS,r** B
Store Floating-Point as Integer Word Indexed	**stfiwx**	**frS,rB**

Some floating-point store instructions require conversions in the LSU. Table 2-38 shows the conversions made by the LSU when performing a Store Floating-Point Single instruction.

Table 2-38. Store Floating-Point Single Behavior

FPR Precision	Data Type	Action
Single	Normalized	Store
Single	Denormalized	Store
Single	Zero Infinity QNaN	Store
Single	SNaN	Store

Double	Normalized	If(exp ≤ 896)
		then Denormalize and Store
		else
		Store

Double	Denormalized	Store Zero
Double	Zero Infinity QNaN	Store
Double	SNaN	Store

Table 2-39 shows the conversions made when performing a Store Floating-Point Double instruction. Most entries in the table indicate that the floating-point value is simply stored. Only in a few cases are any other actions taken.

Table 2-39. Store Floating-Point Double Behavior

FPR Precision	Data Type	Action
Single	Normalized	Store
Single	Denormalized	Normalize and Store
Single	Zero Infinity QNaN	Store
Single	SNaN	Store
Double	Normalized	Store
Double	Denormalized	Store
Double	Zero Infinity QNaN	Store
Double	SNaN	Store

Architecturally, all floating-point numbers are represented in double-precision format within the 620. Execution of a store floating-point single (**stfs**, **stfsu**, **stfsx**, **stfsux**) instruction requires conversion from double- to single-precision format. If the exponent is not greater than 896, this conversion requires denormalization.

Because of how floating-point numbers are implemented in the 620, there is also a case when execution of a store floating-point double (**stfd**, **stfdu**, **stfdx**, **stfdux**) instruction can require internal shifting of the mantissa. This case occurs when the operand of a store floating-point double instruction is a denormalized single-precision value. The value could be the result of a load floating-point single instruction, a single-precision arithmetic instruction, or a floating round to single-precision instruction.

2.3.4.4 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. When the processor encounters one of these instructions, it scans the execution pipelines to determine whether an instruction in progress may affect the particular CR bit. If no interlock is found, the branch can be resolved immediately by checking the bit in the CR and taking the action defined for the branch instruction.

2.3.4.4.1 Branch Instruction Address Calculation

Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word aligned; the PowerPC processors ignore the two low-order bits of the generated branch target address.

Branch instructions compute the effective address (EA) of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register
- Branch conditional to count register

Note that in the 620, all branch instructions (**b**, **ba**, **bl**, **bla**, **bc**, **bca**, **bcl**, **bcla**, **bclr**, **bclrl**, **bcctr**, **bcctrl**) and condition register logical instructions (**crand**, **cror**, **crxor**, **crnand**, **crnor**, **crandc**, **creqv**, **crorc**, and **mcrf**) are executed by the BPU. Some of these instructions can redirect instruction execution conditionally based on the value of bits in the CR. Whenever the CR bits resolve, the branch direction is either marked as correct or mispredicted. Correcting a mispredicted branch requires that the 620 flush speculatively executed instructions and restore the machine state to immediately after the branch. This correction can be done immediately upon resolution of the condition registers bits.

2.3.4.4.2 Branch Instructions

Table 2-40 lists the branch instructions provided by the PowerPC processors. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. See Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual* for a list of simplified mnemonic examples.

Table 2-40. Branch Instructions

Name	Mnemonic	Operand Syntax
Branch	**b (ba bl bla)**	target_addr
Branch Conditional	**bc (bca bcl bcla)**	BO,BI,target_addr
Branch Conditional to Link Register	**bclr (bclrl)**	BO,BI
Branch Conditional to Count Register	**bcctr (bcctrl)**	BO,BI

2.3.4.4.3 Condition Register Logical Instructions

Condition register logical instructions, shown in Table 2-41, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 2-41. Condition Register Logical Instructions

Name	Mnemonic	Operand Syntax
Condition Register AND	**crand**	**crb**D,**crb**A,**crb**B
Condition Register OR	**cror**	**crb**D,**crb**A,**crb**B
Condition Register XOR	**crxor**	**crb**D,**crb**A,**crb**B
Condition Register NAND	**crnand**	**crb**D,**crb**A,**crb**B
Condition Register NOR	**crnor**	**crb**D,**crb**A,**crb**B
Condition Register Equivalent	**creqv**	**crb**D,**crb**A, **crb**B
Condition Register AND with Complement	**crandc**	**crb**D,**crb**A, **crb**B
Condition Register OR with Complement	**crorc**	**crb**D,**crb**A, **crb**B
Move Condition Register Field	**mcrf**	**crf**D,**crf**S

Note that if the LR update option is enabled for any of these instructions, the PowerPC architecture defines these forms of the instructions as invalid.

2.3.4.4.4 Trap Instructions

The trap instructions shown in Table 2-42 are provided to test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap handler is invoked. If the tested conditions are not met, instruction execution continues normally.

Table 2-42. Trap Instructions

Name	Mnemonic	Operand Syntax
Trap Double Word	**td**	TO,**r**A,**r**B
Trap Double Word Immediate	**tdi**	TO,**r**A,SIMM
Trap Word Immediate	**twi**	TO,**r**A,SIMM
Trap Word	**tw**	TO,**r**A,**r**B

See Appendix F, “Simplified Mnemonics,” in *The Programming Environments Manual* for a complete set of simplified mnemonics.

2.3.4.5 System Linkage Instruction

This section describes the System Call (**sc**) instruction that permits a program to call on the system to perform a service. See also Section 2.3.6.1, “System Linkage Instructions,” for additional information.

Table 2-43. System Linkage Instruction—UISA

Name	Mnemonic	Operand Syntax
System Call	**sc**	—

2.3.4.6 Processor Control Instructions

Processor control instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs). See Section 2.3.5.1, "Processor Control Instructions," for the **mftb** instruction and Section 2.3.6.2, "Processor Control Instructions," for information about the instructions used for reading from and writing to the MSR and SPRs.

2.3.4.6.1 Move to/from Condition Register Instructions

Table 2-44 summarizes the instructions for reading from or writing to the condition register.

Table 2-44. Move to/from Condition Register Instructions

Name	Mnemonic	Operand Syntax
Move to Condition Register Fields	**mtcrf**	CRM,rS
Move to Condition Register from XER	**mcrxr**	crfD
Move from Condition Register	**mfcr**	rD

Note that the performance of the **mtcrf** instruction depends greatly on whether only one field is being accessed or either no fields or multiple fields are accessed as follows:

- Those **mtcrf** instructions that update only one field are executed in either of the SCIUs and the CR field is renamed as with any other SCIU instruction.
- Those **mtcrf** instructions that update either multiple fields or no fields are dispatched to the MCIU and a count/link scoreboard bit is set. When that bit is set, no more **mtcrf** instructions of the same type, **mtspr** instructions that update the count or link registers, branch instructions that depend on the condition register and CR logical instructions can be dispatched to the MCIU. The bit is cleared when the **mtctr**, **mtcrf**, or **mtlr** instruction that the bit is executed.

Because **mtcrf** instructions that update a single field do not require such synchronization that other **mtcrf** instructions do, and because two such single-field instructions can execute in parallel, it is typically more efficient to use multiple **mtcrf** instructions that update only one field apiece than to use one **mtcrf** instruction that updates multiple fields. A rule of thumb follows:

- It is always more efficient to use two **mtcrf** instructions that update only one field apiece than to use one **mtcrf** instruction that updates two fields.
  - It is almost always more efficient to use three or four **mtcrf** instructions that update only one field apiece than to use one **mtcrf** instruction that updates three fields.
  - It is often more efficient to use more than four **mtcrf** instructions that update only one field than to use one **mtcrf** instruction that updates four fields.

2.3.4.6.2 Move to/from Special-Purpose Register Instructions

Table 2-45 lists the **mtspr** and **mfspr** instructions.

Table 2-45. Move to/from Special-Purpose Register Instructions (UISA)

Name	Mnemonic	Operand Syntax
Move to Special Purpose Register	**mtspr**	SPR,rS
Move from Special Purpose Register	**mfspr**	rD,SPR

2.3.4.7 Memory Synchronization Instructions

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. See Chapter 3, "Instruction and Data Cache Operation," for additional information about these instructions and about related aspects of memory synchronization.

Table 2-46. Memory Synchronization Instructions—UISA

Name	Mnemonic	Operand Syntax
Load Double Word and Reserve Indexed	**ldarx**	rD,rA,rB
Load Word and Reserve Indexed	**lwarx**	rD,rA,rB
Store Double Word Conditional Indexed	**stdcx.**	rS,rA,rB
Store Word Conditional Indexed	**stwcx.**	rS,rA,rB
Synchronize	**sync**	—

The proper paired use of the **ldarx** and **lwarx** instructions with the **stdcx.** and **stwcx.** instructions allows programmers to emulate common semaphore operations such as "test and set," "compare and swap," "exchange memory," and "fetch and add." The **ldarx** or **lwarx** instruction must be paired with an **stdcx.** or **stwcx.** instruction with the same effective address used for both instructions of the pair. Note that the reservation granularity is implementation-dependent. See 2.3.5.2, "Memory Synchronization Instructions," for details about additional memory synchronization (**eieio** and **isync**) instructions.

Implementation Notes—The following notes describe the 620 implementation of memory synchronization instructions:

- The PowerPC architecture requires that memory operands for Load and Reserve (**ldarx** or **lwarx**) and Store Conditional (**sdwcx.** or **stwcx.**) instructions must be word-aligned. If the operands to these instructions are not word-aligned on the 620, an alignment exception occurs.
- The PowerPC architecture indicates that the granularity with which reservations for **ldarx**, **lwarx**, **stdcx.**, and **stwcx.** instructions are managed is implementation-dependent. In the 620 reservations, this granularity is a 64-byte cache block.

- The **sync** instruction causes the 620 to serialize. The **sync** instruction can be dispatched with other instructions that are before it, in program order. However, no more instructions can be dispatched until the **sync** instruction completes. Instructions already in the instruction buffer, due to prefetching, are not refetched after the **sync** completes. If reflecting is required, **isync** should be executed to flush the instruction buffer after the **sync**. The **sync** is dispatched to the LSU and is broadcast onto the external bus.

In the PowerPC architecture, the Rc bit must be zero for almost all load and store instructions. If the Rc bit is one, the instruction form is invalid. These include the **sync**, **ldarx**, and **lwarx** instructions. In the 620, executing one of these invalid instruction forms causes CR0 to be set to an undefined value. The **stdcx.** and **stwcx.** instructions are the only load/store instructions that has a valid form if Rc is set. If the Rc bit is zero, the result of executing these instructions in the 620 causes CR0 to be set to an undefined value.

2.3.5 PowerPC VEA Instructions

The PowerPC virtual environment architecture (VEA) describes the semantics of the memory model that can be assumed by software processes, and includes descriptions of the cache model, cache control instructions, address aliasing, and other related issues. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

This section describes additional instructions that are provided by the VEA.

2.3.5.1 Processor Control Instructions

In addition to the move to condition register instructions (specified by the UISA), the VEA defines the **mftb** instruction (user-level instruction) for reading the contents of the time base register; see Chapter 3, “Instruction and Data Cache Operation,” for more information. Table 3-34 shows the **mftb** instruction.

Table 2-47. Move from Time Base Instruction

Name	Mnemonic	Operand Syntax
Move from Time Base	**mftb**	**r**D, TBR

Simplified mnemonics are provided for the **mftb** instruction so it can be coded with the TBR name as part of the mnemonic rather than requiring it to be coded as an operand. See Appendix F, “Simplified Mnemonics,” in *The Programming Environments Manual* for simplified mnemonic examples and for simplified mnemonics for Move from Time Base (**mftb**) and Move from Time Base Upper (**mftbu**), which are variants of the **mftb** instruction rather than of **mfspr**. The **mftb** instruction serves as both a basic and simplified mnemonic. Assemblers recognize an **mftb** mnemonic with two operands as the basic form, and an **mftb** mnemonic with one operand as the simplified form.

Implementation Notes—The following information is useful with respect to using the time base implementation in the 620:

- The 620 allows user-mode read access to the time base counter through the use of the Move from Time Base (**mftb**) and the Move from Time Base Upper (**mftbu**) instructions. As a 64-bit PowerPC implementation, the 620 can access the entire TB register at once.
- The time base counter is clocked at the bus clock frequency. Counting is enabled by assertion of the time base enable ($\overline{\text{TBENABLE}}$) input signal.

2.3.5.2 Memory Synchronization Instructions

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. See Chapter 3, "Instruction and Data Cache Operation," for additional information about these instructions and about related aspects of memory synchronization.

Table 2-48 describes the memory synchronization instructions defined by the VEA.

Table 2-48. Memory Synchronization Instructions—VEA

Name	Mnemonic	Operand Syntax	Implementation Notes
Enforce In-Order Execution of I/O	**eieio**	—	The **eieio** instruction is dispatched by the 620 to the LSU. The **eieio** instruction executes after all preceding cache-inhibited or write-through memory instructions execute; all following cache-inhibited or write-through instructions execute after the **eieio** instruction executes. When the **eieio** instruction executes, an EIEIO address-only operation is broadcast on the external bus to allow ordering to be enforced in the external memory system.
Instruction Synchronize	**isync**	—	The **isync** instruction causes the 620 to purge its instruction buffers and fetch the double word containing the next sequential instruction.

System designs that use a second-level cache should take special care to recognize the hardware signaling caused by a SYNC bus operation and perform the appropriate actions to guarantee that memory references that may be queued internally to the second-level cache have been performed globally.

In addition to the **sync** instruction (specified by UISA), the VEA defines the Enforce In-Order Execution of I/O (**eieio**) and Instruction Synchronize (**isync**) instructions. The number of cycles required to complete an **eieio** instruction depends on system parameters and on the processor's state when the instruction is issued. As a result, frequent use of this instruction may degrade performance slightly.

The **isync** instruction causes the processor to wait for any preceding instructions to complete, discard all prefetched instructions, and then branch to the next sequential instruction (which has the effect of clearing the pipeline behind the **isync** instruction).

2.3.5.3 Memory Control Instructions

Memory control instructions include the following types:

- Cache management instructions (user-level and supervisor-level)
- Segment register manipulation instructions
- Translation lookaside buffer management instructions

This section describes the user-level cache management instructions defined by the VEA. See 2.3.6.3, "Memory Control Instructions," for information about supervisor-level cache, segment register manipulation, and translation lookaside buffer management instructions.

The instructions summarized in this section provide user-level programs the ability to manage on-chip caches if they are implemented. See Chapter 3, "Instruction and Data Cache Operation," for more information about cache topics.

The user-level cache instructions provide software a way to help manage processor caches. The following sections describe how these operations are treated with respect to the 620's cache.

As with other memory-related instructions, the effect of the cache management instructions on memory are weakly-ordered. If the programmer needs to ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, a **sync** instruction must be placed in the program following those instructions.

Note that this discussion does not apply to direct-store segment accesses because these are defined to be cache-inhibited and instruction fetch from them is not allowed. Cache operations that access direct-store segment are treated as no-ops. Table 2-49 summarizes the cache instructions defined by the VEA. Note that these instructions are accessible to user-level programs.

Table 2-49. User-Level Cache Instructions

Name	Mnemonic	Operand Syntax	Implementation Notes
Data Cache Block Touch	**dcbt**	rA,rB	The VEA defines this instruction to allow for potential system performance enhancements through the use of software-initiated prefetch hints. Implementations are not required to take any action based off the execution of this instruction, but they may choose to prefetch the cache block corresponding to the effective address into their cache. The 620 performs the prefetch when the address hits in the TLB or the BAT, is permitted load access from the addressed page, is not directed to a direct-store segment, and is directed at a cacheable page. If the operation does not meet these criteria, it is treated as a no-op. The data brought into the cache as a result of this instruction is validated in the same way a load instruction would be (that is, if no other bus participant has a copy, it is marked as Exclusive, otherwise it is marked as Shared). The memory reference of a **dcbt** causes the reference bit to be set. A successful **dcbt** instruction affects the state of the TLB and cache LRU bits as defined by the LRU algorithm.
Data Cache Block Touch for Store	**dcbtst**	rA,rB	This instructions behaves like the **dcbt** instruction.
Data Cache Block Set to Zero	**dcbz**	rA,rB	The effective address is computed, translated, and checked for protection violations as defined in the VEA. If the 620 does not have exclusive access to the block, it presents an operation onto the 620 bus interface that instructs all other processors to invalidate copies of the block that may reside in their cache (this is the kill operation on the bus). After it has exclusive access, the 620 writes all zeros into the cache block. If the 620 already has exclusive access, it immediately writes all zeros into the cache block. If the addressed block is within a noncacheable or a write-through page, or if the cache is locked or disabled, an alignment exception occurs. If the operation is successful, the cache block is marked modified.
Data Cache Block Store	**dcbst**	rA,rB	The effective address is computed, translated, and checked for protection violations as defined in the VEA. If the 620 does not have exclusive access to the block, it broadcasts the essence of the instruction onto the 620 bus (using the clean operation, described in Table 3-5). If the 620 has modified data associated with the block, the processor pushes the modified data out of the cache and into the memory queue for future arbitration onto the 620 bus. In this situation, the cache block is marked exclusive. Otherwise this instruction is treated as a no-op.
Data Cache Block Flush	**dcbf**	rA,rB	The effective address is computed, translated, and checked for protection violations as defined by the VEA. If the 620 does not have exclusive access to the block, it broadcasts the essence of the instruction onto the 620 bus. In addition, if the addressed block is present in the cache, the 620 marks this data as invalid. On the other hand, if the 620 has modified data associated with the block, the processor pushes the modified data out of the cache and into the memory queue for future arbitration onto the 620 bus. In this situation, the cache block is marked invalid.

Table 2-49. User-Level Cache Instructions (Continued)

Name	Mnemonic	Operand Syntax	Implementation Notes
Instruction Cache Block Invalidate	**icbi**	rA,rB	The effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. If the addressed block is in the instruction cache, the 620 marks it invalid. This instruction changes neither the content nor status of the data cache. In addition, the ICBI operation is broadcast on the 620 bus unconditionally to support this function throughout multilayer memory hierarchy.

2.3.5.4 Optional External Control Instructions

The external control instructions allow a user-level program to communicate with a special-purpose device. Two instructions are provided and are summarized in Table 2-50.

Table 2-50. External Control Instructions

Name	Mnemonic	Operand Syntax
External Control In Word Indexed	**eciwx**	rD,rA,rB
External Control Out Word Indexed	**ecowx**	rS,rA,rB

The **eciwx** and **ecowx** instructions should be word-aligned. Misaligned **eciwx** and **ecowx** instructions are treated like cache-inhibited accesses, and may be split into two bus transactions.

2.3.6 PowerPC OEA Instructions

The PowerPC operating environment architecture (OEA) includes the structure of the memory management model, supervisor-level registers, and the exception model. Implementations that conform to the OEA also adhere to the UISA and the VEA. This section describes the instructions provided by the OEA.

2.3.6.1 System Linkage Instructions

This section describes the system linkage instructions (see Table 2-51). The **sc** instruction is a user-level instruction that permits a user program to call on the system to perform a service and causes the processor to take an exception. The **rfi** and **rfid** instructions are supervisor-level instructions that are useful for returning from an exception handler.

Table 2-51. System Linkage Instructions—OEA

Name	Mnemonic	Operand Syntax
System Call	**sc**	—
Return from Interrupt Double Word	**rfid**	—
Return from Interrupt	**rfi**	—

2.3.6.2 Processor Control Instructions

This section describes the processor control instructions that are used to read from and write to the MSR and the SPRs.

Table 2-52 summarizes the instructions used for reading from and writing to the MSR.

Table 2-52. Move to/from Machine State Register Instructions

Name	Mnemonic	Operand Syntax
Move to Machine State Register Double Word	**mtmsrd**	rS
Move to Machine State Register	**mtmsr**	rS
Move from Machine State Register	**mfmsr**	rD

The OEA defines encodings of the **mtspr** and **mfspr** instructions to provide access to supervisor-level registers. The instructions are listed in Table 2-53.

Table 2-53. Move to/from Special-Purpose Register Instructions (OEA)

Name	Mnemonic	Operand Syntax
Move to Special Purpose Register	**mtspr**	SPR,rS
Move from Special Purpose Register	**mfspr**	rD,SPR

Encodings for the 620-specific SPRs are listed in Table 2-54.

Table 2-54 SPR Encodings for 620-Defined Registers (mfspr)

SPR[1]			Register Name
Decimal	spr[5–9]	spr[0–4]	
795	11000	11011	MMCR0
798	11000	11110	MMCR1
787	11000	10011	PMC1
788	11000	10100	PMC2
789	11000	10101	PMC3
790	11000	10110	PMC4
791	11000	10111	PMC5
792	11000	11000	PMC6
793	11000	11001	PMC7
794	11000	11010	PMC8
790	11000	01100	SIA
781	11000	01101	SDA
1008	11111	10000	HID0

Table 2-54 SPR Encodings for 620-Defined Registers (mfspr) (Continued)

SPR[1]			Register Name
Decimal	spr[5–9]	spr[0–4]	
1010	11111	10010	IABR
1016	11111	11000	BUSCSR
1017	11111	11001	L2CR
1018	11111	11010	L2SR
1023	11111	11111	PIR

[1] Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.

For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16–20 of the instruction and the low-order 5 bits in bits 11–15.

Simplified mnemonics are provided for the **mtspr** and **mfspr** instructions in Appendix F, "Simplified Mnemonics," in *The Programming Environments Manual.* For a discussion of context synchronization requirements when altering certain SPRs, refer to Appendix E, "Synchronization Programming Examples," in *The Programming Environments Manual.*

For information on SPR encodings (both user- and supervisor-level) see Chapter 8, "Instruction Set," in *The Programming Environments Manual.* Note that there are additional SPRs specific to each implementation; for implementation-specific SPRs, see the user's manual for that particular processor.

2.3.6.3 Memory Control Instructions

Memory control instructions include the following types of instructions:

- Cache management instructions (supervisor-level and user-level)
- Segment register manipulation instructions
- Translation lookaside buffer management instructions

This section describes supervisor-level memory control instructions. See Section 2.7.3, "Memory Control Instructions—VEA," for more information about user-level cache management instructions.

2.3.6.3.1 Supervisor-Level Cache Management Instruction

Table 2-55 lists the only supervisor-level cache management instruction.

Table 2-55. Cache Management Supervisor-Level Instruction

Name	Mnemonic	Operand Syntax	Implementation Notes
Data Cache Block Invalidate	**dcbi**	rA,rB	The EA is computed, translated, and checked for protection violations as defined in the OEA. The 620 broadcasts the essence of the instruction onto the 620 bus (using the Kill operation). In addition, if the addressed block is present in the cache, the 620 marks this data as invalid regardless of whether the data is clean or modified. Note that this can have the effect of destroying modified data which is why the instruction is privileged and has store semantics with respect to protection.

See Section 2.7.3.1, "User-Level Cache Instructions—VEA," for cache instructions that provide user-level programs the ability to manage the on-chip caches. If the effective address references a direct-store segment, the instruction is treated as a no-op. Note that any cache control instruction that generates an effective address that corresponds to a direct-store segment (segment descriptor[T] = 1) is treated as a no-op.

2.3.6.3.2 Segment Register Manipulation Instructions

The instructions listed in Table 2-56 provide access to the segment registers. These instructions operate completely independently of the MSR[IR] and MSR[DR] bit settings. Refer to "Synchronization Requirements for Special Registers and for Lookaside Buffers," in Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual* for serialization requirements and other recommended precautions to observe when manipulating the segment registers.

Table 2-56. Segment Register Manipulation Instructions

Name	Mnemonic	Operand Syntax
Move to Segment Register Double Word	**mtsrd**	SR,rS
Move to Segment Register	**mtsr**	SR,rS
Move to Segment Register Double Word Indirect	**mtsrdin**	rS,rB
Move to Segment Register Indirect	**mtsrin**	rS,rB
Move from Segment Register	**mfsr**	rD,SR
Move from Segment Register Indirect	**mfsrin**	rD,rB
SLB Invalidate All	**slbia**	—
SLB Invalidate Entry	**slbie**	rB

2.3.6.3.3 Translation Lookaside Buffer Management Instructions

The address translation mechanism is defined in terms of segment descriptors and page table entries (PTEs) used by PowerPC processors to locate the logical to physical address mapping for a particular access. These segment descriptors and PTEs reside in segment tables and page tables in memory, respectively.

Refer to Chapter 7, "Memory Management" for more information about TLB operation. Table 2-57 summarizes the operation of the TLB instructions in the 620.

Table 2-57. Translation Lookaside Buffer Management Instruction

Name	Mnemonic	Operand Syntax	Implementation Notes
TLB Invalidate Entry	**tlbie**	rB	Execution of this instruction causes all entries in the congruence class corresponding to the specified EA to be invalidated in the processor executing the instruction and in the other processors attached to the same bus by causing a TLB invalidate operation on the bus as described in Section 7.2.4, "Address Transfer Attribute Signals." The OEA requires that a synchronization instruction be issued to guarantee completion of a **tlbie** across all processors of a system. The 620 implements the **tlbsync** instruction which causes a TLBSYNC operation to appear on the bus as a distinct operation, different from a SYNC operation. It is this bus operation that causes synchronization of snooped **tlbie** instructions. Multiple **tlbie** instructions can be executed correctly with only one **tlbsync** instruction, following the last **tlbie**, to guarantee all previous **tlbie** instructions have been performed globally. Software must ensure that instruction fetches or memory references to the virtual pages specified by the **tlbie** have been completed prior to executing the **tlbie** instruction. When a snooping 620 detects a TLB invalidate entry operation on the bus, it accepts the operation only if no TLB invalidate entry operation is being executed by this processor and all processors on the bus accept the operation. Once accepted, the TLB invalidation is performed unless the processor is executing a multiple/string instruction, in which case the TLB invalidation is delayed until it has completed. Other than the possible TLB miss on the next instruction prefetch, the **tlbie** does not affect the instruction fetch operation—that is, the prefetch buffer is not purged and does not cause these instructions to be refetched.
TLB Synchronize	**tlbsync**	—	The TLBSYNC operation appears on the bus as a distinct operation, different from a SYNC operation. It is this bus operation that causes synchronization of snooped **tlbie** instructions. See the **tlbie** description above for information regrading using the **tlbsync** instruction with the **tlbie** instruction. For more information about how other processors react to TLB operations broadcast on the system bus of a multiprocessing system, see Chapter 8, "System Interface Operation."

2.3.7 Recommended Simplified Mnemonics

To simplify assembly language coding, a set of alternative mnemonics is provided for some frequently used operations (such as no-op, load immediate, load address, move register, and complement register). Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in this document.

For a complete list of simplified mnemonics, see Appendix F, “Simplified Mnemonics,” in *The Programming Environments Manual.*

Chapter 3
Instruction and Data Cache Operation

This chapter describes the organization of the PowerPC 620's on-chip cache system, the MESI cache coherency protocol, special concerns for cache coherency in single- and multiple-processor systems, cache control instructions, various cache operations, and the interaction between the cache and the memory unit.

To minimize the number of bus accesses, the 620 contains separate 32-Kbyte, eight-way set-associative instruction and data caches and also provides a level 2 (L2) cache interface described in Chapter 9, "Secondary Cache Interface." Systems may also be implemented with a level 3 (L3) cache external to the 620.

The 620's cache block size is 64 bytes. The cache is designed to adhere to a write-back policy, but the 620 allows control of cacheability, write policy, and memory coherency at the page and block level, as defined by the PowerPC architecture. The caches use a least recently used (LRU) replacement policy.

The 620 cache implementation has the following characteristics:

- Separate 32-Kbyte instruction and data caches (Harvard architecture)
- Instruction and data caches are eight-way set associative.
- Caches implement an LRU replacement algorithm within each set.
- The cache directories are physically addressed. The physical address tag is stored in the cache directory. (Note that physical is referred to as real in the architecture specification.)
- Both the instruction and data caches have 64-byte cache blocks. A cache block is the block of memory that a coherency state describes, also referred to as a cache line.
- The coherency state bits for each block of the data cache allow encoding for all four possible MESI states:
  - — Modified (Exclusive) (M)
  - — Exclusive (Unmodified) (E)
  - — Shared (S)
  - — Invalid (I)

- The coherency state bit for each cache block of the instruction cache allows encoding for two possible states:
  - — Invalid (INV)
  - — Valid (VAL)
- The instruction cache can be invalidated or locked, and the data cache can be invalidated by setting the appropriate bits in the hardware implementation dependent register 0 (HID0), a special-purpose register (SPR) specific to the 620.

The 620 uses 16-word burst transactions to transfer cache blocks to and from memory. When requesting burst reads, the 620 presents a quad-word–aligned address. Memory controllers are expected to transfer this quad word of data first, followed by quad words from increasing addresses, wrapping back to the beginning of the 16-word block as required.

Writes of cache blocks by the 620 (for a copy-back operation) always present the first address of the block, and transfer data beginning at the start of the block. However, this does not preclude other masters from transferring critical double words first on the bus for writes.

Note that in this chapter the terms multiprocessor and multiple-processor are used in the context of maintaining cache coherency. These devices could be processors or other devices that can access system memory, maintain their own caches, and function as bus masters requiring cache coherency.

The organization of the 620 instruction and data caches is shown in Figure 3-1.

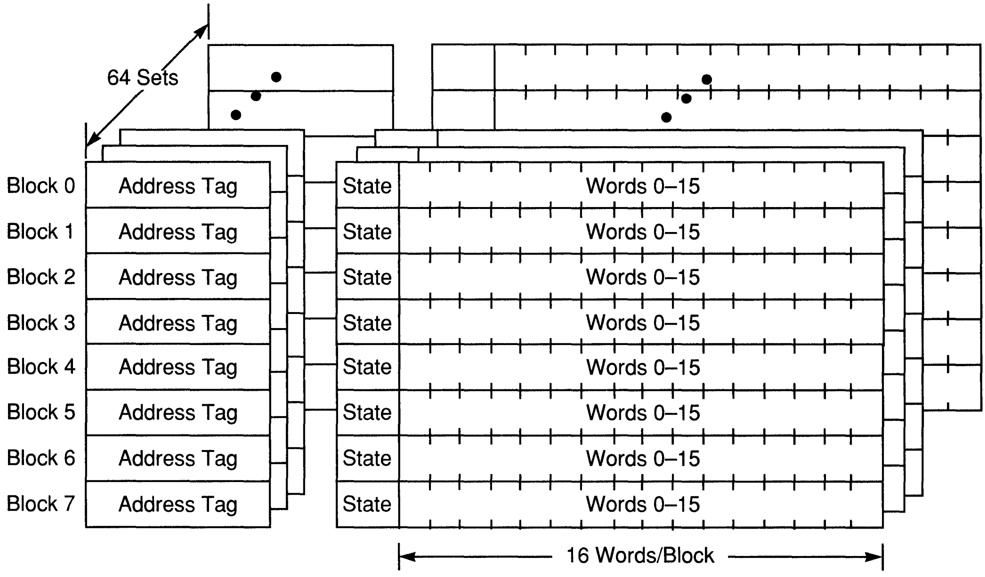


Figure 3-1. Cache Organization

The 620's instruction and data caches are connected to the bus interface unit (BIU) with a 128-bit bus. The 128-bit bus allows four instructions to be loaded into the instruction cache or a quad word (for example, two double-precision floating-point operands) to be loaded into the data cache in a single clock. The instruction cache provides a 156-bit interface (four 32-bit instructions plus 7 predecoded bits per instruction) to the instruction fetcher, so four instructions can be made available to the instruction unit in a single clock cycle.

3.1 Data Cache Organization

The 620's physically-addressed, physically indexed data cache lies between the load/store instruction unit (LSU) and the bus interface unit (BIU), and provides the ability to read and write data in memory by reducing the number of system bus transactions required for execution of load/store instructions.

The LSU transfers data between the data cache and the result bus, which routes data to the other execution units. The LSU supports the address generation and all the data alignment to and from the data cache. The LSU also handles other types of instructions that access memory, such as cache control instructions, and supports out-of-order loads and stores while ensuring the integrity of data.

The 32-Kbyte, eight-way set-associative data cache is a nonblocking write-back cache with hardware reload. The associative capability is implemented using a content addressable memory (CAM) within the cache instead of the traditional n-way bussing and n-way comparator bank external to the cache. The use of a CAM is advantageous in relation to frequency and area considerations, but can impact the effective associativity, since the cache cannot contain multiple entries which have EA[44–57] the same.

The data cache supports a coherent memory system using the four-state MESI coherency (modified/exclusive/shared/invalid) protocol. There are two separate address ports into the data cache; one port for stores and snoop operations that access the cache using a physical address, and one port for load accesses using an effective address. A load or store access may hit in the cache every cycle since there is no write recovery blockage between a store and a subsequent load. The data cache is divided into two halves, with accesses to each cache half selected with a low order address bit. Load and store operations can access the cache in parallel if they address different halves of the data cache. Snoop lookups access the cache such that parallel load operations are stalled only if they access the same cache set.

Each cache block contains 16 contiguous words from memory that are loaded from a 16-word boundary; as a result, cache blocks are aligned with page boundaries. Within a single cycle, the data cache provides a quad-word access to the LSU.

The data cache stores one parity bit for each byte of cached data. This byte parity is calculated within the data cache for any cache array write (allocation of a new cache block or execution of a store or data cache block zero instruction). Parity is checked on every read operation from the cache array (execution of a load instruction or cache control instruction that reads cached data, or copyback operations caused by cache block replacement, snoop operations, or cache control instructions). Parity errors cause the generation of a machine check exception as described in Chapter 4, "Exceptions."

3.2 Instruction Cache Organization

The 32-Kbyte, eight-way set-associative instruction cache is physically-indexed. The instruction cache also contains 7 Kbytes of predecoded instruction bits (7 bits per instruction). The organization of the instruction cache, shown in Figure 3-1, is identical to that of the data cache. Each cache block contains 16 contiguous words from memory that are loaded from an 16-word boundary; as a result, cache blocks are aligned with page boundaries.

The associative capability of the instruction cache is implemented using a content addressable memory (CAM) within the cache instead of the traditional n-way bussing and n-way comparator bank external to the cache. The use of a CAM is advantageous in relation to frequency and area considerations, but can impact the effective associativity, as the cache cannot contain multiple entries which have EA[44–57] the same.

The instruction cache implements a cache reload buffer (CRB), which stores the cache block received as a result of the last instruction cache miss. Instruction from the BIU are loaded directly into the CRB, allowing the processor to access other cache blocks in the instruction cache without waiting for the cache block to be filled. If instructions are accessed while they are still in the CRB, the CRB will supply the instructions to the fetcher. The instructions and associated tag information are loaded into the cache from the CRB when the next instruction cache miss occurs.

Within a single cycle, the instruction cache provides as many as four instructions to the instruction fetch unit. The instruction cache coherency is software-controlled. The instruction cache can be invalidated on a block or invalidate-all granularity. The instruction cache can be enabled, locked, and checked for parity depending on the setting of enable bits provided in HID0.

The instruction cache differs from the data cache in that it does not implement MESI cache coherency protocol, and a single state bit is implemented that indicates only whether a cache block is valid or invalid. If a processor modifies a memory location that may be contained in the instruction cache, software must ensure that memory updates are visible to the instruction fetching mechanism. This can be achieved by the following instruction sequence:

```
dcbst    # update memory
sync     # wait for update
icbi     # remove (invalidate) copy in instruction cache
sync     # wait for ICBI operation to be globally performed
isync    # remove copy in own instruction buffer
```

These operations are necessary because the data cache is a write-back cache. Because instruction fetching bypasses the data cache, changes made to items in the data cache may not be reflected in memory until after a fetch operation completes.

3.3 MMUs/Bus Interface Unit

The bus interface unit (BIU) implements both tenured and split-transaction modes, with an 8-bit tag provided for all address and data transactions. If permitted, the BIU can complete one or more write transactions between the address and data tenures of a read transaction. The BIU has 40-bit address and 128-bit data buses, with the address and data buses protected by word and byte parity, respectively.

The BIU implements the critical-quad-word-first access where the double word requested by the fetcher or the LSU is fetched first and the remaining words in the line are fetched later. The critical quad word as well as other words in the cache block are forwarded to the fetcher or to the LSU before they are written to the cache. When a memory access fails to hit in the cache, the 620 accesses system memory through the bus interface unit. These operations must arbitrate for bus access.

The memory management units (MMUs) provide address translation as specified by the PowerPC OEA, including block address translation and page translation of memory segments. The MMUs and the bus interface unit are shown in Figure 3-2.

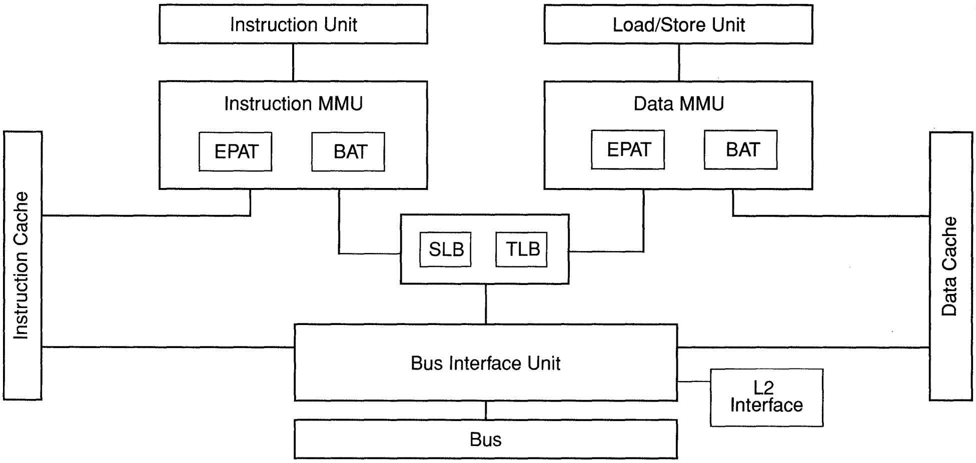


Figure 3-2. Bus Interface Unit and MMU

The 620 performs address translation in two levels. The first-level translation is accomplished by two separate 64-entry, fully-associative effective to physical translation caches (EPATs); one for instruction fetches, and the other for data accesses. The EPAT caches the effective address to physical address pairs returned from the second-level MMU. The second-level MMU consists of a 20-entry, fully-associative SLB and a 128-entry, 2-way set associative TLB. The second-level MMU is shared between the first-level instruction and data MMUs. The 620 provides hardware that performs the TLB reload (also known as page table walk) when a translation is not in a TLB. Memory management is described in Chapter 5, "Memory Management." The BIU handles block fill and write-back requests from either cache, as well as all noncacheable reads and writes.

3.4 Sequential Consistency

The following sections describe issues related to sequential consistency with respect to single processor and multiprocessor systems.

3.4.1 Sequential Consistency Within a Single Processor

The PowerPC architecture requires that all memory operations executed by a single processor be sequentially consistent with respect to that processor. This means that all memory accesses appear to be executed in the order that is specified by the program with respect to exceptions and data dependencies. Even though the 620 has multiple pipelines into the cache and the memory access through the pipeline is out of program order, the 620 achieves the sequential consistent requirement effect by maintaining a centralized store queue to check data dependencies. The completion unit ensures that all exceptions caused by memory accesses will be handled in program order. Note that although memory accesses that miss in the cache are forwarded onto the memory queue for future arbitration onto the bus, all potential synchronous exceptions have been resolved before the cache. In addition, although subsequent memory accesses can address the cache, full coherency checking between the cache and the memory queue is provided to avoid dependency conflicts.

3.4.2 Weak Consistency Between Multiple Processors

The PowerPC architecture requires only weak consistency among processors—that is, memory accesses between processors need not be sequentially consistent and memory accesses among processors can occur in any order. It is important to note that the 620 processor takes advantage of this relaxed requirement in an effort to maximize the effectiveness of the bus. The 620 will allow read operations to go ahead of store operations (except when a dependency exists). In addition, the 620 may re-order store operations unless there is an **eieio** instruction in between. A single store-multiple instruction accessing cache-inhibited memory may be converted into multiple bus operations, and 620 may re-order those operations.

Note that strong ordering of memory accesses with respect to the bus (and therefore, as observed by other processors and other bus participants) can be accomplished by following instructions that access memory with the **sync** instruction.

3.4.3 Sequential Consistency Within Multiprocessor Systems

The PowerPC architecture defines a load operation to have been performed with respect to all other processors (and mechanisms) when the value to be returned by the load can no longer be changed by a subsequent store by any processor (or other mechanism). In addition, it defines a store operation to be performed with respect to all other processors (and mechanisms) when any load operation from the same location returns the value stored (or a subsequently stored value).

In the 620, cacheable load operations and cacheable, non–write-through store operations are performed with respect to all other processors (and mechanisms) when they have arbitrated to address the cache. If a cache miss occurs, these operations may drop a memory request into the processor's memory queue, which is considered an extension to the state of the cache with respect to snooping bus operations.

However, cache-inhibited load operations and cache-inhibited or write-through store operations are performed with respect to other processors (and mechanisms) when they have been successfully presented onto the 620 bus interface. As a result, if multiple processors are performing these types of memory operations to the same addresses without properly synchronizing one another (through the use of the **lwarx/stwcx.** instructions), the results of these instructions are sensitive to the conditions associated with the order in which the processors are granted bus access.

If the 620 uses an L3 cache, the system designer must ensure the memory system responds to bus operations resulting from the execution of **sync** and **eieio** instructions in such a way that the required ordering of memory operations is preserved.

3.5 Memory and Cache Coherency

The 620 can support a fully coherent 16 terabyte (2^{40}) memory address space. Bus snooping is used to drive a four-state (MESI) cache coherency protocol which ensures the coherency of all processor and direct-memory access (DMA) transactions to and from global memory with respect to each processor's cache. It is important that all bus participants employ similar snooping and coherency control mechanisms. The coherency of memory is maintained at a granularity of 64-byte cache blocks (this size is also called the coherency or cache-block size).

All instruction and data accesses are performed under the control of the four memory/cache access attributes:

- Write-through (W attribute)
- Caching-inhibited (I attribute)
- Memory coherency (M attribute)
- Guarded (G attribute)

These attributes are programmed by the operating system for each page and block. The W and I attributes control how the processor performing an access uses its own cache. The M attribute ensures that coherency is maintained for all copies of the addressed memory location. The G attribute prevents speculative loading and prefetching from the addressed memory location.

3.5.1 Data Cache Coherency Protocol

The primary objective of a coherent memory system is to provide the same image of memory to all processors in the system. This is an important feature of multiprocessor systems since it allows for synchronization, task migration, and the cooperative use of shared resources. An incoherent memory system could easily produce unreliable results depending on when and which processor executed a task. For example, when a processor performs a store operation, it is important that the processor have exclusive access to the addressed block before the update is made. If not, another processor could have a copy of the old (or stale) data. Two processors reading from the same memory location would get different answers.

To maintain a coherent memory system, each processor must follow simple rules for managing the state of the cache. These include externally broadcasting the intention to read a cache block not in the cache and externally broadcasting the intention to write into a block that is not owned exclusively. Other processors respond to these broadcasts by snooping their caches and reporting status back to the originating processor. The status returned includes a shared indicator (that is, another processor has a copy of the addressed block) and a retry indicator (that is, another processor either has a modified copy of the addressed block that it needs to push out of the device, or another processor had a queuing problem that prevented appropriate snooping from occurring).

To maximize performance, the 620 provides a second path into the data cache directory for snooping. This allows the mainstream instruction processing to operate concurrently with the snooping operation. The instruction processing is affected only when the snoop control logic detects a situation where a snoop push of modified data is required to maintain memory coherency.

Each 64-byte cache block in the 620 data cache is in one of four states. The four possible states for a block in the cache are the invalid state (I), the shared state (S), the exclusive state (E), and the modified state (M). In a system where multiple cache levels function together to form a single cache the inclusivity of a cache level above in a cache level below is called vertical cache coherence. Coherency between different processors caches is called horizontal cache coherence, and is maintained by all cache devices snooping the bus level below, and optionally snooping the bus level above. The tables in the following sections illustrate the rules concerning vertical and horizontal cache state coherence. The vertical lines in the tables represent the separation between multi-level processor caches, and the horizontal lines represent the separation between cache levels, usually implemented by a bus interface. The box formed by these horizontal and vertical lines contain the allowable cache states for that cache level.

Note that in the description of the MESI states that follow the term exclusive indicates that the cache block referenced is located in a given cache, and in no other cache at the same level. The term modified indicates that the cache block referenced is modified with respect to main memory.

3.5.1.1 Modified Cache State

The modified (M) cache state specifies that a cache block is valid, modified, and exclusive. The cache levels above a cache block marked M may be marked I, M, or S. Cache levels below a block marked M may only be marked M, and other caches at the same level must be marked I. Table 3-1 shows the permissible M cache states in a multi-level, multi-processor cache implementation.

Table 3-1. Cache Level and Modified Cache State

Cache Level	Processor A	Processor B
Level 1 cache	MSI	
Level 2 cache	M	I
Level 3 cache	M	

3.5.1.2 Exclusive Cache State

The exclusive (E) cache state specifies that a cache block is valid, not modified, and exclusive to the cache block. The cache levels above a block marked E may be marked S or I, but not M or E. Cache blocks marked E must be at the lowest cache level, and other cache blocks at the same level must be marked I. Table 3-2 shows the permissible E cache states in a multi-level, multi-processor cache implementation.

Table 3-2. Cache Level and Exclusive Cache State

Cache Level	Processor A	Processor B
Level 1 cache	SI	
Level 2 cache	E	I

3.5.1.3 Shared Cache State

The shared (S) cache state specifies that a cache block is valid, and shared with another cache block. Cache levels above a block marked S may be marked S or I, but not M or E. Cache levels below a block marked S may be marked M, E, or S, but not I. Caches in other processors may mark the cache block S or I, but not M or E. Table 3-3 shows the permissible S cache states in a multi-level, multi-processor cache implementation.

Table 3-3. Cache Level and Shared Cache State

Cache Level	Processor A	Processor B
Level 1 cache	SI	
Level 2 cache	S	SI
Level 3 cache	MES	

3.5.1.4 Invalid Cache State

The invalid (I) cache state specifies that there is not a valid copy of the cache block in the cache. The cache levels above a cache block marked I must be marked I, and cache levels below a cache block marked I may be marked M, E, S or I. Caches in other processor may mark the cache block M, E, S, or I. Table 3-4 shows the permissible I cache states in a multi-level, multi-processor cache implementation.

Table 3-4. Cache Level and Invalid Cache State

Cache Level	Processor A	Processor B
Level 1 cache	I	
Level 2 cache	I	MESI
Level 3 cache	MESI	

3.5.2 Coherency and Secondary Caches

The 620 provides an interface to support a larger, off-chip secondary cache. The use of an L2 cache can serve to further improve performance by reducing the number of bus accesses. The L2 cache must operate with respect to the memory system in a manner that is consistent with the intent of the PowerPC architecture.

External L3 caches must forward all relevant system bus traffic onto the 620 so the 620 can take the appropriate actions to maintain memory coherency as defined by the PowerPC architecture.

3.5.3 Page Table Control Bits

The PowerPC architecture allows certain memory characteristics to be set on a page and on a block basis. These characteristics include the following:

- Write-back/write-through (using the W bit)
- Cacheable/noncacheable (using the I bit)
- Memory coherency enforced/not enforced (using the M bit)

An additional page control bit, G, handles guarded memory and is not considered here. This ability allows both single- and multiple-processor system designs to exploit numerous system-level performance optimizations.

The PowerPC architecture defines two of the possible eight decodings of these bits to be unsupported (WIM = 110 or 111).

Note that software must exercise care with respect to the use of these bits if coherent memory support is desired. Careless specification of these bits may create situations that present coherency paradoxes to the processor. In particular, this can happen when the state of these bits is changed without appropriate precautions (such as flushing the pages that correspond to the changed bits from the caches of all processors in the system) or when the address translations of aliased physical addresses specify different values for any of the WIM bits. These coherency paradoxes can occur within a single processor or across several processors.

It is important to note that in the presence of a paradox, the operating system software is responsible for correctness. The next section provides a few simple examples to convey the meaning of a paradox.

3.5.4 MESI State Diagram

The 620 provides dedicated hardware to provide data cache coherency by snooping bus transactions. The address retry capability of the 620 enforces the MESI protocol, as shown in Figure 3-3. Figure 3-3 assumes that the WIM bits are set to 001; that is, write-back, caching-not-inhibited, and memory coherency enforced.

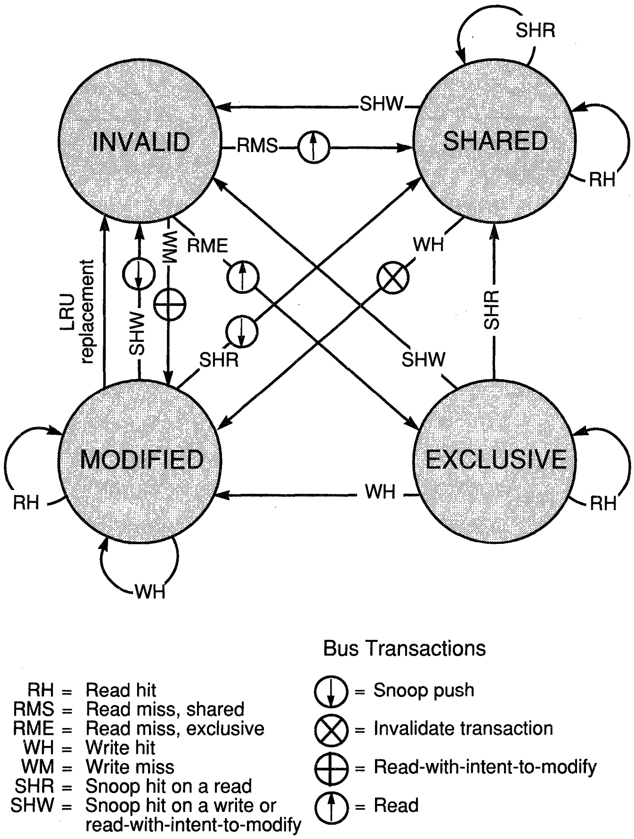


Figure 3-3. MESI Cache Coherency Protocol—State Diagram (WIM = 001)

3.5.5 Coherency Paradoxes in Single-Processor Systems

The following coherency paradoxes can be encountered within a single processor:

- Load or store operations to a page with WIM = 0b011 and a cache hit occurs. Caching was supposed to be inhibited for this page. Any load operation to a cache-inhibited page that hits in the cache presents a paradox to the processor. The 620 ignores the data in the cache and the state of the cache block is unchanged.
- Store operation to a page with WIM = 0b10X and a cache hit on a modified cache block occurs. This page was marked as write-through yet the processor was given access to the cache (write-through page are always main memory). Any store operation to a write-through page that hits a modified cache block in the cache

presents a coherency paradox to the processor. The 620 writes the data both to the cache and to main memory (note that only the data for this store is written to main memory and not the entire cache block). The state of the cache block is unchanged.

3.5.6 Coherency Paradoxes in Multiple-Processor Systems

It is possible to create a coherency paradox across multiple processors. Such paradoxes are particularly difficult to handle since some scenarios could result in the purging of modified data, and others may lead to unforeseen bus deadlocks.

Most of these paradoxes center around the interprocessor coherency of the memory coherency bit (or the M bit). Improper use of this bit can lead to multiple processors accepting a cache block into their caches and marking the data as exclusive. In turn, this can lead to a state where the same cache block is modified in multiple processor caches. Noncoherent cache states occur in the L1/L2 cache state pairings E/M, M/E, E/E, M/S, E/S, M/I, E/I, and S/I. The 620 does not detect these cache states and processor behavior is undefined if they occur.

3.6 Cache Configuration

There are several bits in the HID0 register that can be used to configure the instruction and data cache. These are described as follows:

- Bit 1—Enable cache parity checking. Enables a machine check exception based on the detection of a cache parity error. If this bit is cleared, cache parity errors are ignored. Note that the machine check exception is further affected by the MSR[ME] bit, which specifies whether the processor enters checkstop state or continues processing
- Bit 16—Instruction cache enable. If this bit is cleared, the instruction cache is neither accessed nor updated. Disabling the caches forces all pages to be accessed as if they were marked cache-inhibited (WIM = X1X). All potential cache accesses from the bus are ignored.
- Bit 17—Data cache enable. If this bit is cleared, the data cache is neither accessed nor updated. Disabling the cache forces all pages to be accessed as if they were marked cache-inhibited (WIM = X1X). All potential cache accesses from the bus, such as snoop and cache operations are ignored.
- Bit 18—Instruction cache lock. Setting this bit locks the instruction cache, in which case all cache misses are treated as cache-inhibited. Cache hits occur as normal. Cache operations and the **icbi** instruction continue to work as normal.
- Bit 20—Instruction cache invalidate all. When this bit is set, the instruction cache begins an invalidate operation marking the state of each cache block in the desired cache as invalid without copying back any data to memory. It is assumed that no data in the instruction cache is modified. Invalidation of the instruction cache is

completed in two processor clock cycles, and access to the cache is blocked during this time. The bits are reset when the invalidation operation begins (usually the cycle immediately following the write to the register beginning an invalidate operation).

- Bit 21—Data cache invalidate all. When this bit is set, the data cache begins an invalidate operation marking the state of each cache block in the desired cache as invalid without copying back any modified lines to memory. Access to the cache is blocked during this time. The bits are reset when the invalidation operation begins (usually the cycle immediately following the write to the register). Any accesses to the cache from the bus are signaled as a miss during the time that the invalidate-all operation is in progress.
- Bits 27–28—Instruction fetch modes (with address translation enabled).
  — 00: no speculative fetch from main memory
  — 01: no speculative fetch from main memory with more than one pending branch
  — 10: no speculative fetch from main memory with more than two pending branches
  — 11: allow speculative instruction fetching from main memory

The HID0 register can be accessed with the **mtspr** and **mfspr** instructions.

3.7 Cache Management Instructions

The VEA and OEA portions of the PowerPC architecture define instructions that can be used for controlling caches in both single- and multiprocessor systems. The exact behavior of these instruction in the 620 is described in the following sections.

Several of these instructions are required to broadcast their essence (such as a Kill, Clean, or Flush operation) onto the 620 bus interface so that all processors in a multiprocessor system can take the appropriate actions. The 620 contains snooping logic to monitor the bus for these commands and control logic to keep the cache and the memory queue coherent. Additional details on the specific bus operations can be found in Chapter 8, "System Interface Operation."

3.7.1 Instruction Cache Block Invalidate (icbi)

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. If the addressed block is in the instruction cache, the 620 marks this instruction cache block as invalid. This instruction changes neither the content nor status of the data cache. The ICBI operation is broadcast on the 620 bus unconditionally to support this function throughout a system's memory hierarchy.

3.7.2 Instruction Synchronize (isync)

The **isync** instruction causes the 620 to purge its instruction buffers and fetch the next sequential instruction.

3.7.3 Data Cache Block Touch (dcbt) and Data Cache Block Touch for Store (dcbtst)

The Data Cache Block Touch (**dcbt**) and Data Cache Block Touch for Store (**dcbtst**) instructions provide potential system performance improvement through the use of software-initiated prefetch hints. The 620 treats these instructions identically. Note that PowerPC implementations are not required to take any action based on the execution of this instruction, but they may choose to prefetch the cache block corresponding to the effective address into their cache. The 620 fetches the data into the cache when the address hits in the TLB or the BAT, is permitted load access from the addressed page, is not directed to a direct-store segment, and is directed at a cacheable page. Otherwise, the 620 treats these instructions as no-ops.

Regarding MESI cache coherency, the data brought into the cache as a result of these instructions is validated in the same manner that a load instruction would be (that is, if no other bus participant has a copy, it is marked as exclusive; otherwise it is marked as shared). The memory reference of a **dcbt** instruction causes the reference bit to be set.

Note also that the successful execution of the **dcbt** instruction affects the state of the TLB and cache LRU bits as defined by the LRU algorithm.

3.7.4 Data Cache Block Set to Zero (dcbz)

As defined in the VEA, when the **dcbz** instruction is executed the effective address is computed, translated, and checked for protection violations. If the 620 does not already have exclusive access to this cache block, it presents a Kill operation onto the 620 bus—a Kill operation instructs all other processors to invalidate copies of the cache block that may reside in their caches. After it has exclusive access to the cache block, the 620 writes all zeros into the cache block. In the event that the 620 already has exclusive access, it immediately writes all zeros into the cache block. If the addressed block is within a noncacheable or a write-through page, or if the cache is locked or disabled, an alignment exception occurs.

3.7.5 Data Cache Block Store (dcbst)

As defined in the VEA, when a Data Cache Block Store (**dcbst**) instruction is executed, the effective address is computed, translated, and checked for protection violations. If the 620 does not have modified data in this block, the 620 broadcasts a Clean operation onto the bus. If modified (dirty) data is associated with the cache block, the processor pushes the modified data out of the cache and into the memory queue for future arbitration onto the 620 bus. In this situation, the cache block is marked as exclusive. Otherwise this instruction is treated as a no-op.

3.7.6 Data Cache Block Flush (dcbf)

As defined in the VEA, when a Data Cache Block Flush (**dcbf**) instruction is executed, the effective address is computed, translated, and checked for protection violations. If the 620 does not have modified data in this cache block, it broadcasts a Flush operation onto the 620 bus. If the addressed cache block is in the cache, the 620 marks this data as invalid. However, if the cache block is present and modified, the processor pushes the modified data into the memory queue for arbitration onto the 620 bus and the cache block is marked as invalid.

3.7.7 Data Cache Block Invalidate (dcbi)

As defined in the OEA, when a Data Cache Block Invalidate (**dcbi**) instruction is executed, the effective address is computed, translated, and checked for protection violations.

The 620 broadcasts a Kill operation onto the 620 bus. If the addressed cache block is in the cache, the 620 marks this data as invalid regardless of whether the data is modified. Because this instruction may effectively destroy modified data, it is privileged and has store semantics with respect to protection; that is, write permission is required for the DCBI (Kill) operation.

3.8 Basic Cache Operations

This section describes operations that can occur to the cache, and how these operations are implemented in the 620.

3.8.1 Cache Reloads

A cache block is reloaded after a read miss occurs in the cache. The cache block that contains the address is updated by a burst transfer of the data from system memory. Note that if a read miss occurs in a multiprocessor system, and the data is modified in another cache, the modified data is first written to external memory before the cache reload occurs.

3.8.2 Cache Cast-Out Operation

The 620 uses an LRU replacement algorithm to determine which of the eight possible cache locations should be used for a cache update. Updating a cache block causes any modified data associated with the least-recently used element to be written back, or cast out, to system memory.

3.8.3 Cache Block Push Operation

When a cache block in the 620 is snooped and hit by another processor and the data is modified, the cache block must be written to memory and made available to the snooping device. The cache block that is hit is said to be pushed out onto the bus.

3.8.4 Atomic Memory References

The **lwarx/stwcx.** instruction combination can be used to emulate atomic memory references. These instructions are described in Chapter 2, “Programming Model.”

3.9 Cache State Response to Instruction Execution and Bus Operations

The following sections define the cache state transitions for the 620 caches with respect to processor instructions and bus operations. External L3 cache state transitions due to snoop responses to transactions on the bus above are also described to assist in the design of systems that implement an external L3 cache.

3.9.1 Cache State Transitions Due to Instruction Execution

Table 3-5 below describes the cache coherency state transitions, bus operations, and ARESPIN signal states associated with cache load, store, and control instructions. Each of the columns in Table 3-5 are defined as follows:

- Instruction or operation—Lists the instruction or operation that causes the state transition. All entries in this column are instructions except deallocate, which indicates a cache block deallocation due to a cache block castout. Note that LD and ST explicitly denote all load and store operations except for LARX (lwarx and ldarx instructions) and STCX (stwcx. and stdcx. instructions). Also, the setting of the PTE M bit is ignored for all instructions except the DCBTST operation.
- WIM state—Refers to the WIM memory access mode bits in the PTE. Refer to Chapter 5, "Memory Management" for additional information about supported WIMG bit configurations.
- Coherency state—Defines the coherency for the addressed cache block. The notation "->" indicates that the states shown on the left will transition to the states shown on the right following the execution of the instruction. If the "->" notation is not shown, the state does not change following instruction execution. The cache is not accessed during instruction execution if the coherency state entry is blank.
- Bus operation—Defines the bus operation, if any, that occurs as a result of instruction execution.
- ARESPIN signal state—Reflects the input to the 620 from an external arbiter that combines the responses from the ARESPOUT signals of other bus masters. The ARESPIN responses are defined as follows:
  - — S, Null: If ARESPIN is S, coherency state is S, if ARESPIN is Null, coherency state is E.
  - — M, $\overline{\text{M}}$: Indicates whether response is modified or retry.
  - — Blank entry: Indicates that snoop response for this case is "don't care".

Table 3-5. Cache State Transitions Due to Instruction Execution

Instruction or Operation	WIM State	Coherency State	Bus Operation	ARESPIN	Comments
LD, DCBT	Cacheable	MES			
LARX		MES	LARX-reserve		LARX-reserve broadcast if L3 cache enabled
LD, DCBT, LARX, DCBTST (M = 0)		I -> S	Burst read	M	M indicates data intervention
		I -> SE	Burst read	S, Null	
LD, LARX	Noncacheable		Single-beat read		
DCBT					No operation on noncacheable blocks
ST, DCBTST, STCX	Cacheable, writeback	M			
ST, STCX		E -> M			
DCBTST		E			
DCBTST M=0		S			
ST, STCX, and DCBTST M=1		S -> M	DClaim		
		I -> M	RWITM	M, $\overline{M}$	M indicates data intervention
ST	Cacheable, write through	MESI	Write with Flush		STCX not supported for cacheable, write through
ST,STCX	Noncacheable		Write with Flush		
DCBTST	Cacheable, write through, and noncacheable				No operation on write through or noncacheable blocks
Deallocate	Cacheable	M -> I	Write with Kill (CB)		
		ESI -> I			
DCBF		M -> I	Write with Kill (F)		
		ESI -> I	Flush		
DCBI		MESI -> I	DKill		
DCBST		M -> SE	Write with Clean		E if no L3
		SE -> SE	Clean	S, Null	Clean optional for ES if no L3
		I	Clean		

Table 3-5. Cache State Transitions Due to Instruction Execution

Instruction or Operation	WIM State	Coherency State	Bus Operation	ARESPIN	Comments
DCBZ	Cacheable, writeback	EM -> M			
	Cacheable, writeback	IS -> M	DClaim		
	Cacheable, write through, and noncacheable				Alignment exception

Chapter 3, "Addressing Modes and Instruction Set Summary," and Chapter 8, "Instruction Set," in *The Programming Environments Manual* describe the cache control instructions in detail. Several of the cache control instructions broadcast onto the 620 interface so that all processors in a multiprocessor system can take appropriate actions. The 620 contains snooping logic to monitor the bus for these commands and the control logic required to keep the cache and the memory queues coherent. For additional details about the specific bus operations performed by the 620, see Chapter 8, "System Interface Operation."

3.9.2 Cache State Transitions Due to Bus Snoop Operations

Table 3-6 describes the state transitions that occur due to bus snoop operations. Note that the following conditions apply when the 620 is snooping bus operations:

- Bus operations that are marked not memory coherent (M = 0) are not snooped by the 620
- Bus operations that are marked memory coherent (M = 1) are snooped by the 620 regardless of the state of the cache-inhibited (I) bit.
- A Write-with-Clean bus operation will always be marked not memory coherent (M = 0) and will be ignored by the 620 snooper.
- The 620 will always have at least one cache block store buffer reserved for a cache block push.

Table 3-6. Cache State Transitions Due to Bus Snoop Operations

Bus Operation		Snooper State	Reservation State	ARESPOUT	ARESPIN	Comments
Read-Burst	N=1, S=0 N=1, S=1, $\overline{\text{l2en}}$ N=1, S=1, l2en, l3en	M -> S		M		Causes C -> MC data-only operation (Intervention)
	N=1, S=1, l2en, $\overline{\text{l3en}}$	M -> E		M		
	N=0, S=0 N=0, S=1, $\overline{\text{l2en}}$ N=0, S=1, l2en, l3en	M -> S		Retry		Causes Write with Clean (push)
	N=0, S=1, l2en, $\overline{\text{l3en}}$	M -> E		Retry		
		S		S	Note3	
	S=0	E -> S		S	Note3	
	S=1			S	ReRun	
	S=1	E		S	$\overline{\text{ReRun}}$	
		I	R=0			
			R=1	S		
RWITM	N=1	M -> I		M		Causes C->C data-only operation (Intervention)
	N=0	M -> I		Retry		Causes Write with Kill (push)
		ESI -> I			$\overline{\text{ReRun}}$	
		E -> S			ReRun	
		IS				
Write-With-Kill, DKill, DClaim		MESI -> I			$\overline{\text{ReRun}}$	
		E -> S			ReRun	
		MSI				
Write-With-Flush		M -> I		Retry		Causes Write with Kill (push)
		ESI -> I			$\overline{\text{ReRun}}$	
		E -> S			ReRun	
		SI				

Table 3-6. Cache State Transitions Due to Bus Snoop Operations (Continued)

Bus Operation	Snooper State	Reservation State	ARESPOUT	ARESPIN	Comments
Read-Non-Burst	M -> S		Retry		Causes Write with Clean (push)
	ES -> S		S	Note3	
	I	R=0			
		R=1	S		
Clean	M -> ES		M[1]		Causes Write with Clean (push) E if lowest cache level, S otherwise
	S		S		
	E		S[2]	$\overline{\text{ReRun}}$	
	E -> S		S[2]	ReRun	
	I	R=0			
		R=1	S		
Flush	M -> I		M[1]		Causes Write with Kill (push)
	ESI -> I			$\overline{\text{ReRun}}$	
	E -> S			ReRun	
	SI				
SYNC, TLBSYNC			ReRun, Null		Will ReRun until done. Will Null when done.

Notes:

1. M overrides ReRun to optimize performance.
2. ARESPOUT = Shared is not significant to any other snooping device that has this block marked invalid. Note that the L2 E state implies that there is no L3 cache.
3. Burst read and single-beat read operations will mark the block S for the ReRun response, in addition to the Null and Shared responses.

3.9.3 L3 Cache State Transitions Due to Bus-Above Operations

Although the 620 does not implement an L3 cache, this information in this section is provided to show how coherence is maintained between the 620 and a cache external to the 620. For additional information about the operation of the bus and external cache refer to Chapter 8, “System Interface Operation.” Note that all bus operations occurring above the L3 cache are snooped regardless of the state of the A, W, I, M, and N address attribute bits. The L3 cache state transition attributes described in Table 3-7 are as follows:

- Bus above operation—Lists the instruction or operation that causes the state transition.
- M in L2—Indicates when the state of a cache block at a higher level has been modified.
- Snooper state—State of the L3 cache following a snoop operation.
- ARESPOUT—State of the ARESPOUT signal during snoop operation.
- ARESPIN—State of the ARESPIN signal during snoop operation.
- Comments—Describes actions taken on busses above and below the L3 cache.
  - — Bus above—Describes actions taken with respect to the bus above the L3 cache
  - — Bus below—Describes actions taken with respect to the bus below the L3 cache
  - — Address above—Indicates that the L3 decodes the address to be above the L3 cache
  - — Address below—Indicates that the L3 decodes the address to be below the L3 cache

The L3 state transitions that occur due to bus-above transactions are described in Table 3-7.

Table 3-7. L3 Cache State Transitions Due to Bus-Above Operations

Bus-Above Operation	M State in L2	L3 State Following Snoop	ARESPOUT State	ARESPIN State	Comments
Read Burst, RWITM, Write-with-Flush, Read-Non-Burst, Clean, Flush	Y	M			No Action
Read Burst	N	M	M		Bus Above: Source Data-Only Operation
		ES	S		**Address Above:** No Action **Address Below:** Bus Above: Data-Only Operation
		I -> ES	S		**Address Above:** 1. L3 allocates cache block 2. Bus Above: Sink Data-Only Operation **Address Below:** 1. Bus Below: Read-Burst, E-state if lowest cache level and $\overline{S}$ response, else S-state. 2. Bus Above: Source Data-Only Operation

Table 3-7. L3 Cache State Transitions Due to Bus-Above Operations (Continued)

Bus-Above Operation	M State in L2	L3 State Following Snoop	ARESPOUT State	ARESPIN State	Comments
RWITM	N -> Y	M	M Null		**Address Above:** Data is sourced by the L3 with the M response. **Address Below:** Bus Above: Source Data-Only Operation with Null response.
	N -> Y	E -> M	Null		**Address Above:** No Action and Null response because data is sourced from memory. **Address Below:** Bus Above: Source Data-Only Operation with Null response.
	N -> Y	S -> M	Null ,M		**Address Above:** 1a. No Action and Null response if data is sourced from memory. 1b. Modified response if data is sourced by L3. **Address Below:** 1. Bus Below: DClaim 2. Bus Above: Source Data-Only Operation with Null response.
	N -> Y	I -> M			**Address Above:** 1. L3 allocates cache block 2. Bus Above: Sink Data-Only Operation **Address Below:** 1. Bus Below: RWITM 2. Bus Above: Source Data-Only Operation with Null response.
Write-with-Kill, DKill	-> N	MESI -> I			**Address Above:** No Action **Address Below:** Bus Below: Write-with-Kill or DKill
Write-with-Clean	Y -> N	M -> ES			**Address Above:** No Action, E if lowest cache level. **Address Below:** Bus Below: Write-with-Clean, E if lowest cache level.

Table 3-7. L3 Cache State Transitions Due to Bus-Above Operations (Continued)

Bus-Above Operation	M State in L2	L3 State Following Snoop	ARESPOUT State	ARESPIN State	Comments
DClaim	N -> Y	MESI -> M			MInL2 must start as N Bus Below: DClaim if initial state is $\overline{M}$
Write-with-Flush, Read-Non-Burst	N	M -> I	Retry		**Address Above:** 1. Bus Above: Write with Kill (Push) 2. Bus Below: DKill if lower cache level exists. **Address Below:** Bus Below: Write with Kill (Push)
		ESI -> I			**Address Above:** No Action **Address Below:** Bus Below: Write-with-Flush or Read-Non-Burst
Clean	N	M -> ES	S		**Address Above:** 1. Bus Above: Write with Clean, E-state if lowest cache level, else S-state. 2. Bus Below: Clean **Address Below:** 1. Bus Below: Write with Clean
		ES	S		Bus Below: Clean
		I			Bus Below: Clean
Flush	N	M -> I			**Address Above:** 1. Bus Above: Write with Kill (Push) 2. Bus Below: DKill if lower cache level exists. **Address Below:** Bus Below: Write with Kill (Push)
		E -> I			No Action
		SI -> I			Bus Below: Flush

3.10 Access to Direct-Store Segments

The 620 supports both memory-mapped and I/O-mapped access to I/O devices. In addition to the high-performance bus protocol for memory-mapped I/O accesses, the 620 provides the ability to map memory areas to the direct-store interface (SR[T] = 1) with the following two kinds of operations:

- Direct-store operations—These operations are considered to address the noncoherent and noncacheable direct-store; therefore, the 620 does not maintain coherency for these operations, and the cache is bypassed completely.
- Memory-forced direct-store operations—These operations are considered to address memory space and are therefore subject to the same coherency control as memory accesses. These operations are global memory references within the 620 and are considered to be noncacheable.

Cache behavior (write-back, cache-inhibition, and enforcement of MESI coherency) for these operations is determined by the settings of the WIM bits.

Chapter 4
Exceptions

The OEA portion of the PowerPC architecture defines the mechanism by which PowerPC processors implement exceptions (referred to as interrupts in the architecture specification). Exception conditions may be defined at other levels of the architecture. For example, the UISA defines conditions that may cause floating-point exceptions; the OEA defines the mechanism by which the exception is taken.

PowerPC exception mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (exception vector) predetermined for each exception. Processing of exceptions begins in supervisor mode.

Although multiple exception conditions can map to a single exception vector, a more specific condition may be determined by examining a register associated with the exception—for example, the DSISR and the floating-point status and control register (FPSCR). Additionally, certain exception conditions can be explicitly enabled or disabled by software.

The PowerPC architecture requires that exceptions be taken in program order; therefore, although a particular implementation may recognize exception conditions out of order, they are handled strictly in order with respect to the instruction stream. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the exception is taken. For example, if a single instruction encounters multiple exception conditions, those exceptions are taken and handled sequentially. Likewise, exceptions that are asynchronous and precise are recognized when they occur, but are not handled until all instructions currently in the execute stage successfully complete execution and report their results.

Note that exceptions can occur while an exception handler routine is executing, and multiple exceptions can become nested. It is up to the exception handler to save the states if it is desired to allow control to ultimately return to the excepting program.

In many cases, after the exception handler handles an exception, there is an attempt to execute the instruction that caused the exception. Instruction execution continues until the next exception condition is encountered. This method of recognizing and handling exception conditions sequentially guarantees that the machine state is recoverable and processing can resume without losing instruction results.

To prevent the loss of state information, exception handlers must save the information stored in SRR0 and SRR1 soon after the exception is taken to prevent this information from being lost due to another exception being taken.

In this chapter, the following terminology is used to describe the various stages of exception processing:

Recognition — Exception recognition occurs when the condition that can cause an exception is identified by the processor.

Taken — An exception is said to be taken when control of instruction execution is passed to the exception handler; that is, the context is saved and the instruction at the appropriate vector offset is fetched and the exception handler routine is begun in supervisor mode (referred to as privileged state in the architecture specification).

Handling — Exception handling is performed by the software linked to the appropriate vector offset. Exception handling is begun in supervisor-mode.

Note that the PowerPC architecture documentation refers to exceptions as interrupts. In this book, the term interrupt is reserved to refer to asynchronous exceptions, and sometimes to the event that causes the exception to be taken. Also, the PowerPC architecture uses the word exception to refer to IEEE-defined floating-point exceptions, conditions that may cause a program exception to be taken (See Section 4.6.7, “Program Exception (0x00700).) The occurrence of these IEEE exceptions may in fact not cause an exception to be taken. IEEE-defined exceptions are referred to as IEEE floating-point exceptions or floating-point exceptions.

4.1 PowerPC 620 Microprocessor Exceptions

As specified by the PowerPC architecture, all exceptions can be described as either precise or imprecise and either synchronous or asynchronous. Asynchronous exceptions are caused by events external to the processor's execution; synchronous exceptions are caused by instructions.

The types of exceptions implemented by the 620 are shown in Table 4-1. Note that all exceptions except for the system management interrupt and performance monitoring exception are defined by the PowerPC architecture.

Table 4-1. Exception Classifications

Type	Exception
Asynchronous/nonmaskable	Machine Check System Reset
Asynchronous/maskable	External interrupt Decrementer interrupt System management interrupt (620-specific) Performance monitoring exception (620-specific)
Synchronous/precise	Instruction-caused exceptions

Exceptions implemented in the 620, and conditions that cause them, are listed in Table 4-2.

Table 4-2. Exceptions and Conditions—Overview

Exception Type	Vector Offset (hex)	Causing Conditions
Reserved	00000	—
System reset	00100	The causes of system reset exceptions are implementation-dependent. In the 620 a system reset is caused by the assertion of either the soft reset or hard reset signal. If the conditions that cause the exception also cause the processor state to be corrupted such that the contents of SRR0 and SRR1 are no longer valid or such that other processor resources are so corrupted that the processor cannot reliably resume execution, the copy of the RI bit copied from the MSR to SRR1 is cleared.
Machine check	00200	On the 620 a machine check exception is signaled by the assertion of the machine check input ($\overline{\text{MCP}}$) signal. If the MSR[ME] is cleared, the processor enters the checkstop state when one of these signals is asserted. Note that MSR[ME] is cleared when an exception is taken. The machine check exception is also caused by parity errors on the address or data bus or in the instruction or data caches. Regardless of the state of MSR[ME} the 620 enters the checkstop state if parity errors are detected on the address or data bus. Note that the machine check exception is imprecise with respect to the instruction that originated the bus operation. The machine check exception is disabled when MSR[ME] = 0. If a machine check exception condition exists and the ME bit is cleared, the processor goes into the checkstop state. If the conditions that cause the exception also cause the processor state to be corrupted such that the contents of SRR0 and SRR1 are no longer valid or such that other processor resources are so corrupted that the processor cannot reliably resume execution, the copy of the RI bit copied from the MSR to SRR1 is cleared.
DSI	00300	A DSI exception occurs when a data memory access cannot be performed for any of the reasons described in Section 4.6.3, "DSI Exception (0x00300)." Such accesses can be generated by load/store instructions, certain memory control instructions, and certain cache control instructions.
ISI	00400	An ISI exception occurs when an instruction fetch cannot be performed for a variety of reasons described in Section 4.6.4, "ISI Exception (0x00400)."

Table 4-2. Exceptions and Conditions—Overview (Continued)

Exception Type	Vector Offset (hex)	Causing Conditions
External interrupt	00500	An external interrupt exception occurs when the external exception signal, $\overline{\text{INT}}$, is asserted. This signal is expected to remain asserted until the exception handler begins execution. If MSR[EE] is set and the assertion of the $\overline{\text{INT}}$ signal is detected, the 620 completes the oldest instruction in the completion queue and cancels all outstanding instructions. Any exceptions associated with dispatched instructions are taken before the exception is taken.
Alignment	00600	An alignment exception may occur when the processor cannot perform a memory access for reasons described in Section 4.6.6, "Alignment Exception (0x00600)." Note that the PowerPC architecture defines a wider range of conditions that may cause an alignment exception than required in the 620. In these cases, the 620 provides logic to handle these conditions without requiring the processor to invoke the alignment exception handler.
Program	00700	A program exception is caused by one of the following exception conditions, which correspond to bit settings in SRR1 and arise during execution of an instruction: • Floating-point enabled exception—A floating-point enabled exception condition is generated when either MSR[FE0] or MSR[FE1] and FPSCR[FEX] are set. The settings of FE0 and FE1 are described in Table 4-4. FPSCR[FEX] is set by the execution of a floating-point instruction that causes an enabled exception or by the execution of a Move to FPSCR instruction that sets both an exception condition bit and its corresponding enable bit in the FPSCR. These exceptions are described in Chapter 6, "Exceptions," of *The Programming Environments Manual.* • Illegal instruction—An illegal instruction program exception is generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields or when execution of an optional instruction not provided in the specific implementation is attempted (these do not include those optional instructions that are treated as no-ops). The PowerPC instruction set is described in Section 2.3, "Instruction Set Summary." • Privileged instruction—A privileged instruction type program exception is generated when the execution of a privileged instruction is attempted and the MSR register user privilege bit, MSR[PR], is set. This exception is also generated for **mtspr** or **mfspr** with an invalid SPR field if spr[0] = 1 and MSR[PR] = 1. • Trap—A trap type program exception is generated when any of the conditions specified in a trap instruction is met. For more information, refer to Section 4.6.7, "Program Exception (0x00700)."
Floating-point unavailable	00800	A floating-point unavailable exception occurs when the floating-point available bit in the MSR is cleared (MSR[FP] = 0), and an attempt is made to execute a floating-point instruction (including floating-point load, store, or move instructions).
Decrementer	00900	The decrementer interrupt exception is taken if the exception is enabled (MSR[EE] = 1) and the exception is pending. The exception is created when the most significant bit changes of the decrementer register from 0 to 1. If it is not enabled (MSR[EE] = 0), the exception remains pending until it is taken.
Reserved	00A00	Reserved for implementation-specific exceptions; this exception is not implemented by the 620. For example, the 601 uses this vector offset for direct-store exceptions.

Table 4-2. Exceptions and Conditions—Overview (Continued)

Exception Type	Vector Offset (hex)	Causing Conditions
Reserved	00B00	—
System call	00C00	A system call exception occurs when a System Call (**sc**) instruction is executed.
Trace	00D00	The trace exception, which is implemented in the 620, is defined by the PowerPC architecture but is optional. A trace exception occurs if either MSR[SE] = 1 and any instruction (except **rfid**, **rfi**, **sc**, or trap instruction whose condition is true) successfully completed or MSR[BE] = 1 and a branch instruction is completed.
Floating-point assist	00E00	The 620 does not implement the floating-point assist exception.
Performance monitoring interrupt	00F00	The performance monitoring interrupt is a 620-specific exception and is used with the 620 performance monitor, described in Section 4.6.13, "Performance Monitoring Interrupt (0x00F00)." The performance monitoring facility can be enabled to signal an exception when the value in one of the performance monitor counter registers (PMC1 or PMC2) goes negative. The conditions that can cause this exception can be enabled or disabled through bits in the monitor mode control register 0 (MMCR0). Although the exception condition may occur when the MSR[EE] bit is cleared, the actual interrupt is masked by the MSR[EE] bit and cannot be taken until the MSR[EE] bit is set.
Reserved	01000–012FF	Reserved for implementation-specific exceptions not implemented on the 620.
Instruction address breakpoint	01300	An instruction address breakpoint exception occurs when the address (bits 0 to 61) in the IABR matches the next instruction to complete in the completion unit, the IABR enable bit (IABR[62]) is set, and the IABR break on translation bit (IABR[63]) is equal to MSR[IR].
System management interrupt	01400	A system management interrupt exception is caused when MSR[EE] = 1 and the $\overline{\text{SMI}}$ input signal is asserted. This exception is provided for use with the nap mode.
Reserved	014FF–02FFF	Reserved for implementation-specific exceptions not implemented on the 620.

4.2 Exception Recognition and Priorities

Exceptions are roughly prioritized by exception class, as follows:

1. Nonmaskable, asynchronous exceptions have priority over all other exceptions—system reset and machine check exceptions (although the machine check exception condition can be disabled so the condition causes the processor to go directly into the checkstop state). These exceptions cannot be delayed, and do not wait for the completion of any precise exception handling.
2. Synchronous, precise exceptions are caused by instructions and are taken in strict program order.
3. Imprecise exceptions (imprecise mode floating-point enabled exceptions) are caused by instructions and they are delayed until higher priority exceptions are taken.
4. Maskable asynchronous exceptions (external interrupt and decrementer exceptions) are delayed until higher priority exceptions are taken.

Exception priorities are described in "Exception Priorities," in Chapter 4, "Exceptions," in *The Programming Environments Manual.*

The following is a summary of the exception priorities for the 620, including both exceptions defined by the PowerPC architecture as well as the 620-specific exceptions.

1. System reset
2. Machine check
3. Instruction-dependent
   A) Integer loads and stores
      a) Instruction address breakpoint
      b) Alignment
      c) DSI
      d) Trace
   B) Floating-point loads and stores
      a) Instruction address breakpoint
      b) Floating-point unavailable
      c) Alignment
      d) DSI
      e) Trace

C) Other floating-point instructions
   a) Instruction address breakpoint
   b) Floating-point unavailable
   c) Program: Precise-mode floating-point enabled
   d) Trace

D) **rfi**, **rfid**, **mtmsr**, and **mtmsrd**
   a) Instruction address breakpoint for **mtmsr** and **mtmsrd** only
   b) Program: Precise-mode floating-point enabled
   c) Trace for **mtmsr** and **mtmsrd** only

E) Other instructions
   a) Instruction address breakpoint
   b) Exceptions mutually exclusive and same priority
      — Program: Trap
      — System call
      — Program: Privileged instruction
      — Program: Illegal instruction
   c) Trace

F) ISI

4. System management interrupt
   (Note that the 620 does not implement imprecise-mode floating-point exceptions)
5. External
6. Performance monitor
7. Decrementer

System reset and machine check exceptions may occur at any time and are not delayed even if an exception is being handled. As a result, state information for the interrupted exception may be lost; therefore, these exceptions are typically nonrecoverable.

All other exceptions have lower priority than system reset and machine check exceptions, and the exception may not be taken immediately when it is recognized.

If an imprecise exception is not forced by either the context or the execution synchronizing mechanism and if the instruction addressed by SRR0 did not cause the exception then that instruction appears not to have begun execution. For more information on context-synchronization, see Chapter 6, "Exceptions," in *The Programming Environments Manual.*

4.3 Support for 32-Bit Operating Systems

The 620 supports the optional 64-bit bridge as defined by the PowerPC architecture and supports the following architecture-defined, exception-related functionality:

- **mtmsr**—32-bit version of Move to Machine State Register Double Word (**mtmsrd**) instruction
- **rfi**—32-bit version of Return from Interrupt Double Word (**rfid**) instruction
- MSR[ISF]—New MSR bit that copies to MSR[SF] when an exception is taken

4.4 Exception Processing

When an exception is taken, the processor uses the save/restore registers, SRR0 and SRR1, to save the contents of the machine state register for user-level mode and to identify where instruction execution should resume after the exception is handled.

When an exception occurs, the address saved in machine status save/restore register 0 (SRR0) is used to help calculate where instruction processing should resume when the exception handler returns control to the interrupted process. Depending on the exception, this may be the address in SRR0 or at the next address in the program flow. All instructions in the program flow preceding this one will have completed execution and no subsequent instruction will have begun execution. This may be the address of the instruction that caused the exception or the next one (as in the case of a system call or trap exception). The SRR0 register is shown in Figure 4-1.

SRR0 (holds EA for instruction in interrupted program flow)	
0	63

Figure 4-1. Machine Status Save/Restore Register 0

The save/restore register 1(SRR1) is used to save machine status (selected bits from the MSR and possibly other status bits as well) on exceptions and to restore those values when **rfid** (or **rfi**) is executed. SRR1 is shown in Figure 4-2.

Exception-specific information and MSR bit values	
0	63

Figure 4-2. Machine Status Save/Restore Register 1

Typically, when an exception occurs, bits 33–36 and 42–47 of SRR1 are loaded with exception-specific information and bits 0–32, 37–41, and 48–63 of SRR1 are loaded with equivalent bits from the MSR. Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1.

Note that in other implementations every instruction fetch that occurs when MSR[IR] = 1, and every instruction execution requiring address translation when MSR[DR] = 1, may modify SRR1.

In the 620 and in other 64-bit PowerPC implementations, the MSR bits are as shown in Figure 4-3.

▨ Reserved

SF	0	ISF[1]	0 0000 ... 0000 0	POW	0	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	0	IP	IR	DR	0	PMM[2]	RI	LE
0	1	2	3 44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

[1] The ISF bit is optional and implemented only as part of the 64-bit bridge; this bit is cleared to 0 on hard reset.
[2] 620-specific

Figure 4-3. Machine State Register (MSR)—64-Bit Implementation

Table 4-3 shows the bit definitions for the MSR.

Table 4-3. MSR Bit Settings

Bit(s)	Name	Description
0	SF	Sixty-four bit mode 0 The 64-bit processor runs in 32-bit mode. Note that this is the default setting following a hard reset. 1 The 64-bit processor runs in 64-bit mode.
1	—	Reserved
2	ISF	Exception sixty-four bit mode (optional to OEA). When an exception occurs, this bit is copied into MSR[SF] to select 64- or 32-bit mode for the context established by the exception.
3–45	—	Reserved
46	—	Reserved
47	ILE	Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception.
48	EE	External interrupt enable 0 While the bit is cleared the processor delays recognition of external interrupts and decrementer exception conditions. 1 The processor is enabled to take an external interrupt or the decrementer exception.
49	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
50	FP	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions.
51	ME	Machine check enable 0 Machine check exceptions are disabled. 1 Machine check exceptions are enabled.
52	FE0	Floating-point exception mode 0.

Table 4-3. MSR Bit Settings (Continued)

Bit(s)	Name	Description
53	SE	Single-step trace enable (Optional in the architecture; implemented in the 620) 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of the next instruction. Note: If the function is not implemented, this bit is treated as reserved.
54	BE	Branch trace enable (optional) 0 The processor executes branch instructions normally. 1 The processor generates a branch trace exception after completing the execution of a branch instruction, regardless of whether the branch was taken. Note: If the function is not implemented, this bit is treated as reserved.
55	FE1	Floating-point exception mode 1.
56	—	Reserved
57	IP	Exception prefix. The setting of this bit specifies whether an exception vector offset is prepended with Fs or 0s. In the following description, *nnnnn* is the offset of the exception vector. See Table 4-2. 0 Exceptions are vectored to the physical address 0x0000_0000_000*n_nnnn* in 64-bit implementations. 1 Exceptions are vectored to the physical address 0xFFFF_FFFF_FFF*n_nnnn* in 64-bit implementations.
58	IR	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information see Chapter 5, "Memory Management."
59	DR	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. For more information see Chapter 5, "Memory Management."
60	—	Reserved
61	PMM	Performance monitor mark (620-specific). Used to mark specific processes. In conjunction with the MMCR0[3–4], FCM0, and FCM1 provides control for the processes in which the performance monitor is enabled or disabled.
62	RI	Recoverable exception (for system reset and machine check exceptions). 0 Exception is not recoverable. 1 Exception is recoverable. For more information see Section 4.6.1, "System Reset Exception (0x00100),"and Section 4.6.2, "Machine Check Exception (0x00200)."
63	LE	Little-endian mode enable 0 The processor runs in big-endian mode. 1 The processor runs in little-endian mode.

The IEEE floating-point exception mode bits (FE0 and FE1) together define whether floating-point exceptions are handled precisely, imprecisely, or whether they are taken at all. The possible settings and default conditions for the 620 are shown in Table 4-4. For further details, see Chapter 6, “Exceptions,” of *The Programming Environments Manual.*

Table 4-4. IEEE Floating-Point Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions disabled
0	1	Floating-point precise mode
1	0	Floating-point imprecise recoverable. In the 620, this bit setting causes the 620 to operate in floating-point precise mode.
1	1	Floating-point precise mode

MSR bits are guaranteed to be written to SRR1 when the first instruction of the exception handler is encountered.

4.4.1 Enabling and Disabling Exceptions

When a condition exists that may cause an exception to be generated, it must be determined whether the exception is enabled for that condition.

- IEEE floating-point enabled exceptions (a type of program exception) are ignored when both MSR[FE0] and MSR[FE1] are cleared. If either of these bits are set, all IEEE enabled floating-point exceptions are taken and cause a program exception.
- Asynchronous, maskable exceptions (that is, the external, decrementer, and system management interrupts) are enabled by setting the MSR[EE] bit. When MSR[EE] = 0, recognition of these exception conditions is delayed. MSR[EE] is cleared automatically when an exception is taken, to delay recognition of conditions causing those exceptions.
- A machine check exception can occur only if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs. Individual machine check exceptions can be enabled and disabled through bits in the HID0 register, which is described in Table 4-7.
- System reset exceptions cannot be masked.

4.4.2 Steps for Exception Processing

After it is determined that the exception can be taken (by confirming that any instruction-caused exceptions occurring earlier in the instruction stream have been handled, and by confirming that the exception is enabled for the exception condition), the processor does the following:

1. The machine status save/restore register 0 (SRR0) is loaded with an instruction address that depends on the type of exception. See the individual exception description for details about how this register is used for specific exceptions.
2. Bits 33–36 and 42–47 of SRR1 are loaded with information specific to the exception type.
3. Bits 0–32, 37–41, and 48–63 of SRR1 are loaded with a copy of the corresponding bits of the MSR. Note that depending on the implementation, reserved bits may not be copied.
4. The MSR is set as described in Table 4-3. The new values take effect beginning with the fetching of the first instruction of the exception-handler routine located at the exception vector address.

   Note that MSR[IR] and MSR[DR] are cleared for all exception types; therefore, address translation is disabled for both instruction fetches and data accesses beginning with the first instruction of the exception-handler routine.
5. Instruction fetch and execution resumes, using the new MSR value, at a location specific to the exception type. The location is determined by adding the exception's vector (see Table 4-2) to the base address determined by MSR[IP]. If IP is cleared, exceptions are vectored to the physical address 0x0000_0000_000*n*_*nnnn*. If IP is set, exceptions are vectored to the physical address 0xFFFF_FFFF_FFF*n*_*nnnn*. For a machine check exception that occurs when MSR[ME] = 0 (machine check exceptions are disabled), the checkstop state is entered (the machine stops executing instructions). See Section 4.6.2, "Machine Check Exception (0x00200)."

4.4.3 Setting MSR[RI]

The operating system should handle MSR[RI] as follows:

- In the machine check and system reset exceptions—If MSR1[RI] is cleared, the exception is not recoverable. If it is set, the exception is recoverable with respect to the processor.
- In each exception handler—When enough state information has been saved that a machine check or system reset exception can reconstruct the previous state, set MSR[RI].
- In each exception handler—Clear MSR[RI], set the SRR0 and SRR1 registers appropriately, and then execute **rfid** (or **rfi**).
- Not that the RI bit being set indicates that, with respect to the processor, enough processor state data is valid for the processor to continue, but it does not guarantee that the interrupted process can resume.

4.4.4 Returning from an Exception Handler

The Return from Interrupt instructions, **rfid** (or **rfi**), perform context synchronization by allowing previously issued instructions to complete before returning to the interrupted process. In general, execution of the **rfid** (or **rfi**) instruction ensures the following:

- All previous instructions have completed to a point where they can no longer cause an exception. If a previous instruction causes a direct-store interface error exception, the results must be determined before this instruction is executed.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The **rfid** (or **rfi**) instruction copies SRR1 bits back into the MSR.
- The instructions following this instruction execute in the context established by this instruction.

For a complete description of context synchronization, refer to Chapter 6, "Exceptions," of *The Programming Environments Manual.*

4.5 Process Switching

The operating system should execute one of the following when processes are switched:

- The **sync** instruction, which orders the effects of instruction execution. All instructions previously initiated appear to have completed before the **sync** instruction completes, and no subsequent instructions appear to be initiated until the **sync** instruction completes. For an example showing use of the **sync** instruction, see Chapter 2, "PowerPC Register Set," of *The Programming Environments Manual.*
- The **isync** instruction, which waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, protection, etc.) established by the previous instructions.
- The **stwcx.** instruction, to clear any outstanding reservations, which ensures that an **lwarx** instruction in the old process is not paired with an **stwcx.** instruction in the new process.

The operating system should set the MSR[RI] bit as described in Section 4.4.3, "Setting MSR[RI]."

4.6 Exception Definitions

Table 4-5 shows all the types of exceptions that can occur with the 620 and the MSR bit settings when the processor transitions to supervisor mode due to an exception. Depending on the exception, certain of these bits are stored in SRR1 when an exception is taken.

Table 4-5. MSR Setting Due to Exception

Exception Type	MSR Bit																	
	SF	ISF	POW	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	IP	IR	DR	PMM	RI	LE
System reset	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Machine check	ISF	—	0	—	0	0	0	0	0	0	0	0	—	0	0	0	0	ILE
DSI	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
ISI	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
External	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Alignment	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Program	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Floating-point unavailable	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Decrementer	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
System call	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Trace exception	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
System management	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Performance monitor	ISF	—	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE

0	Bit is cleared.
ILE	Bit is copied from the ILE bit in the MSR.
—	Bit is not altered

Reserved bits are read as if written as 0.

The setting of the exception prefix bit (IP) determines how exceptions are vectored. If the bit is cleared, exceptions are vectored to the physical address 0x0000_0000_000*n_nnnn* (where *nnnnn* is the vector offset); if IP is set, exceptions are vectored to the physical address 0xFFFF_FFFF_FFF*n_nnnn*. Table 4-2 shows the exception vector offset of the first instruction of the exception handler routine for each exception type.

4.6.1 System Reset Exception (0x00100)

The system reset exception is a non-maskable interrupt that is signaled to the 620 either through the assertion of an input signal to the chip ($\overline{\text{SRESET}}$) or internally during the power-on reset sequence.

The system reset exception is considered asynchronous in the manner that it has no relation to any specific instruction. Although the exception is asynchronous, the 620 synchronizes it to an instruction completion boundary. When the 620 detects a system reset exception condition, it completes the oldest instruction in the processor and then cancels all outstanding instructions before taking the system reset exception. As a result, the system reset exception is considered recoverable if the completing instruction (oldest instruction) does not cause an interrupt. If the completing instruction does not cause an exception, SRR0 holds the effective address of the next instruction that would have executed if the exception were not present. If the completing instruction causes an exception, SRR0 contains the effective address of the first instruction of the exception handler.

Note that system software can use HID0[15] to indicate why the processor is taking a system reset exception. To implement that, system software sets this bit at the end of the system reset exception routine. HID0[15] is cleared when the processor takes a hard reset.

Register settings for the system reset exception are described in Table 4-6.

Table 4-6. System Reset Exception—Register Settings

Register	Setting Description
SRR0	If the completing instruction does not cause an exception, SRR0 holds the effective address of the next instruction that would have executed if the exception were not present. If the completing instruction causes an exception, SRR0 contains the effective address of the first instruction of the exception handler.
SRR1	0–32 Loaded with equivalent bits from the MSR 33–36 Cleared 37–41 Loaded with equivalent bits from the MSR 42–47 Cleared 48–61 Loaded with equivalent bits from the MSR 62 Loaded from the equivalent MSR bit, MSR[RI], if the exception is recoverable; otherwise cleared. 63 Loaded with equivalent bit from the MSR Note that depending on the implementation, reserved bits in the MSR may not be copied to SRR1. If the processor state is corrupted to the extent that execution cannot resume reliably, the bit corresponding to MSR[RI], (SRR1[62]), is cleared.
MSR	SF Set to value of ISF ISF — POW 0 ILE — EE 0 PR 0 FP 0 ME — FE0 0 SE 0 BE 0 FE1 0 IP — IR 0 DR 0 PMM 0 RI 0 LE Set to value of ILE

The $\overline{\text{SRESET}}$ input provides a "warm" reset capability. This input is used to avoid causing the 620 to perform the entire power-on reset sequence, thereby preserving the contents of the architected registers. This capability is useful when recovering from certain checkstop or machine check states. When a system reset exception is taken, instruction execution continues at offset 0x00100 from the physical base address indicated by MSR[IP].

4.6.2 Machine Check Exception (0x00200)

The 620 implements the machine check exception as defined in the PowerPC architecture (OEA). It conditionally initiates a machine check exception after an address or data parity error occurred on the bus or in a cache, or after the machine check interrupt ($\overline{\text{MCP}}$) signal had been asserted. As defined in the OEA, the exception is not taken if the MSR[ME] is cleared.

Although a machine check exception is asynchronous, the 620 synchronizes it to an instruction completion boundary. When the 620 detects a machine check exception, it completes the oldest instruction in the processor and then cancels all outstanding instructions before invoking the machine check exception handler. If the completing instruction (oldest instruction) causes another type of exception, SRR0 contains the effective address of the first instruction of the exception handler. If the completing instruction does not cause another type of interrupt, SRR0 contains the effective address of the next instruction that would have executed if the exception were not present.

The 620 generates a machine check exception under the following conditions:

- $\overline{\text{MCP}}$ asserted—$\overline{\text{MCP}}$ is an input to the 620. When $\overline{\text{MCP}}$ is asserted and HID0[0] is set, the 620 takes a machine check exception. This is a recoverable machine check exception and is the only machine check exception in which the 620 copies the MSR[RI] bit to SRR1, instead of clearing it.
- Processor internal cache parity error—Parity checking for the 620's internal caches is enabled by setting HID0[1]. A machine check exception occurs when internal cache parity checking is enabled and a parity error is detected for either the data or the instruction cache. The resulting machine check exception is unrecoverable in the 620.
- Processor interface parity error—Parity checking for the 620 processor interface is enabled by setting HID0[2] for address parity checking or HID0[3] for data parity checking. A machine check exception occurs when address parity checking is enabled and a parity error is detected in the address bus. If data parity checking is enabled and a parity error is detected in the data bus, a machine check exception occurs regardless of the state of the MSR[ME] bit. The resulting machine check exceptions result in the 620 entering an internal checkstop state.
- L2 interface uncorrectable ECC error—ECC checking is enabled for the L2 interface when L2CR[46] is set. A machine check exception occurs when ECC checking is enabled and an uncorrectable error is detected. Note that the L2 can be accessed by either a processor or bus operation. The resulting machine check exception is unrecoverable in the 620.
- $\overline{\text{DERR}}$ bus signal—The $\overline{\text{DERR}}$ bus signal indicates that the data from a processor load or bus read is corrupted. $\overline{\text{DERR}}$ may be used to indicate an uncorrectable read memory error. A machine check exception occurs when BUSCSR[51] is set and a $\overline{\text{DERR}}$ assertion is detected. This machine check exception is unrecoverable in the

620. Note that a direct-store load last operation uses $\overline{\text{DERR}}$ to indicate that a direct-store error has occurred and to expect a reply. This case causes a DSI exception, but not a machine check exception.

Machine check conditions can be enabled and disabled using bits in the HID0 described in Table 4-7.

Table 4-7. Machine Check Enable Bits

HID0 Bit	Description
0	Enable machine check input signal
1	Enable cache parity checking
2	Enable machine check on address bus parity error.
3	Enable machine check on data bus parity error.

If MSR[ME] and the appropriate HID0 bits are set, the exception is recognized and handled; if MSR[ME] is cleared and the appropriate HID0 bits are set, the 620 will enter an internal checkstop state. When a processor is in checkstop state, instruction processing is suspended and generally cannot continue without restarting the processor. Note that many conditions may lead to the checkstop condition; the disabled machine check exception is only one of these.

Machine check exceptions are enabled when MSR[ME] = 1; this is described in Section 4.6.2.1, "Machine Check Exception Enabled (MSR[ME] = 1)." If MSR[ME] = 0 and a machine check occurs, the processor enters the checkstop state. Checkstop state is described in Section 4.6.2.2, "Checkstop State (MSR[ME] = 0)."

4.6.2.1 Machine Check Exception Enabled (MSR[ME] = 1)

When a machine check exception is taken, registers are updated as shown in Table 4-8.

Table 4-8. Machine Check Exception—Register Settings

Register	Setting Description
SRR0	On a best-effort basis implementations can set this to an EA of some instruction that was executing or about to be executing when the machine check condition occurred.
SRR1	0–32 Loaded from MSR 33–35 Cleared 36 Set if L2 ECC error 42 Set if L1 data cache parity error 43 Set if L1 instruction cache parity error 44 Set if machine-check signal input 45 Set if bus error ($\overline{\text{DERR}}$) 46 Set if bus data parity error 47 Set if bus address parity error 48–61 Loaded from MSR 62 Loaded from MSR[62] if the processor is in a recoverable state; otherwise cleared 63 Loaded from MSR
MSR	SF Set to value of ISF PR 0 BE 0 DR 0 ISF — FP 0 FE1 0 PMM 0 POW 0 ME [1] 0 IP — RI 0 ILE — FE0 0 IR 0 LE Set to value of ILE EE 0 SE 0

[1] Note that when a machine check exception is taken, the exception handler should set MSR[ME] as soon as it is practical to handle another machine check exception. Otherwise, subsequent machine check exceptions cause the processor to automatically enter the checkstop state.

The machine check exception is usually unrecoverable in the sense that execution cannot resume in the same context that existed before the exception. If the condition that caused the machine check does not otherwise prevent continued execution, MSR[ME] is set to allow the processor to continue execution at the machine check exception vector address. Typically earlier processes cannot resume; however, the operating systems can then use the machine check exception handler to try to identify and log the cause of the machine check condition.

When a machine check exception is taken, instruction execution resumes at offset 0x00200 from the physical base address indicated by MSR[IP].

4.6.2.2 Checkstop State (MSR[ME] = 0)

When a processor is in the checkstop state, instruction processing is suspended and generally cannot resume without the processor being reset. The contents of all latches are frozen within two cycles upon entering checkstop state.

The 620 enters checkstop state when any of three conditions are present:

- If MSR[ME] = 0 and a machine check exception occurs due to any of the reasons mentioned previously.
- If $\overline{\text{CHECKSTOP}}$ is asserted externally.
- If an internal watchdog time-out is generated.
- If a processor interface parity error occurs.

Upon entering the checkstop state the processor clocks are stopped and internal state is kept. The $\overline{\text{CHECKSTOP}}$ signal is asserted to notify the system when the checkstop is caused by a machine check exception with MSR[ME] = 0, or a processor internal watchdog time-out or processor interface parity error occurs.

For all machine check events except those initiated by processor interface parity errors, the number of cycles between the time in which a machine check event is detected and the time in which the $\overline{\text{CHECKSTOP}}$ signal is asserted is undetermined since the processor synchronizes the Machine Check event to an instruction completion boundary before it reports it. When processor interface parity errors cause the checkstop state to be entered the CHECKSTOP signal is asserted two processor clock cycles after the parity error is detected, and all latches are frozen within six processor clock cycles.

Note that not all PowerPC processors provide the same level of error checking. The reasons a processor can enter checkstop state are implementation-dependent.

4.6.3 DSI Exception (0x00300)

A DSI exception occurs when no higher priority exception exists and a data memory access cannot be performed. The DSI exception is implemented as it is defined in the PowerPC architecture (OEA), and occurs for all the conditions defined in the PowerPC architecture. A DSI exception also occurs for direct-store load or store errors.

Note that a DABR breakpoint match does not cause a DSI exception when the 620 is in tracing mode (MSR[SE]=1) or when the performance monitor is turned on.

4.6.4 ISI Exception (0x00400)

An ISI exception occurs when no higher priority exception exists and an attempt to fetch the next instruction fails. This exception is implemented as it is defined by the PowerPC architecture (OEA). When an ISI exception is taken, instruction execution resumes at offset 0x00400 from the physical base address indicated by MSR[IP].

4.6.5 External Interrupt Exception (0x00500)

An external interrupt is signaled to the processor by the assertion of the external interrupt signal ($\overline{\text{INT}}$). The $\overline{\text{INT}}$ signal is expected to remain asserted until the 620 takes the external interrupt exception. If the external interrupt signal is negated early, recognition of the interrupt request is not guaranteed.

Once an external interrupt is detected and external interrupts are enabled, MSR[EE] = 1, the 620 completes the oldest instruction in the processor, cancels all outstanding instructions, and takes the external interrupt as defined in the OEA.

After the 620 begins execution of the external interrupt handler, the system can safely negate the $\overline{\text{INT}}$. The interrupt may be delayed by other higher priority exceptions or if the MSR[EE] bit is cleared when the exception occurs. Register settings for this exception are described in Chapter 6, "Exceptions," in *The Programming Environments Manual.*

When an external interrupt exception is taken, instruction execution resumes at offset 0x00500 from the physical base address indicated by MSR[IP].

4.6.6 Alignment Exception (0x00600)

The 620 implements the alignment exception as defined by the PowerPC architecture (OEA). An alignment exception is initiated when any of the following conditions is met:

- The operand of a floating-point load or store is not word-aligned
- The operand of a fixed-point load-double or store-double is not word-aligned
- The operand of **lmw**, **stmw**, **lwarx**, or **stwcx.** is not word-aligned
- The operand of **ldarx** or **stdcx.** is not doubleword-aligned.
- A floating-point memory access is attempted to a direct-store segment.
- If the address for a **dcbz** instruction is to a memory space marked write-through or cache-inhibited, an alignment exception is taken.

4.6.7 Program Exception (0x00700)

The 620 implements the program exception as it is defined by the PowerPC architecture (OEA). A program exception occurs when no higher priority exception exists and one or more of the exception conditions defined in the OEA occur.

The 620 invokes the system illegal instruction program exception when it detects any instruction from the illegal instruction class.

The 620 fully decodes the SPR field of the instruction. If an undefined SPR is specified, a program exception is taken.

The UISA defines the **mtspr** and **mfspr** instructions with the record bit (Rc) set to cause a program exception or provide a boundedly undefined result. In the 620, the appropriate CR should be treated as undefined. Likewise, the PowerPC architecture states that the Floating Compared Unordered (**fcmpu**) or Floating Compared Ordered (**fcmpo**) instruction with the record bit set can either cause a program exception or provide a boundedly undefined result. In the 620, CR field BF for these cases should be treated as undefined.

When a program exception is taken, instruction execution resumes at offset 0x00700 from the physical base address indicated by MSR[IP].

Two MSR bits are used to determine the mode used for handling floating-point exceptions The 620 operates the floating-point unit (FPU) in either ignore exception mode (FE0 = 0, FE1 = 0) or precise mode (FE0, FE1= (1,0), (0,1), or (1,1)). The FPU uses an internal pipeline to gain overlapped execution of instructions. If an exception occurs during the floating-point arithmetic or conversion operations, the FPU sends the exception signal to the completion block in the instruction flow unit, and the FPU may continue its operation as normal. The completion block performs the necessary exception handling as defined in PowerPC architecture. Register settings for this exception are described in Chapter 6, "Exceptions," in *The Programming Environments Manual.*

4.6.8 Floating-Point Unavailable Exception (0x00800)

The floating-point unavailable exception is implemented as defined in the PowerPC architecture. A floating-point unavailable exception occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, or move instructions), and the floating-point available bit in the MSR is disabled, (MSR[FP] = 0). Register settings for this exception are described in Chapter 6, "Exceptions," in *The Programming Environments Manual.*

When a floating-point unavailable exception is taken, instruction execution resumes at offset 0x00800 from the physical base address indicated by MSR[IP].

4.6.9 Decrementer Exception (0x00900)

The decrementer exception is implemented in the 620 as it is defined by the PowerPC architecture. The decrementer exception occurs when no higher priority exception exists, a decrementer exception condition occurs (for example, the decrementer register has completed decrementing), and MSR[EE] = 1. In the 620, the decrementer register is decremented at the bus clock rate. Register settings for this exception are described in Chapter 6, "Exceptions," in *The Programming Environments Manual.*

When a decrementer exception is taken, instruction execution resumes at offset 0x00900 from the physical base address indicated by MSR[IP].

4.6.10 System Call Exception (0x00C00)

A system call exception occurs when a System Call (**sc**) instruction is executed. In the 620, the system call exception is implemented as it is defined in the PowerPC architecture. Register settings for this exception are described in Chapter 6, "Exceptions," in *The Programming Environments Manual.*

When a system call exception is taken, instruction execution resumes at offset 0x00C00 from the physical base address indicated by MSR[IP].

4.6.11 Trace Exception (0x00D00)

The trace exception is taken when the single step trace enable bit (MSR[SE]) or the branch trace enable bit (MSR[BE]) is set and an instruction successfully completes. When a trace exception is taken, the values written to SRR1 are implementation-specific; those values for the 620 are shown in Table 4-9.

Table 4-9. Trace Exception—Register Settings

Register	Setting
SRR0	Address of the instruction the processor would have attempted to execute next if no exception condition were present.
SRR1	0–32 Copied from MSR 33 Set 34 Cleared 35 Set for load, **dcbt**, or **dcbtst** instruction 36 Set for store instruction (won't set either bit for other cache control instructions) 37–41 Copied from MSR 42 Set for **lswx** or **stswx** instruction. 43 Set for **mtspr** to any privileged register, or if SLB or TLB is updated since last exception. 44 Set for taken branch instruction. 45–47 Cleared 48–63 Copied from MSR.

The 620 does take the trace exception if the source instruction causes another type of exception. Therefore, the 620 does not take the trace exception on **sc**, **rfid** (or **rfi**), and **trap** instructions whose condition is true.

When either MSR[SE] or MSR[BE] is set, the 620 operates in single-instruction serialization mode.

When a trace exception is taken, instruction execution resumes as offset 0x00D00 from the base address indicated by MSR[IP].

4.6.12 Floating-Point Assist Exception (0x00E00)

The optional floating-point assist exception defined by the PowerPC architecture is not implemented in the 620.

4.6.13 Performance Monitoring Interrupt (0x00F00)

The PowerPC 620 performance monitor is a software-accessible mechanism that provides detailed information concerning the dispatch, execution, completion, and memory access of PowerPC instructions. The performance monitor is provided to help system developers to debug their systems and to increase system performance with efficient software, especially in a multiprocessor system where memory hierarchy behavior must be monitored and studied in order to develop algorithms that schedule tasks (and perhaps partition them) and distribute data optimally.

Register settings are described in Table 4-10.

Table 4-10. Performance Monitoring Interrupt—Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present
SRR1	0–32 Loaded from MSR 33 Set if the content of SDA and SIA register is for the same instruction. 34–36 Cleared 37–41 Loaded from MSR 42–47 Cleared 48–63 Loaded from MSR

The performance monitor uses the following SPRs:

- Monitor mode control register 0 and 1 (MMCR0 and MMCR1)—Controls the behavior of the performance monitor. Provides the ability to select the events to count and when they will be counted, set the threshold value, select the time base input, enable history mode, and select the conditions that enable a performance monitor exception.
- Performance monitor counters 1–8 (PMC1–PMC8)—Store the number of times a software selectable event (maximum of one event per counter at a time) has occurred since the performance monitor was enabled for counting.
- Sampled instruction address (SIA)—Stores the address of a sampled instruction.
- Sampled data address (SDA)—Stores the address associated with the data used by the sampled instruction.

The 620 supports a performance monitor interrupt that is caused by a counter negative condition or by a time-base flipped bit counter defined in the MMCR0 register.

As with other PowerPC exceptions, the performance monitoring interrupt follows the normal PowerPC exception model with a defined exception vector offset (0x00F00). The priority of the performance monitoring interrupt is below the external interrupt and above the decrementer interrupt. The contents of the SIA and SDA are described in Section 2.1.2.7, “Performance Monitor Registers.” The performance monitor is described in Chapter 10, “Performance Monitor.”

4.6.14 Instruction Address Breakpoint Exception (0x01300)

The instruction address breakpoint exception occurs when an attempt is made to execute an instruction that matches the address in the instruction address breakpoint register (IABR) and the breakpoint is enabled. Specifically the following three conditions must be met:

- Instruction address (0–61) = IABR[0–61]
- IABR[62] = 1
- IABR[63] = MSR[IR]

The instruction that triggers the instruction address breakpoint exception is not executed before the exception handler is invoked. The vector offset of the instruction address breakpoint exception is 0x01300. Register settings are described in Table 4-11.

Table 4-11. Instruction Address Breakpoint Exception—Register Settings

Register	Setting Descriptions
SRR0	Set to the effective address of the instruction that causes the exception.
SRR1	0–32 Loaded from MSR 33–36 Cleared 37–41 Loaded from MSR 42–47 Cleared 48–63 Loaded from MSR.

4.6.15 System Management Interrupt (0x01400)

The 620 implements a system management interrupt exception, which is not defined by the PowerPC architecture. The system management exception is very similar to the external interrupt exception and is particularly useful in implementing the nap mode. It has priority over an external interrupt and it uses a different exception vector in the exception table (at offset 0x01400).

The system management interrupt exception is taken when $\overline{\text{SMI}}$ is asserted and the exception is enabled by MSR[EE].

Register settings for the system management exception are described in Table 4-12.

Table 4-12. System Management Interrupt Exception—Register Settings

Register	Setting Descriptions
SRR0	Set to the effective address of the instruction that causes the exception.
SRR1	0–32 Loaded from MSR 33–36 Cleared 37–41 Loaded from MSR 42–47 Cleared 48–63 Loaded from MSR.

Like the external interrupt, a system management interrupt is signaled to the 620 by the assertion of an input signal. The system management interrupt signal ($\overline{\text{SMI}}$) is expected to remain asserted until the exception is taken. If the $\overline{\text{SMI}}$ signal is negated early, recognition of the exception request is not guaranteed. After the 620 begins execution of the exception handler, the system can safely negate the $\overline{\text{SMI}}$ signal. When MSR[EE] is set and assertion of the $\overline{\text{SMI}}$ signal is detected, the 620 completes the oldest instruction in the processor, cancels all outstanding instructions, and takes the system management interrupt exception.

When the exception is taken, the 620 begins fetching instructions from exception vector offset, 0x01400.

Chapter 5
Memory Management

This chapter describes the PowerPC 620 microprocessor's implementation of the memory management unit (MMU) specifications provided by the operating environment architecture (OEA) for PowerPC processors. The primary function of the MMU in a PowerPC processor is the translation of logical (effective) addresses to physical addresses (referred to as real addresses in the architecture specification) for memory accesses, I/O accesses (most I/O accesses are assumed to be memory-mapped), and direct-store interface accesses (which are optional to the PowerPC architecture). In addition, the MMU provides access protection on a segment, block or page basis. In addition, the 620 implements the optional 64-bit bridge facility defined in the OEA. This facility provides resources that may allow some 32-bit operating systems to operate in the 64-bit addressing environment of the 620.

This chapter describes the specific hardware used to implement the MMU model of the OEA in the 620. Refer to Chapter 7, "Memory Management," in *The Programming Environments Manual* for a complete description of the conceptual model.

Two general types of accesses generated by PowerPC processors require address translation—instruction accesses and data accesses to memory generated by load and store instructions. Generally, the address translation mechanism is defined in terms of segment descriptors and page tables used by PowerPC processors to locate the effective-to-physical address mapping for instruction and data accesses. The segment information translates the effective address to an interim virtual address, and the page table information translates the interim virtual address to a physical address.

The 64-bit MMU model implements segment descriptors, which are used to generate the interim virtual addresses, as entries in a segment table, which are managed in memory much as page tables are in both the 32- and 64-bit MMU models. On 32-bit implementations, the segment descriptors are stored as on-chip segment registers, which are emulated in the 620. In addition, a unified translation lookaside buffer (UTLB) keeps recently-used page address translations on-chip.

The 620 implements two level of segmented address translation. The first-level translation is accomplished by separate instruction and data 64-entry, fully-associative effective to physical translation caches (EPATs). The EPATs cache the effective to physical translation pairs that are returned from the second-level MMU. The second-level MMU consists of a

20-entry, fully-associate SLB, and a 128-entry, 2-way set-associative TLB. The second-level MMU is shared between the first-level instruction and data MMUs.

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations on-chip. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor special-purpose registers (SPRs). There are separate instruction and data BAT mechanisms, and in the 620, they reside in the instruction and data MMUs respectively.

The MMUs, together with the exception processing mechanism, provide the necessary support for the operating system to implement a paged virtual memory environment and for enforcing protection of designated memory areas. Exception processing is described in Chapter 4, "Exceptions." Section 4.4, "Exception Processing," describes the MSR, which controls some of the critical functionality of the MMUs.

5.1 MMU Overview

The 620 implements the memory management specification of the PowerPC OEA for 64-bit implementations, including the optional 64-bit bridge facility. Thus it provides 2^{64} bytes of effective address space accessible to supervisor and user programs with a 4-Kbyte page size and 256-Mbyte segment size. The 620's MMU provides an interim virtual address (80 bits) and hashed page tables for the generation of 40-bit physical addresses. PowerPC processors also have a BAT mechanism for mapping large blocks of memory. Block sizes range from 128 Kbyte to 256 Mbyte and are software-programmable.

Basic features of the 620 MMU implementation defined by the OEA are as follows:

- Support for physical addressing mode—Logical-to-physical address translation can be disabled separately for data and instruction accesses.
- Block address translation—Each of the BAT array entries (four IBAT entries and four DBAT entries) provides a mechanism for translating blocks as large as 256 Mbytes from the 64-bit effective address space into the physical memory space. This can be used for translating large address ranges whose mappings do not change frequently.
- Direct-store segments (optional to the PowerPC architecture)—If the T bit in the indexed segment descriptor is set for any load or store request, this request accesses a direct-store segment; bus activity is different and the memory space used has different characteristics with respect to how it can be accessed. The address used on the bus consists of bits from the EA and the segment descriptor.
- Segmented address translation—The 64-bit effective address is extended to an 80-bit virtual address by having EA[0–35] select a 52-bit virtual segment ID (VSID), EA[36–51] forming bits 52–67, which together form a virtual page number (VPN). The remaining 12 bits, EA[52–63] remain intact to form the low-order 12 bits of the virtual address.

The 620 also provides the following features that are not required by the PowerPC architecture:

- A 20-entry content-addressable memory (CAM) segment lookaside buffer (SLB) with FIFO replacement algorithm for storing the most recently used segment table entries (STEs). The SLB is fully-associative.
- Effective-to-Physical-Address Translators (EPATs)—The instruction and data first-level MMUs each contain 64-entry, fully-associative EPATs that cache effective-to-physical translation pairs that are returned from the second level MMU. The EPATs are implemented as CAMs with an invalid-first replacement algorithm.
- Segment table lookup operations performed in hardware—The MMU attempts to fetch the STE, which contains the segment descriptor, from the SLB on-chip. If the STE is not in the SLB (that is, a SLB miss occurs), the hardware performs a segment look-up operation (using a hashing function) to search for the STE.
- Unified translation lookaside buffer (UTLB)—The 128-entry, 2-way set UTLB saves recently-used page address translations on-chip for both instruction and data accesses. Valid UTLB entries are also forwarded to the EPATs in the first-level IMMU and DMMU.
- Table search operations performed in hardware—If a page address translation is not found in the EPATs, the MMU attempts to locate the STE. If the translation is not found in the SLB (that is, a SLB miss occurs), the hardware performs a table search operation (using a hashing function) to search for the STE. Once the segment descriptor is located, an 80-bit virtual address is formed and the MMU attempts to fetch the PTE, which contains the physical address, from the TLB. If the translation is not found in a TLB (that is, a TLB miss occurs), the hardware performs a table-search operation (similar to that performed for the STE) to search for the PTE.
- SLB invalidation—The 620 implements the optional SLB Invalidate Entry (**slbie**) and SLB Invalidate All (**slbia**) instructions, which can be used to invalidate SLB and EPAT entries. For more information on the **slbie** and **slbia** instructions, see Section 5.4.4, "SLB Invalidation."
- TLB invalidation—The 620 implements the optional TLB Invalidate Entry (**tlbie**) and TLB Synchronize (**tlbsync**) instructions, which can be used to invalidate TLB entries. For more information on the **tlbie** and **tlbsync** instructions, see Section 5.4.6, "TLB Invalidation."
- The optional 64-bit bridge—The OEA defines an additional, optional bridge to the 64-bit architecture that may make it easier for 32-bit operating systems to migrate to 64-bit processors. The 64-bit bridge retains certain aspects of the 32-bit architecture that otherwise are not supported, and in some cases not permitted, by the 64-bit version of the architecture. In processors that implement this bridge, segment descriptors are implemented by using 16 SLB entries to emulate segment registers, which, like those defined for the 32-bit architecture, divide the 32-bit memory space (4 Gbytes) into sixteen 256-Mbyte segments. These segment descriptors however

use the format of the segment table entries as defined in the 64-bit architecture and are maintained in SLBs rather than in hardware segment registers as defined by the architecture for 32-bit addressing.

Table 5-1 summarizes the 620 MMU features, including those defined by the PowerPC architecture (OEA) for 32-bit processors and those specific to the 620.

Table 5-1. OEA-Defined PowerPC 620 Microprocessor MMU Features Summary

Feature	Conventional	64-Bit Bridge
Address ranges	2^{64} bytes of effective address	2^{32} bytes of effective address
	2^{80} bytes of virtual address	2^{52} bytes of virtual address
	$\leq 2^{64}$ bytes of physical address	$\leq 2^{32}$ bytes of physical address
Page size	4 Kbytes	Same
Segment size	256 Mbytes	Same
Block address translation	Range of 128 Kbyte–256 Mbyte	Same
	Implemented with IBAT and DBAT registers in BAT array	Same
Memory protection	Segments selectable as no-execute	Same
	Pages selectable as user/supervisor and read-only	Same
	Blocks selectable as user/supervisor and read-only	Same
Page history	Referenced and changed bits defined and maintained	Same
Page address translation	Translations stored as PTEs in hashed page tables in memory	Same
	Page table size determined by size programmed into SDR1 register	Same
TLBs	Instructions for maintaining optional TLBs	Same
Segment descriptors	Stored as STEs in hashed segment tables in memory	Stored in 16 SLB entries in the same format as the STEs defined for 64-bit implementations.
	Instructions for maintaining optional SLBs	16 SLB entries are required to emulate the segment registers defined for 32-bit addressing. The **slbie** and **slbia** instructions should not be executed when using the 64-bit bridge.

Table 5-2 lists implementation-specific features of the 620 MMU.

Table 5-2. PowerPC 620 Microprocessor-Specific MMU Features Summary

Feature	Description
Separate MMUs	Separate memory management units (MMUs) for instructions and data
EPATs	Independent 64-entry fully-associative effective-to-physical address translation (EPAT) cache with invalid-first replacement algorithm for instructions and data
TLBs	Unified instruction and data TLB
	TLB is 128-entry and two-way set-associative
	LRU replacement algorithm
	Hardware broadcast of TLB and control instructions
SLBs	20-entry CAM segment lookaside buffer (SLB) with FIFO replacement algorithm
Hardware miss handling	SLB, TLB, and EPAT cache miss handling performed by 620 hardware
Referenced/changed bits	Hardware update of page frame table referenced and changed bits
40 bit addressing	40-bit physical memory address for up to one terabyte

5.1.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next instruction. The effective address is translated to a physical address according to the procedures described in Chapter 7, "Memory Management," in *The Programming Environments Manual*, augmented with information in this chapter. The memory subsystem uses the physical address for the access.

For a complete discussion of effective address calculation, see Section 2.3.2.3, "Effective Address Calculation."

5.1.2 MMU Organization

The PowerPC architecture defines two methods of address translation—segmented address translation and block address translation (BAT). These translations occur in parallel, with the BAT taking precedence.

The segmented address translation on the 620 is implemented in two levels. The first-level translation is accomplished by two separate 64-entry, fully-associative EPATs, one for instruction fetches and the other for data accesses. The EPAT caches translation pairs that are returned from the second-level MMU. The second-level MMU consists of a more traditional 20-entry, fully-associative SLB and a 128-entry, 2-way set-associative TLB. This second-level MMU is used for both instruction and data accesses.

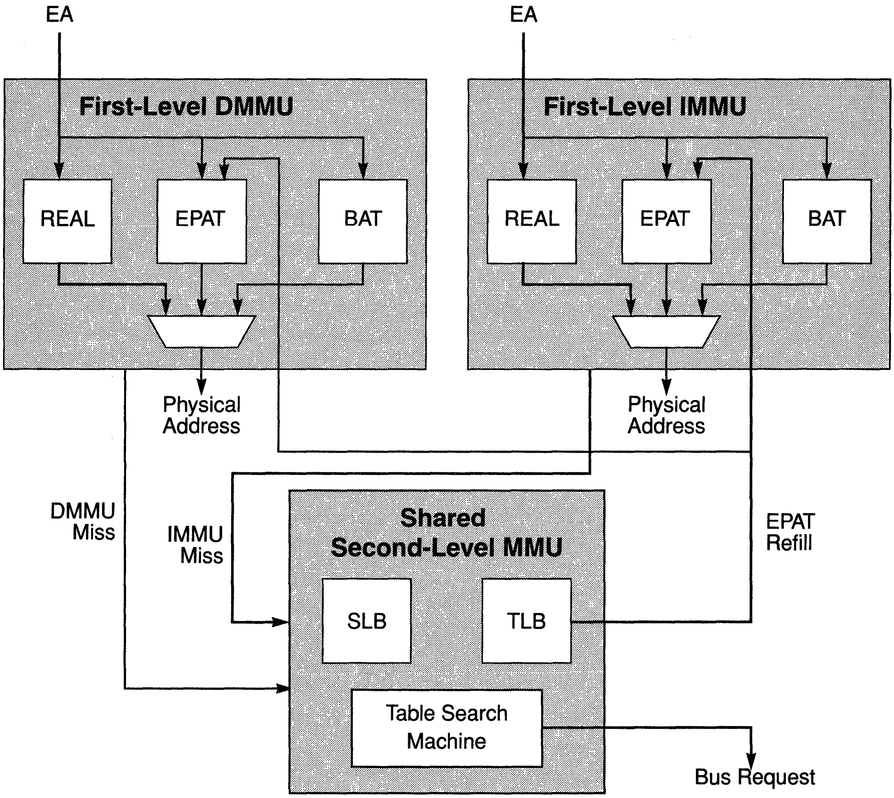


Figure 5-1. Two-Level MMU Organization

The second-level MMU consists of a hardware table-search mechanism that is initiated to reload the EPAT, SLB, or TLB when both MMU levels fail to translate an effective address. This mechanism searches translation tables that have been previously constructed in main memory. The table-search mechanism is nonspeculative in that it waits for any refill request to commit before generating bus requests.

The MMU does not make requests to the L1 cache. Instead, it immediately makes requests into the L2 and main memory. If the requested cache block was modified in the L1 cache, that block is copied back to main memory before the MMU's request is serviced. The MMU's request for a cache block always appears as a cacheable coherent burst read operation on the bus, and any writes caused by referenced and changed bits appear as write-through coherent stores. Stores for referenced or changed updates are always single byte store operations.

Figure 5-2 shows the conceptual organization of the MMU in a 64-bit implementation; note that it does not describe the specific hardware used to implement the memory management function for a particular processor and other hardware features (invisible to the system software) not depicted in the figure may be implemented. For example, the memory management function can be implemented with parallel MMUs that translate addresses for instruction and data accesses independently.

The instruction addresses shown in the figure are generated by the processor for sequential instruction fetches and addresses that correspond to a change of program flow. Memory addresses are generated by load and store instructions (both for memory and the direct-store interface) and by cache instructions.

As shown in Figure 5-2, after an address is generated, the higher-order bits of the effective address, EA0–EA51 (or a smaller set of address bits, EA0–EA*n*, in the cases of blocks), are translated into physical address bits PA0–PA27. The lower-order address bits, A28–A39 are untranslated and therefore identical for both effective and physical addresses. After translating the address, the MMU passes the resulting 40-bit physical address to the memory subsystem.

In addition to the higher-order address bits, the MMU automatically keeps an indicator of whether each access was generated as an instruction or data access and a supervisor/user indicator that reflects the state of the PR bit of the MSR when the effective address was generated. In addition, for data accesses, there is an indicator of whether the access is for a load or a store operation. This information is then used by the MMU to appropriately direct the address translation and to enforce the protection hierarchy programmed by the operating system. See Section 2.1.1, “Register Set,” for more information about the MSR.

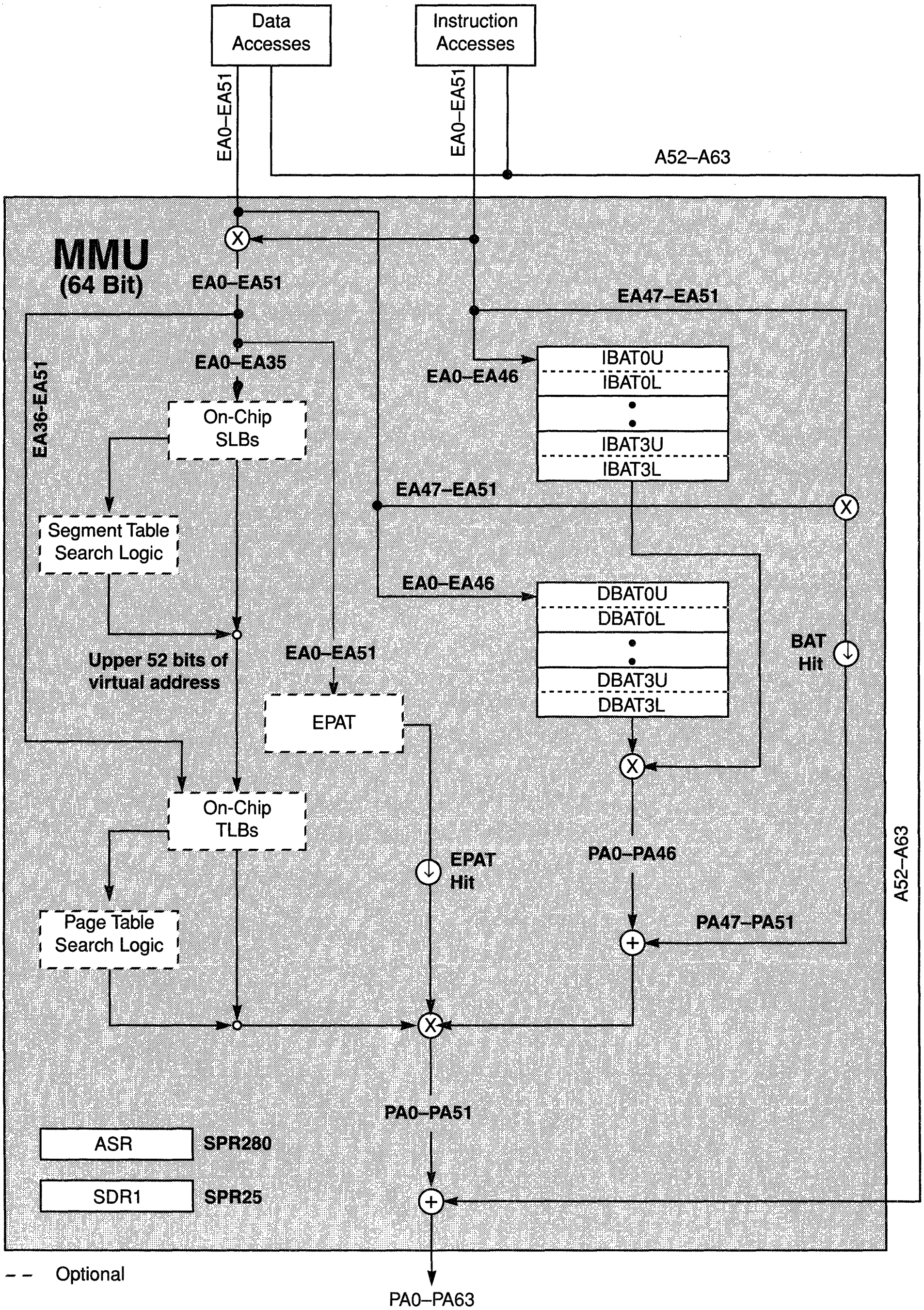


Figure 5-2. MMU Conceptual Block Diagram

As shown in Figure 5-2, the 620 implements a unified translation lookaside buffer (UTLB) and supports the automatic search of the page tables for page table entries (PTEs), both of which the OEA defines as optional.

In 64-bit implementations, the address space register (ASR) defines the physical address of the base of the segment table in memory. The segment table entries (STEs) contain the segment descriptors, which define the virtual address for the segment. Some 64-bit implementations may have dedicated hardware to search for STEs in memory, and copies of STEs may be cached on-chip in segment lookaside buffers (SLBs) for quicker access.

The 620's ASR implementation includes the optional V bit, which together with MSR[SF] enable the use of the 64-bit bridge functionality.

5.1.3 Address Translation Mechanisms

The PowerPC architecture defines four types of address translation:

- Block address translation—Translates the block number for blocks that range in size from 128 Kbyte to 256 Mbyte
- Page address translation—Translates the page frame address for a 4-Kbyte page size
- Direct-store address translation—Used to generate direct-store interface accesses on the external bus; not optimized for performance. Direct-store addressing is optional and is present in the 620 for compatibility only
- Real addressing mode address translation—When address translation is disabled, the physical address is identical to the effective address

Figure 5-3 shows the four address translation mechanisms provided by the MMU. In addition, it shows the 620-specific EPATs, which are a cache of the address translations for page address and direct-store accesses.

The segment descriptors shown in the figure control both the page and direct-store segment address translation mechanisms. When an access uses the page or direct-store segment address translation, the appropriate segment descriptor is required. In 64-bit implementations, the segment descriptor is located via a search of the segment table in memory for the appropriate segment table entry (STE). One of the 16 emulated segment registers (which contain segment descriptors) is selected by the highest-order effective address bits.

Processors, such as the 620, that implement the 64-bit bridge divide the 32-bit address space into sixteen 256-Mbyte segments defined by a table of 16 STEs maintained in 16 SLB entries.

A control bit in the corresponding segment descriptor then determines if the access is to memory (memory-mapped) or to a direct-store segment. Note that the direct-store interface is present to allow certain older I/O devices to use this interface. When an access is determined to be to the direct-store interface space, the implementation invokes an elaborate hardware protocol for communication with these devices. The direct-store interface protocol is not optimized for performance, and therefore, its use is discouraged. The most efficient method for accessing I/O is by memory-mapping the I/O areas.

For memory accesses translated by a segment descriptor, the interim virtual address is generated using the information in the segment descriptor. Page address translation corresponds to the conversion of this virtual address into the 64-bit physical address used by the memory subsystem. In some cases, the physical address for the page resides in an on-chip EPAT or TLB and is available for quick access. However, if the page address translation misses in both the EPAT and TLB, the MMU searches the page table in memory (using the virtual address information and a hashing function) to locate the required physical address. Some implementations may have dedicated hardware to perform the page table search automatically, while others may define an exception handler routine that searches the page table with software.

Block address translation occurs in parallel with page and direct-store segment address translation and is similar to page address translation, except that there are fewer upper-order effective address bits to be translated into physical address bits (more lower-order address bits (at least 17) are untranslated to form the offset into a block). Also, instead of segment descriptors and a page table, block address translations use the on-chip BAT registers as a BAT array. If an effective address matches the corresponding field of a BAT register, the information in the BAT register is used to generate the physical address; in this case, the results of the page translation (occurring in parallel) are ignored. Note that a matching BAT array entry takes precedence over a translation provided by the segment descriptor in all cases (even if the segment is a direct-store segment).

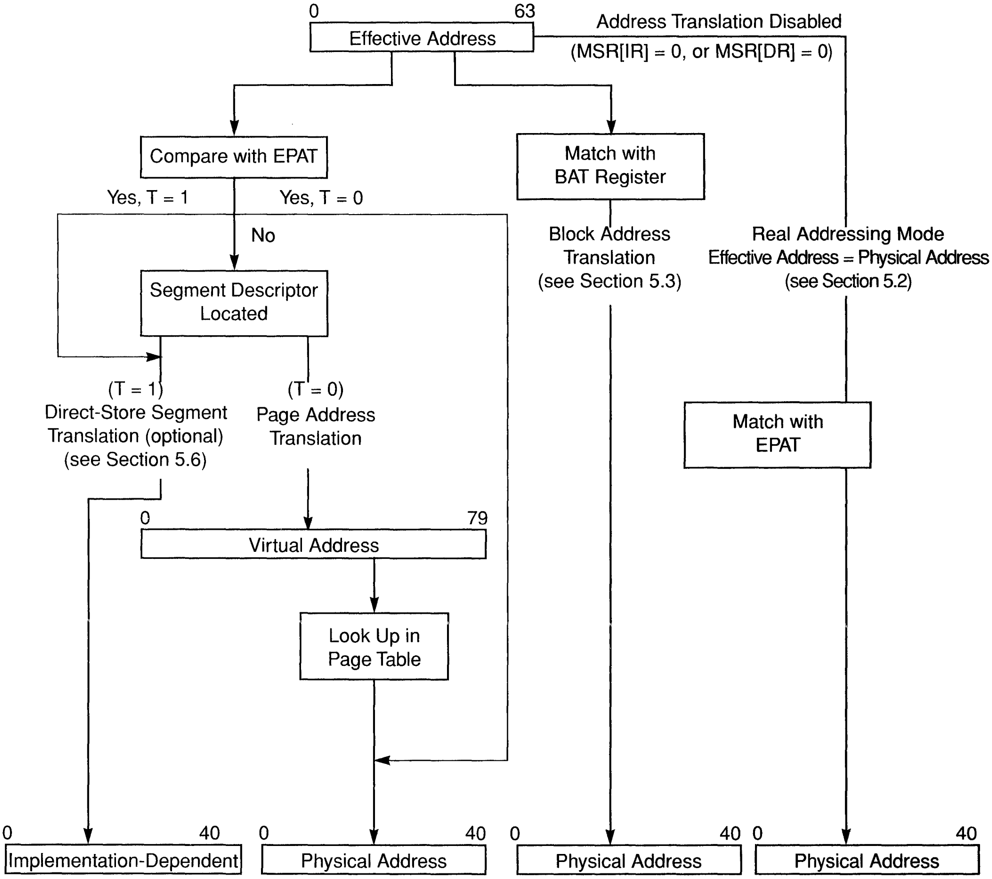


Figure 5-3. Address Translation Types

Direct-store address translation is used when the direct-store translation control bit (T bit) in the corresponding segment descriptor is set. In this case, the remaining information in the segment descriptor is interpreted as identifier information that is used with the remaining effective address bits to generate the packets used in a direct-store interface access on the external interface; additionally, no TLB lookup or page table search is performed.

Translation is disabled for real addressing mode. In this case the physical address generated is identical to the effective address. Instruction and data address translation is enabled with the MSR[IR] and MSR[DR] bits, respectively. Thus when the processor generates an access, and the corresponding address translation enable bit in MSR (MSR[IR] for instruction accesses and MSR[DR] for data accesses) is cleared, the resulting physical address is identical to the effective address and all other translation mechanisms are ignored.

5.1.4 Memory Protection Facilities

In addition to the translation of effective addresses to physical addresses, the MMUs provide access protection of supervisor areas from user access and can designate areas of memory as read-only as well as no-execute. Table 5-3 shows the protection options supported by the MMUs for pages.

Table 5-3. Access Protection Options for Pages

Option	User Read		User Write	Supervisor Read		Supervisor Write
	I-Fetch	Data		I-Fetch	Data	
Supervisor-only	—	—	—	√	√	√
Supervisor-only-no-execute	—	—	—	—	√	√
Supervisor-write-only	√	√	—	√	√	√
Supervisor-write-only-no-execute	—	√	—	—	√	√
Both user/supervisor	√	√	√	√	√	√
Both user-/supervisor-no-execute	—	√	√	—	√	√
Both read-only	√	√	—	√	√	—
Both read-only-no-execute	—	√	—	—	√	—

√ Access permitted
— Protection violation

The operating system determines whether instruction can be fetched from an area of memory for which the no-execute option is provided in the segment descriptor. Each of the remaining options is enforced based on a combination of information in the segment descriptor and the page table entry. Thus, the supervisor-only option allows only read and write operations generated while the processor is operating in supervisor mode (corresponding to MSR[PR] = 0) to access the page. User accesses that map into a supervisor-only page cause an exception to be taken.

Finally, the VEA and OEA define a facility that allows pages or blocks to be designated as guarded preventing out-of-order accesses that may cause undesired side effects. For example, areas of the memory map used to control I/O devices can be marked as guarded so that accesses (for example, instruction prefetches) do not occur unless they are explicitly required by the program.

For more information on memory protection, see “Memory Protection Facilities,” in Chapter 7, “Memory Management,” in the *The Programming Environments Manual.*

5.1.5 Page History Information

The MMUs of PowerPC processors also define referenced (R) and changed (C) bits in the page address translation mechanism that can be used as history information relevant to the page. This information can then be used by the operating system to determine which areas of memory to write back to disk when new pages must be allocated in main memory. While these bits are initially programmed by the operating system into the page table, the architecture specifies that the R and C bits may be maintained either by the processor hardware (automatically) or by some software-assist mechanism that updates these bits when required.

Implementation Notes—Because the 620 performs TLB- and SLB-miss table search operations nonspeculatively, it does not speculatively update the reference bit in the PTE or STE. The 620 also does not set the R or C bit if the instruction causing the update causes an exception. For more information, see Section 5.4.1, "Page History Recording."

5.1.6 General Flow of MMU Address Translation

The following sections describe the general flow used by PowerPC processors to translate effective addresses to virtual and then physical addresses.

5.1.6.1 Real Addressing Mode and Block Address Translation Selection

When an instruction or data access is generated and the corresponding instruction or data translation is disabled (MSR[IR] = 0 or MSR[DR] = 0), real addressing mode is used (physical address equals effective address) and the access continues to the memory subsystem as described in Section 5.2, "Real Addressing Mode."

Figure 5-4 shows the flow used by the MMUs in determining whether to select real addressing mode, block address translation or to use the segment descriptor to select either direct-store interface or page address translation.

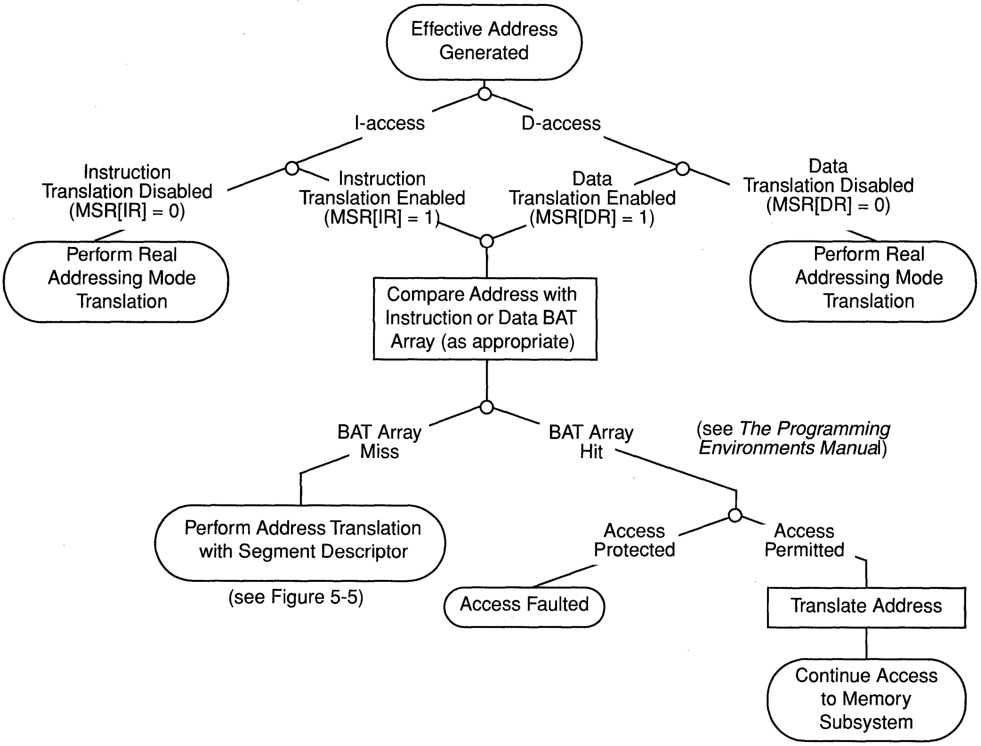

Effective Address Generated
I-access
D-access
Instruction Translation Disabled (MSR[IR] = 0)
Instruction Translation Enabled (MSR[IR] = 1)
Data Translation Enabled (MSR[DR] = 1)
Data Translation Disabled (MSR[DR] = 0)
Perform Real Addressing Mode Translation
Compare Address with Instruction or Data BAT Array (as appropriate)
Perform Real Addressing Mode Translation
BAT Array Miss
BAT Array Hit
(see The Programming Environments Manual)
Perform Address Translation with Segment Descriptor
(see Figure 5-5)
Access Protected
Access Permitted
Access Faulted
Translate Address
Continue Access to Memory Subsystem

Figure 5-4. General Flow of Address Translation (Real Addressing Mode and Block)

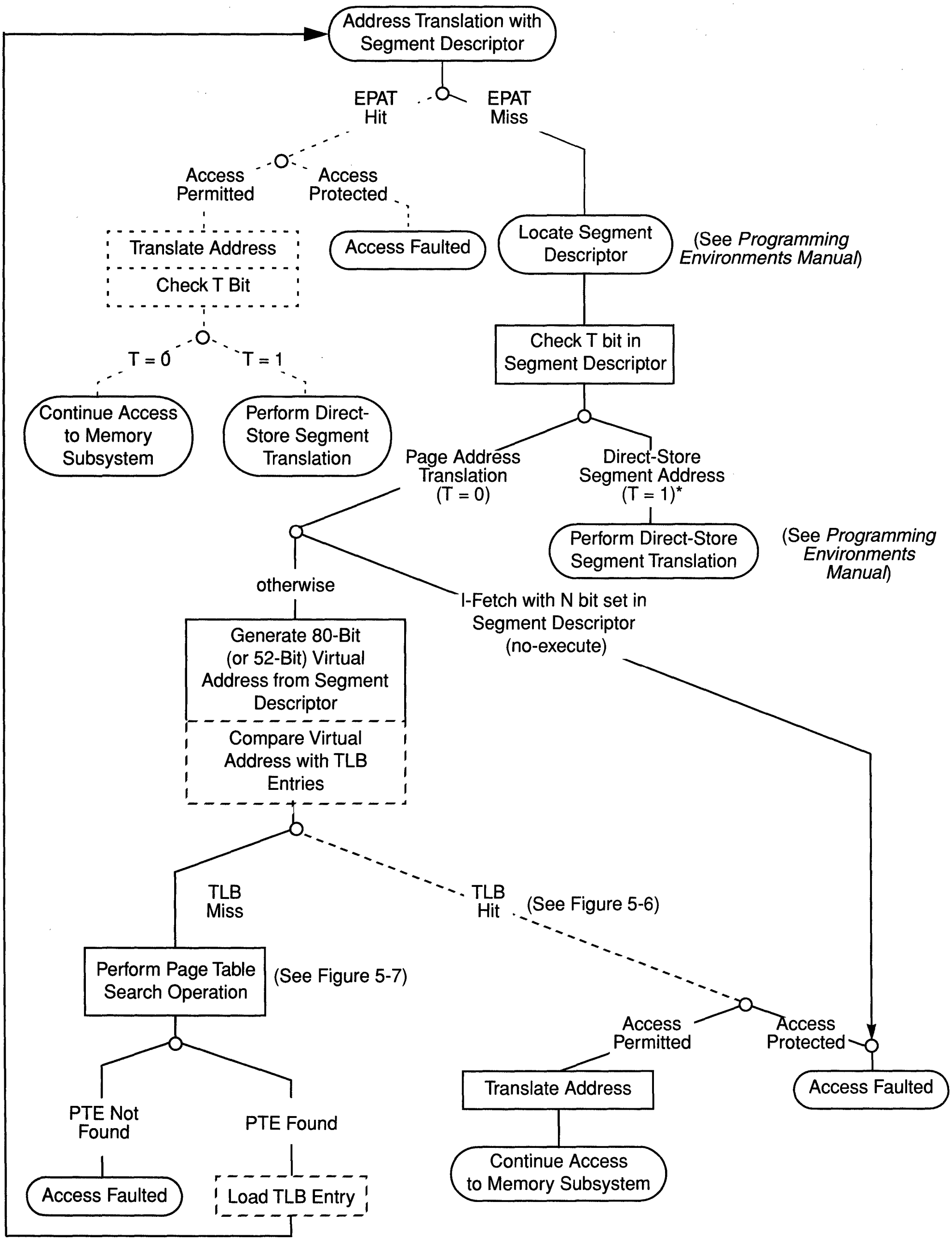
Note that if the BAT array search results in a hit, the access is qualified with the appropriate protection bits. If the access violates the protection mechanism, an exception (ISI or DSI exception) is generated.

Implementation Note—In real mode, 620 load and store operations are treated as though they are marked as guarded, cacheable, writeback, and memory coherent (WIMG = 0011). Instruction fetch operations are treated as guarded, cacheable, and memory coherent (WIMG = 0011). However, these base WIMG values can be modified by configuration specified in both HID0 and BUSCSR. Note however, that the 620 ignores the settings of the W and G bits in the IBATs.

5.1.6.2 Page and Direct-Store Interface Address Translation Selection

When an effective address is generated, and address translation is enabled, the effective address is checked against both the BATs and the EPATs. If the address translation is not present in the BATs and it is found in the EPATs, the address translation, WIMG settings, protection information, and whether the access is to a page in memory or is a direct-store access is determined immediately and the memory access can proceed.

However, if the effective address translation is in neither the BATs or the EPATs, the segment descriptor must be located. The T bit in the segment descriptor selects whether the translation is to a page or to a direct-store segment as shown in Figure 5-5. In addition, Figure 5-5 also shows how no-execute protection is enforced; if the N bit in the segment descriptor is set and the access is an instruction fetch, the access is faulted as described in Chapter 7, "Memory Management," in *The Programming Environments Manual.* Note that the figure shows the flow for these cases as described by the PowerPC OEA, and so the TLB references are shown as optional. As the 620 implements TLBs, these branches are valid. TLBs are described in more detail throughout this chapter.


Address Translation with Segment Descriptor
EPAT Hit
EPAT Miss
Access Permitted
Access Protected
Translate Address
Check T Bit
Access Faulted
Locate Segment Descriptor
(See *Programming Environments Manual*)
T = 0
T = 1
Continue Access to Memory Subsystem
Perform Direct-Store Segment Translation
Check T bit in Segment Descriptor
Page Address Translation (T = 0)
Direct-Store Segment Address (T = 1)*
Perform Direct-Store Segment Translation
(See *Programming Environments Manual*)
otherwise
I-Fetch with N bit set in Segment Descriptor (no-execute)
Generate 80-Bit (or 52-Bit) Virtual Address from Segment Descriptor
Compare Virtual Address with TLB Entries
TLB Miss
TLB Hit
(See Figure 5-6)
Perform Page Table Search Operation
(See Figure 5-7)
Access Permitted
Access Protected
Translate Address
Access Faulted
Continue Access to Memory Subsystem
PTE Not Found
PTE Found
Access Faulted
Load TLB Entry

Notes:
* Not allowed for instruction accesses (causes ISI exception)

- - - Implementation-specific

Figure 5-5. General Flow of Page and Direct-Store Interface Address Translation

5.1.6.3 Selection of Page Address Translation

If the effective address hits in the EPAT, the EPAT provides the physical address translation (the value of the physical page number, PTE[RPN]) required for generating the effective-to-physical address translation. The EPAT saves information about the translation that indicates whether it is a page address or a direct-store access.

If the EA is not in the EPATs, the segment descriptor is located. If the T bit in the corresponding segment descriptor is 0, page address translation is selected. Otherwise, the information in the segment descriptor is then used to generate the 80-bit virtual address. The virtual address is then used to identify the page address translation information (stored as page table entries (PTEs) in a page table in memory). For increased performance, the 620 has a unified TLB in the second-level MMU that stores recently-used PTEs on-chip. When the TLBs are updated, copies of the page translation information are forwarded to the EPATs.

If an access hits in the appropriate TLB, the page translation occurs and the physical address bits are forwarded to the memory subsystem. If the required PTE is not resident, the MMU must search the page table. In this case, logic in the 620's second-level MMU directs the page table search operation. If the PTE is found, a new TLB and EPAT entry is created and the page translation is once again attempted. This time, the EPAT is guaranteed to hit. Once the PTE is located, the access is qualified with the appropriate protection bits. If the access is a protection violation (not allowed), either an ISI or DSI exception is generated.

If the PTE is not found by the table search operation, a page fault condition exists, and an ISI or DSI exception occurs so software can handle the page fault.

5.1.6.4 Selection of Direct-Store Interface Address Translation

All effective addresses are checked against the content of the EPATs in the first-level IMMU and DMMU. EPAT entries contain information that indicates whether the translation is for a page address or a direct-store access.

If the translation is not provided in the EPAT, the segment descriptor must be located. When the segment descriptor has the T bit set, the access is considered a direct-store interface access and the direct-store interface protocol of the external interface is used to perform the access to direct-store space. The selection of address translation type differs for instruction and data accesses only in that instruction accesses are not allowed from direct-store segments; attempting to fetch an instruction from a direct-store segment causes an ISI exception. See Section 5.6, "Direct-Store Interface Address Translation," for more detailed information about the translation of addresses in direct-store space.

5.1.7 MMU Exceptions Summary

In order to complete any memory access, the effective address must be translated to a physical address. As specified by the architecture, an MMU exception condition occurs if this translation fails for one of the following reasons:

- There is no valid entry in the page table for the page specified by the effective address (and segment descriptor) and there is no valid BAT translation.
- There is no valid segment descriptor and there is no valid BAT translation.
- An address translation is found but the access is not allowed by the memory protection mechanism.

The translation exception conditions defined by the OEA for 32-bit implementations cause either the ISI or the DSI exception to be taken as shown in Table 5-5.

The translation exception conditions cause either the ISI or the DSI exception to be taken as shown in Table 5-4. The state saved by the processor for each of these exceptions contains information that identifies the address of the failing instruction. Refer to Chapter 4, "Exceptions," for a more detailed description of exception processing, and the bit settings of SRR1 and DSISR when an exception occurs.

Table 5-4. Translation Exception Conditions

Condition	Description	Exception
Page fault (no PTE found)	No matching PTE found in page tables (and no matching BAT array entry)	I access: ISI exception SRR1[33] = 1
		D access: DSI exception DSISR[1] = 1
Segment table fault (no STE found)	No matching STE found in the segment tables (for 64-bit implementations) and no matching BAT array entry	I access: ISI exception SRR1[42] = 1
		D access: DSI exception DSISR[10] =1
Block protection violation	Conditions described in *Programming Environments Manual*	I access: ISI exception SRR1[36] = 1
		D access: DSI exception DSISR[4] = 1
Page protection violation	Conditions described in *Programming Environments Manual*	I access: ISI exception SRR1[36] = 1
		D access: DSI exception DSISR[4] = 1

Table 5-4. Translation Exception Conditions (Continued)

Condition	Description	Exception
No-execute protection violation	Attempt to fetch instruction when or STE[N] = 1	ISI exception SRR1[35] = 1
Instruction fetch from direct-store segment	Attempt to fetch instruction when SR[T] = 1 or STE[T] = 1	ISI exception SRR1[35] = 1
Instruction fetch from guarded memory	Attempt to fetch instruction when MSR[IR] = 1 and either: matching xBAT[G] = 1, or no matching BAT entry and PTE[G] = 1	ISI exception SRR1[35] = 1

In addition to the translation exceptions, there are other MMU-related conditions (some of them implementation-specific) that can cause an exception to occur. These conditions map to the exceptions as shown in Table 5-5. The only MMU exception conditions that occur when MSR[DR] = 0 are the conditions that cause the alignment exception for data accesses. For more detailed information about the conditions that cause the alignment exception (in particular for string/multiple instructions), see Section 4.6.6, "Alignment Exception (0x00600)." Refer to Chapter 4, "Exceptions," for a complete description of the SRR1 and DSISR bit settings for these exceptions.

Table 5-5. Other MMU Exception Conditions

Condition	Description	Exception
dcbz with W = 1 or I = 1 (may cause exception or operation may be performed to memory)	**dcbz** instruction to *write-through* or cache-inhibited segment or block	Alignment exception (Optional in the PowerPC architecture)
ldarx, **stdcx.**, **lwarx**, or **stwcx.** with W = 1 (may cause exception or execute correctly)	*Reservation* instruction to write-through segment or block	DSI exception (implementation-dependent) DSISR[5] = 1
ldarx, **stdcx.**, **lwarx**, **stwcx.**, **eciwx**, or **ecowx** instruction to direct-store segment (may cause exception or may produce boundedly-undefined results)	Reservation instruction or external control instruction when SR[T] = 1 or STE[T] = 1	DSI exception (implementation-dependent) DSISR[5] = 1
Floating-point load or store to direct-store segment (may cause exception or instruction may execute correctly)	Floating-point memory access when SR[T] = 1 or STE[T] = 1	Alignment exception (implementation-dependent)
Load or store operation that causes a direct-store error	Direct-store interface protocol signalled with an error condition	DSI exception DSISR[0] = 1
eciwx or **ecowx** attempted when external control facility disabled	**eciwx** or **ecowx** attempted with EAR[E] = 0	DSI exception DSISR[11] = 1
lmw, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** instruction attempted in *little-endian* mode	**lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** instruction attempted while MSR[LE] = 1	Alignment exception
Operand misalignment	Translation enabled and operand is misaligned as described in Chapter 4, "Exceptions."	Alignment exception (some of these cases are implementation-dependent).

5.1.8 MMU Instructions and Register Summary

The MMU instructions and registers provide the operating system with the ability to set up the segment descriptors. Additionally, the operating system has the resources to set up the block address translation areas and the page tables in memory.

5.1.8.1 MMU Registers

Table 5-6 summarizes the registers that the operating system uses to program the MMU. These registers are accessible to supervisor-level software only (supervisor level is referred to as privileged state in the architecture specification). These registers are described in detail in Chapter 2, "Programming Model."

Table 5-6. MMU Registers

Register	Description
Segment registers (SR0–SR15) (64-bit bridge only)	The sixteen 32-bit segment registers are defined for 32-bit implementations of the PowerPC architecture, but are emulated by the first 16 SLBs when the 620 enables the 64-bit bridge. The fields in the segment register are interpreted differently depending on the value of bit 0. The segment registers are accessed by the **mtsr**, **mtsrin**, **mfsr**, and **mfsrin** instructions.
BAT registers (IBAT0U–IBAT3U, IBAT0L–IBAT3L, DBAT0U–DBAT3U, and DBAT0L–DBAT3L)	There are 16 BAT registers, organized as four pairs of instruction BAT registers (IBAT0U–IBAT3U paired with IBAT0L–IBAT3L) and four pairs of data BAT registers (DBAT0U–DBAT3U paired with DBAT0L–DBAT3L). The BAT registers are defined as 64-bit for 64-bit implementations. These special-purpose registers are accessed by the **mtspr** and **mfspr** instructions.
SDR1 register	The SDR1 register specifies the base and size of the page tables in memory. SDR1 is a 64-bit register for 64-bit implementations. This is a special-purpose register that is accessed by the **mtspr** and **mfspr** instructions.
Address space register (ASR)	The 64-bit ASR specifies the physical address in memory of the segment table for 64-bit implementations. This is a special-purpose register that is accessed by the **mtspr** and **mfspr** instructions. Because it implements the 64-bit bridge, the 620 defines ASR[63] as a valid bit that specifies whether the access uses 32- or 64-bit addressing. The ASR is described in more detail in Section 5.1.8.2, "Address Space Register (ASR) and the 64-Bit Bridge."

5.1.8.2 Address Space Register (ASR) and the 64-Bit Bridge

The OEA defines an additional, optional bridge to the 64-bit architecture that allows 64-bit implementations to retain certain aspects of the 32-bit architecture that otherwise are not supported, and in some cases not permitted by the 64-bit architecture. The bridge facilities allow the option of defining bit 63 as ASR[V], the STABORG field valid bit. If this bit is implemented, STABORG is valid only when ASR[V] is set. This bit is optional, but is implemented if any of the following instructions, which are optional to a 64-bit processor, are implemented: **mtsr**, **mtsrin**, **mfsr**, **mfsrin**, **mtsrd**, or **mtsrdin**. Processors that do not implement ASR[V] treat ASR[63] as reserved except that it is assumed to be 1 for address translation.

The following further describes programming considerations that are affected by the ASR[V] bit:

- If ASR[V] is cleared, having the STABORG field refer to a nonexistent memory location does not cause a machine check exception. Also, if ASR[V] is cleared, the segment table in memory is not searched and the result is the same as if the search had failed.
- For a 64-bit operating system that uses the segment register manipulation instructions as if it were running on a 32-bit implementation: if ASR[V] = 0, a segment fault can occur only if the operating system contains a bug that allows the generation of an effective address larger than $2^{32}- 1$ when MSR[SF] = 1 or if the operating system fails to ensure that the first 16 ESIDs are established (that is, that the corresponding SLB entries are valid)
- Note that **slbie** or **slbia** can be executed regardless of the setting of ASR[V]; however, the instructions should not be used if ASR[V] is cleared.

If ASR[V] is implemented, the ASR must point to a valid segment table whenever address translation is enabled, the effective address is not covered by BAT translation, and ASR[V] = 1.

5.1.8.3 MMU Instructions

Because the implementation of TLBs and SLBs is optional, the instructions that refer to these structures are also optional. However, as these structures serve as caches of the page table (and segment table, in the case of an SLB), there must be a software protocol for maintaining coherency between these caches and the tables in memory whenever changes are made to the tables in memory. Therefore, the PowerPC OEA specifies that a processor implementing a TLB is guaranteed to have a means for doing the following:

- Invalidating an individual TLB entry (supported by the 620 through the **tlbie** instruction)
- Invalidating the entire TLB (through the architecture defined **tlbia** instruction; not supported by the 620)

Similarly, a processor that implements an SLB is guaranteed to have a means for doing the following:

- Invalidating an individual SLB entry (the architecture defines an optional **slbie** instruction for this purpose)
- Invalidating the entire SLB (the architecture defines an optional **slbia** instruction for this purpose)

Note that while the implementation of SLBs in 64-bit processors is optional, processors that implement the 64-bit bridge are required to implement at least 16 SLB entries to provide a means of emulating the segment registers as they are defined in the 32-bit architecture. When the processor is using the 64-bit bridge, neither the **slbie** or **slbia** instruction should be executed.

When the tables in memory are changed, the operating system purges these caches of the corresponding entries, allowing the translation caching mechanism to refetch from the tables when the corresponding entries are required.

A processor may implement one or more of the instructions listed in this section to support table invalidation. If an instruction is implemented that matches the semantics of an instruction listed here (and described in this document), the operation will be as described. Alternatively, an algorithm may be specified that performs one of the functions listed above (a loop invalidating individual TLB entries may be used to invalidate the entire TLB, for example), or instructions with different semantics may be implemented.

A processor may also perform additional functions (not described here) as well as those described in the implementation of some of these instructions. For example, an instruction whose semantics are to purge a TLB entry may be implemented so as to purge all TLB entries in a congruence class (that is, all TLB entries indexed by the specified EA which can include corresponding entries in data and instruction TLBs) or the entire TLB.

Because the MMU specification for PowerPC processors is so flexible, it is recommended that the software that uses these instructions and registers be "encapsulated" into subroutines to minimize the impact of migrating across the family of implementations.

Table 5-7 summarizes the PowerPC instructions that specifically control the MMU. For more detailed information about the instructions, refer to the *Programming Environment's Manual*.

Table 5-7. Instruction Summary—Control MMU

Instruction	Description
mtsr SR,**r**S	Move to Segment Register SR[SR]← **r**S (32-bit instruction used with 64-bit bridge)
mtsrin **r**S,**r**B	Move to Segment Register Indirect SR[**r**B[0–3]]←**r**S (32-bit instruction used with 64-bit bridge)
mtsrd SR,**r**S	Move to Segment Register Double Word SLB[SR]← **r**S (64-bit bridge only)
mtsrdin **r**S,**r**B	Move to Segment Register Indirect Double Word SLB(**r**B[32-35]) ← (**r**S) (64-bit bridge only)
mfsr **r**D,SR	Move from Segment Register **r**D←SR[SR] (32-bit instruction used with 64-bit bridge)
mfsrin **r**D,**r**B	Move from Segment Register Indirect **r**D←SR[**r**B[0–3]] (32-bit instruction used with 64-bit bridge)

Table 5-7. Instruction Summary—Control MMU (Continued)

Instruction	Description
tlbia (not implemented)	Translation Lookaside Buffer Invalidate All—This instruction is optional to the PowerPC architecture and is not supported in the 620. Attempting to execute it causes an illegal instruction program exception.
tlbie rB	Translation Lookaside Buffer Invalidate Entry If TLB hit (for effective address specified as **r**B), TLB[V]←0 Causes EPAT and TLB invalidation of entry in all processors in system. This instruction is optional to the PowerPC architecture.
tlbsync	Translation Lookaside Buffer Synchronize Ensures that all **tlbie** instructions previously executed by the processor executing the **tlbsync** instruction have completed on all processors. This instruction is optional to the PowerPC architecture.
slbia	Segment Table Lookaside Buffer Invalidate All For all SLB entries, SLB[V]←0 Also invalidated all EPATs. 64-bit implementations only. This instruction is optional to the PowerPC architecture.
slbie rB	Segment Table Lookaside Buffer Invalidate Entry If SLB hit (for effective address specified as **r**B), SLB[V]←0 Invalidates all EPAT entries that match the specified address (ESID). This instruction is optional to the PowerPC architecture.

5.2 Real Addressing Mode

If address translation is disabled (MSR[IR] = 0 or MSR[DR] = 0) for a particular access, the effective address is treated as the physical address and is passed directly to the memory subsystem as described in Chapter 7, “Memory Management,” in *The Programming Environments Manual.*

For information on the synchronization requirements for changes to MSR[IR] and MSR[DR], refer to Section 2.3.2.4, “Synchronization.”

5.3 Block Address Translation

The 620 implements block address translation as it is defined by the PowerPC architecture. The block address translation (BAT) mechanism in the OEA provides a way to map ranges of effective addresses larger than a single page into contiguous areas of physical memory. Such areas can be used for data that is not subject to normal virtual memory handling (paging), such as a memory-mapped display buffer or an extremely large array of numerical data.

Block address translation in the 620 is described in Chapter 7, “Memory Management,” in *The Programming Environments Manual* for 32-bit implementations.

5.4 Memory Segment Model

The 620 adheres to the memory segment model as defined in Chapter 7, "Memory Management," in *The Programming Environments Manual* for 64-bit implementations. Memory in the PowerPC OEA is divided into 256-Mbyte segments. This segmented memory model provides a way to map 4-Kbyte pages of effective addresses to 4-Kbyte pages in physical memory (page address translation), while providing the programming flexibility afforded by a large virtual address space (80 bits).

The segment/page address translation mechanism may be superseded by the block address translation (BAT) mechanism described in Section 5.3, "Block Address Translation." Also if the page address translation has been saved in an EPAT in the first-level instruction or data MMU, there is no need to access the segment descriptors. Information pertaining to memory protection or whether the translation is for a direct-store access is saved in the EPAT entry. Note that the EPATs are not defined by the PowerPC architecture.

If the translation is not already available, the translation proceeds in the following two steps:

1. From effective address to the virtual address (which never exists as a specific entity but can be considered to be the concatenation of the virtual page number and the byte offset within a page), and
2. From virtual address to physical address.

This section highlights those areas of the memory segment model defined by the OEA that are specific to the 620.

5.4.1 Page History Recording

Referenced (R) and changed (C) bits reside in each PTE to keep history information about the page. They are maintained by a combination of the 620 table search hardware and the system software. The operating system uses this information to determine which areas of memory to write back to disk when new pages must be allocated in main memory. Referenced and changed recording is performed only for accesses made with page address translation and not for translations made with the BAT mechanism or for accesses that correspond to direct-store (T = 1) segments. Furthermore, R and C bits are maintained only for accesses made while address translation is enabled (MSR[IR] = 1 or MSR[DR] = 1).

In the 620, the referenced and changed bits are updated as follows:

- For TLB hits, the C bit is updated according to Table 5-8.
- For TLB misses, when a table search operation is in progress to locate a PTE. The R and C bits are updated (set, if required) to reflect the status of the page based on this access.

Table 5-8 shows that the status of the C bit in the TLB entry (in the case of a TLB hit) is what causes the processor to update the C bit in the PTE (the R bit is assumed to be set in the page tables if there is a TLB hit). Therefore, when software clears the R and C bits in the page tables in memory, it must invalidate the TLB entries associated with the pages whose referenced and changed bits were cleared.

Table 5-8. Table Search Operations to Update History Bits—TLB Hit Case

R and C bits in TLB Entry	Processor Action
00	Combination doesn't occur
01	Combination doesn't occur
10	Read: No special action Write: The 620 initiates a table search operation to update C.
11	No special action for read or write

The **dcbt** and **dcbtst** instructions can execute if there is a TLB/BAT hit or if the processor is in real addressing mode. In case of a TLB/BAT miss, these instructions are treated as no-ops; they do not initiate a table search operation and they do not set either the R or C bits.

As defined by the PowerPC architecture, the referenced and changed bits are updated as if address translation were disabled (real addressing mode). Additionally, these updates are performed with single-beat read and byte write transactions on the bus.

5.4.1.1 Referenced Bit

The referenced (R) bit of a page is located in the PTE in the page table. Every time a page is referenced (with a read or write access) and the R bit is zero, the 620 sets the R bit in the page table. The OEA specifies that the referenced bit may be set immediately, or the setting may be delayed until the memory access is determined to be successful. Because the reference to a page is what causes a PTE to be loaded into the TLB, the referenced bit in all TLB entries is effectively always set. The processor never automatically clears the referenced bit.

The referenced bit is only a hint to the operating system about the activity of a page. At times, the referenced bit may be set although the access was not logically required by the program or even if the access was prevented by memory protection. Examples of this in PowerPC systems include the following:

- Fetching of instructions not subsequently executed
- Accesses generated by an **lswx** or **stswx** instruction with a zero length
- Accesses generated by an **stwcx.** or **stdcx.** instruction when no store is performed because a reservation does not exist
- Accesses that cause exceptions and are not completed

5.4.1.2 Changed Bit

The changed bit of a page is located both in the PTE in the page table and in the copy of the PTE loaded into the TLB (if a TLB is implemented, as in the 620). Whenever a data store instruction is executed successfully, if the TLB search (for page address translation) results in a hit, the changed bit in the matching TLB entry is checked. If it is already set, the processor does not change the C bit. If the TLB changed bit is 0, the 620 sets it and a table search operation is performed to also set the C bit in the corresponding PTE in the page table. The 620 initiates the table search operation for setting the C bit in this case.

The changed bit (in both the TLB and the PTE in the page tables) is set only when a store operation is allowed by the page memory protection mechanism and the store is guaranteed to be in the execution path (unless an exception, other than those caused by the **sc**, **rfi**, or trap instructions, occurs). Furthermore, the following conditions may cause the C bit to be set:

- The execution of an **stwcx.** or **stdcx.** instruction is allowed by the memory protection mechanism but a store operation is not performed.
- The execution of an **stswx** instruction is allowed by the memory protection mechanism but a store operation is not performed because the specified length is zero.
- The store operation is not performed because an exception occurs before the store is performed.

Again, note that although the execution of the **dcbt** and **dcbtst** instructions may cause the R bit to be set in some PowerPC processors, they never cause the C bit to be set.

5.4.1.3 Scenarios for Referenced and Changed Bit Recording

This section provides a summary of the model (defined by the OEA) that is used by PowerPC processors for maintaining the referenced and changed bits. In some scenarios, the bits are guaranteed to be set by the processor, in some scenarios, the architecture allows that the bits may be set (not absolutely required), and in some scenarios, the bits are guaranteed to not be set. Note that when the 620 updates the R and C bits in memory, the accesses are performed as if MSR[DR] = 0 and G = 0 (that is, as nonguarded cacheable operations in which coherency is required).

Table 5-9 defines a prioritized list of the R and C bit settings for all scenarios. The entries in the table are prioritized from top to bottom, such that a matching scenario occurring closer to the top of the table takes precedence over a matching scenario closer to the bottom of the table. For example, if an **stwcx.** instruction causes a protection violation and there is no reservation, the C bit is not altered, as shown for the protection violation case. Note that in the table, load operations include those generated by load instructions, by the **eciwx** instruction, and by the cache management instructions that are treated as a load with respect to address translation. Similarly, store operations include those operations generated by store instructions, by the **ecowx** instruction, and by the cache management instructions that are treated as a store with respect to address translation.

Table 5-9. Model for Guaranteed R and C Bit Settings

Priority	Scenario	Causes Setting of R Bit		Causes Setting of C Bit	
		OEA	620	OEA	620
1	No-execute protection violation	No	No	No	No
2	Page protection violation	Maybe	No	No	No
3	Out-of-order instruction fetch or load operation	Maybe	No	No	No
4	Out-of-order store operation contingent on a branch, trap, **sc** or **rfi** instruction, or a possible exception	Maybe	No	No	No
5	Out-of-order store operation contingent on an exception, other than a trap or **sc** instruction, not occurring	Maybe	No	No	No
6	Zero-length load (**lswx**)	Maybe		No	No
7	Zero-length store (**stswx**)	Maybe[1]		Maybe[1]	
8	Store conditional (**stwcx.** or **stdcx.**) that does not store	Maybe[1]		Maybe[1]	
9	In-order instruction fetch	Yes[2]	Yes	No	No
10	Load instruction or **eciwx**	Yes	Yes	No	No
11	Store instruction, **ecowx**, or **dcbz** instruction	Yes	Yes	Yes	Yes
12	**icbi**, **dcbt**, **dcbtst**, **dcbst**, or **dcbf** instruction	Maybe	No	No	No
13	**dcbi** instruction	Maybe[1]	No	Maybe[1]	No

[1] If C is set, R is also guaranteed to be set.

[2] This includes the case in which the instruction was fetched out-of order and R was not set (does not apply for 620).

For more information, see "Page History Recording" in Chapter 7, "Memory Management," of *The Programming Environments Manual.*

5.4.2 Page Memory Protection

The 620 implements page memory protection as it is defined in Chapter 7, "Memory Management," in *The Programming Environments Manual.*

5.4.3 SLB Description

The 620 implements a 20-entry, FIFO segment-lookaside buffer (SLB). The SLB implementation supports both 64-bit addressing as defined by the PowerPC architecture and 32-bit addressing which is supported through the 64-bit bridge defined as optional by the PowerPC architecture.

When the processor is in 64-bit bridge mode, the first 16 entries of the SLB act like 64-bit segment register. The remaining four SLB entries function as SLB entries as they are defined for 64-bit addressing.

The SLB supports simultaneous updates from the 64-bit bridge's implementation of the **mtsr** instruction as well as automatic hardware refills provided for 64-bit addressing, and as such, allows complex combinations of translations based on STEs and segment registers. In these hybrid modes, software must maintain SLB consistency.

5.4.4 SLB Invalidation

The 620 supports the **slbie** and the **slbia** instructions defined by the PowerPC architecture. Note that the instructions invalidate the SLB entries as well as the EPAT entries that match the specified ESID. These instructions are not broadcast on the bus.

5.4.5 TLB Description

The UTLB contains 128 entries organized as a two-way set associative array with 64 sets. If the address in one of the two TLB entries is valid and matches the virtual address, that TLB entry contains the physical address. If no match is found, a TLB miss occurs.

Unless the access is the result of an out-of-order access, a hardware table search operation begins if there is a TLB miss. If the access is out of order, the table search operation is postponed until the access is required, at which point the access is no longer out of order. When the matching PTE is found in memory, it is loaded into a particular TLB entry selected by the least-recently-used (LRU) replacement algorithm and the appropriate EPAT, and the translation process begins again, this time with an EPAT hit.

TLB entries are on-chip copies of PTEs in the page tables in memory and are similar in structure. Formats for the PTE are given in "PTE Format for 64-Bit Implementations," in Chapter 7, "Memory Management," of *The Programming Environments Manual.*

Software cannot access TLB arrays directly, except to invalidate an entry with the **tlbie** instruction.

Each set of TLB entries is associated with one LRU bit, which is accessed when those entries in the same set are indexed. LRU bits are updated whenever a TLB entry is used or after the entry is replaced. Invalid entries are always the first to be replaced.

Although both SLBs and TLBs can be accessed in the same clock, only one exception is reported at a time.

Although address translation is disabled on a reset condition, the valid bits of the BAT array and TLB entries are not automatically cleared. Thus, TLB entries must be explicitly cleared by the system software (with the **tlbie** instruction) before the valid entries are loaded and address translation is enabled.

5.4.6 TLB Invalidation

For PowerPC processors such as the 620 that implement TLB structures to maintain on-chip copies of the PTEs that are resident in physical memory, the optional TLB Invalidate Entry (**tlbie**) instruction provides a way to invalidate the TLB entries. Note that the **tlbia** instruction is not implemented by the 620.

When a **tlbie** instruction is executed, the following actions occur:

- Processor Function—The processor invalidates its own instruction and data TLBs and EPATs. The **tlbie** invalidates all members of the congruence class indexed by the EA. No SLB lookup and tag compare is required for this instruction. The processor issues the **tlbie** bus operation.
- Finished and Complete—The **tlbie** instruction is finished and complete when the LSU sends the **tlbie** bus operation to the bus unit.
- Performed—The **tlbie** instruction is performed when the **tlbsync** instruction is completes its operations on the bus.
- Snooper Function—The snooper indexes into both instruction and data EPATs and into TLBs, and it invalidates the entire congruence class in each TLB without comparing the tags for the TLBs.
- The 620 snooper handles only one **tlbie** instruction at a time—For more information see Section 8.4.18, "ASTATIN and ARESPIN Retry."
- Memory function—No operation

5.4.7 TLB Synchronization

The **tlbsync** instruction guarantees that all loads and stores in all processors that may have used a TLB entry that has been invalidated by a **tlbie** instruction have been performed. The **tlbsync** instruction may only be issued by one processor at a time; this must be software controlled.

Note that as a **tlbsync** master or snooper, the 620 does not guarantee that all previous **tlbie** instructions have been issued to the bus or, if completed on the bus, have been completed by the 620 snooper. To guarantee that all **tlbie** instructions executed prior to the **tlbsync** will have been completed on the bus and completed by all 620 snoopers before the **tlbsync** is issued to the bus, the **tlbsync** instruction must follow a **sync**. The completion of the **tlbsync** bus operation guarantees that all instructions on other processors that may have used a translation invalidated by a **tlbie** are complete.

When the processor issues a **tlbsync** bus operation, ARESPIN Retry causes the **tlbsync** bus operation to be reissued. ARESPIN ReRun causes the **tlbsync** operation to be issued R = 1. ARESPIN Null is treated as the completion of the **tlbsync** bus operation. ARESPIN Shared and Modified are undefined. For more information, see Section 8.4.18, "ASTATIN and ARESPIN Retry."

The **tlbsync** instruction is finished and complete when the **tlbsync** bus operation is complete.

The snooper ensures that all loads, stores, and instruction fetches that used any TLBs have been either flushed or performed. A snooped **tlbsync** has the same effect on a processor that a **sync** would have if it were executed on that processor. A snooper receives the **tlbsync** and accepts it only if ASTATIN is not Retry. If ASTATIN is Retry, the snooper must back out of the **tlbsync** operation. A snooper may continue the **tlbsync** bus operation if ARESPIN is Retry. A snooper issues ARESPOUT ReRun for as long as it takes to complete the **tlbsync.**

The **tlbsync** operation appears on the bus as a distinct operation, that causes synchronization of snooped **tlbie** instructions. Section 5.4.6, "TLB Invalidation," describes the TLB invalidation mechanisms in the 620.

5.4.8 Page Address Translation Summary

Figure 5-6 provides the detailed flow for the page address translation mechanism, it includes the checking of the N bit in the segment descriptor and then expands on the "TLB Hit" branch of Figure 5-7. The detailed flow for the "TLB Miss" branch of Figure 5-7 is described in Section 5.4.9, "Page Table Search Operation." Note that, as in the case of block address translation, if the **dcbz** instruction is attempted to be executed either in write-through mode or as cache-inhibited (W = 1 or I = 1), the alignment exception is generated. The checking of memory protection violation conditions for page address translation is described in Chapter 7, "Memory Management," in *The Programming Environments Manual.*

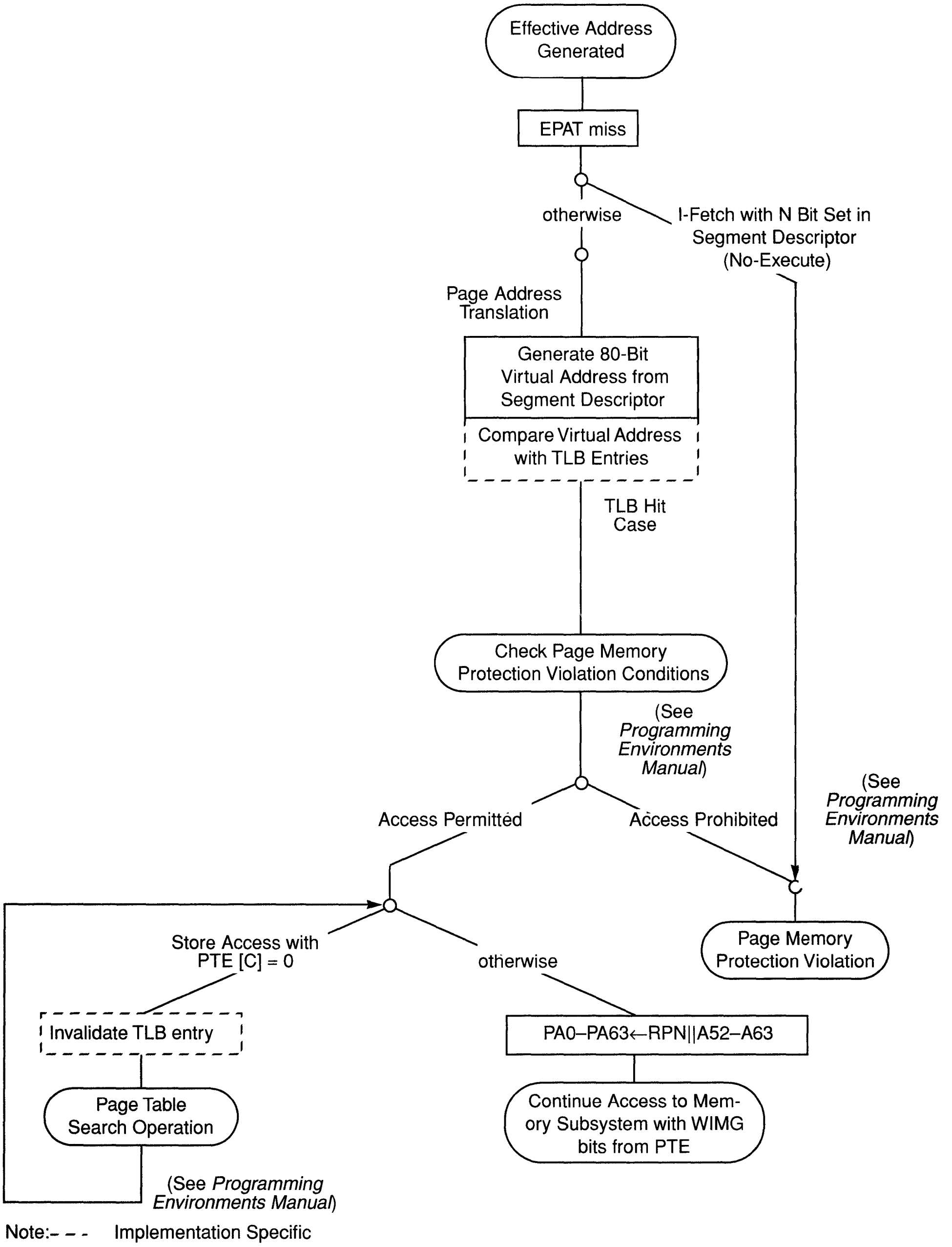

Effective Address Generated
EPAT miss
otherwise
I-Fetch with N Bit Set in Segment Descriptor (No-Execute)
Page Address Translation
Generate 80-Bit Virtual Address from Segment Descriptor
Compare Virtual Address with TLB Entries
TLB Hit Case
Check Page Memory Protection Violation Conditions
(See *Programming Environments Manual*)
(See *Programming Environments Manual*)
Access Permitted
Access Prohibited
Store Access with PTE [C] = 0
otherwise
Page Memory Protection Violation
Invalidate TLB entry
PA0–PA63←RPN||A52–A63
Page Table Search Operation
Continue Access to Memory Subsystem with WIMG bits from PTE
(See *Programming Environments Manual*)
Note:- - - Implementation Specific

Figure 5-6. Page Address Translation Flow—TLB Hit

5.4.9 Page Table Search Operation

If the translation is not found in the TLBs (a TLB miss), the 620 initiates a table search operation which is described in this section. Formats for the PTE are given in "PTE Format for 32-Bit Implementations," in Chapter 7, "Memory Management," of *The Programming Environments Manual.*

The following is a summary of the page table search process performed by the 620:

1. The 64-bit physical addresses of the primary and secondary PTEGs are generated as described in the *Programming Environments Manual.*
2. As many as 16 PTEs (from the primary and secondary PTEGs) are read from memory (the architecture does not specify the order of these reads, allowing multiple reads to occur in parallel). PTE reads occur with an implied WIM memory/cache mode control bit setting of 0b001. Therefore, they are considered cacheable.
3. The PTEs in the selected PTEGs are tested for a match with the virtual page number (VPN) of the access. The VPN is the VSID concatenated with the page index field of the virtual address. For a match to occur, the following must be true:
   - PTE[H] = 0 for primary PTEG; PTE[H] = 1 for secondary PTEG
   - PTE[V] = 1
   - PTE[VSID] = VA[0-51]
   - PTE[API] = VA[52-56]
4. If a match is not found within the eight PTEs of the primary PTEG and the eight PTEs of the secondary PTEG, an exception is generated as described in step 8. If a match (or multiple matches) is found, the table search process continues.
5. If multiple matches are found, all of the following must be true:
   - PTE[RPN] is equal for all matching entries
   - PTE[WIMG] is equal for all matching entries
   - PTE[PP] is equal for all matching entries
6. If one of the fields in step 5 does not match, the translation is undefined, and R and C bit of matching entries are undefined. Otherwise, the R and C bits are updated based on one of the matching entries.
7. A copy of the PTE is written into the on-chip TLB and appropriate EPAT, and the R bit is updated in the PTE in memory (if necessary). If there is no memory protection violation, the C bit is also updated in memory (if necessary) and the table search is complete.
8. If a match is not found within the primary or secondary PTEG, the search fails, and a page fault exception condition occurs (either an ISI or DSI exception).

Reads from memory for page table search operations are performed as if the WIMG bit settings were 0b0010 (that is, as unguarded cacheable operations in which coherency is required).

Reads from memory for table search operations should be performed as global (but not exclusive), cacheable operations, and can be loaded into the on-chip cache.

Figure 5-7 shows how the conceptual model for the primary and secondary page table search operations, described in *The Programming Environments Manual* are realized in the 620.

Figure 5-7 shows the case of a **dcbz** instruction that is executed with W = 1 or I = 1, and that the R bit may be updated in memory (if required) before the operation is performed or the alignment exception occurs. The R bit may also be updated if memory protection is violated.

5

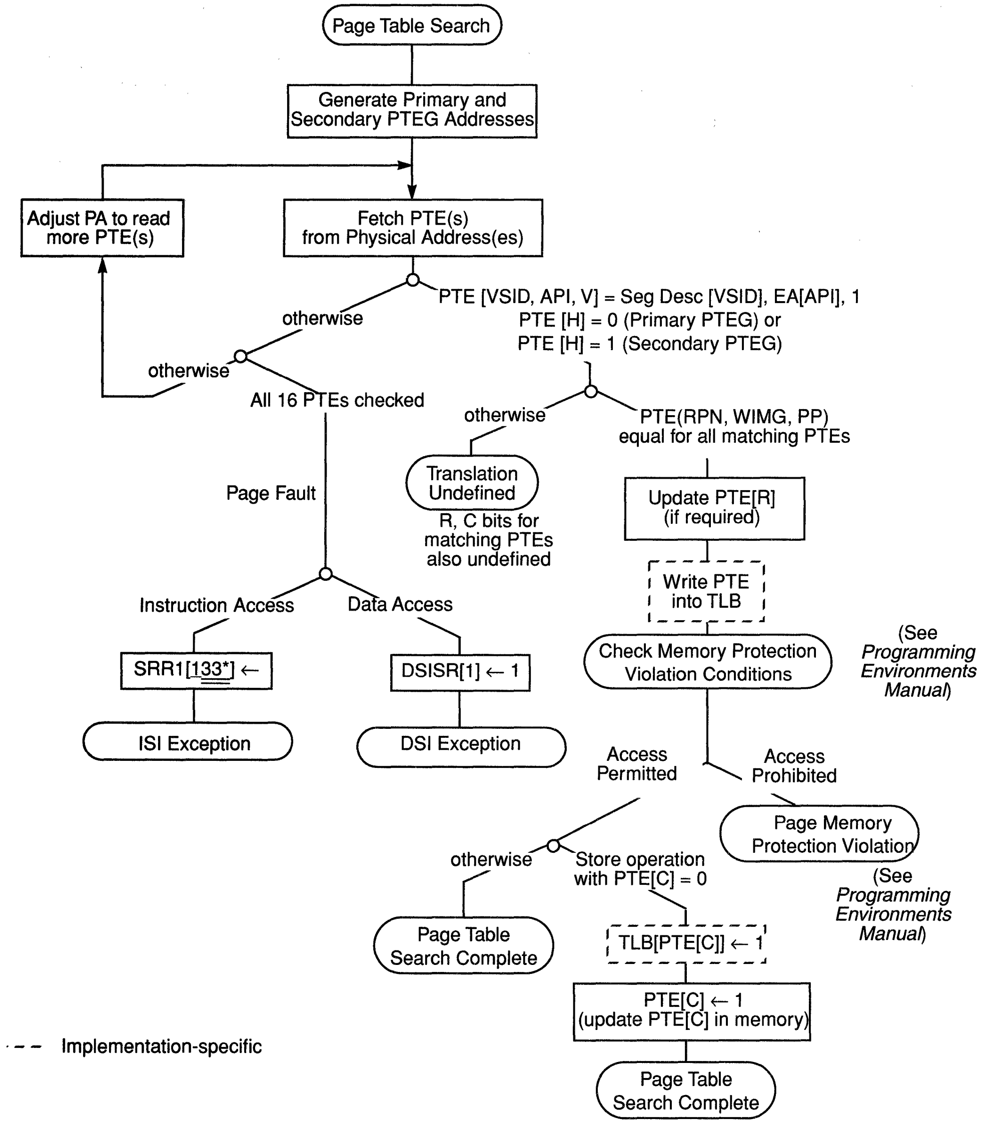

Page Table Search
Generate Primary and Secondary PTEG Addresses
Adjust PA to read more PTE(s)
Fetch PTE(s) from Physical Address(es)
PTE [VSID, API, V] = Seg Desc [VSID], EA[API], 1
PTE [H] = 0 (Primary PTEG) or
PTE [H] = 1 (Secondary PTEG)
otherwise
otherwise
All 16 PTEs checked
otherwise
PTE(RPN, WIMG, PP) equal for all matching PTEs
Page Fault
Translation Undefined
R, C bits for matching PTEs also undefined
Update PTE[R] (if required)
Write PTE into TLB
Instruction Access
Data Access
SRR1[133*] ←
DSISR[1] ← 1
Check Memory Protection Violation Conditions
(See Programming Environments Manual)
ISI Exception
DSI Exception
Access Permitted
Access Prohibited
Page Memory Protection Violation
otherwise
Store operation with PTE[C] = 0
(See Programming Environments Manual)
Page Table Search Complete
TLB[PTE[C]] ← 1
PTE[C] ← 1 (update PTE[C] in memory)
Page Table Search Complete
- - - Implementation-specific

Figure 5-7. Page Table Search

If the address in one of the two selected TLB entries is valid and matches the virtual address, that TLB entry contains the physical address. If no match is found, a TLB miss occurs and, if this is an in-order access, a hardware table search operation begins. Once the matching PTE is found in memory, it is loaded into the appropriate TLB entry depending on the LRU bit setting and translation continues.

Note that when a TLB miss occurs, the MMU does not begin the table search operation if the access is out of order.

5.5 Porting a 32-Bit Operating System

The 620 provides optional support, defined by the PowerPC architecture, that makes it easier to modify a 32-bit operating system to work in the 64-bit environment.

5.5.1 MMU Support for 32-bit OS

The primary feature added to the MMU is support for segment register emulation that resembles the segment registers defined by that OEA for 32-bit implementations. This is accomplished by a modification to the SLB that allows the **mtsr** instruction to directly load an entry into one of the first 16 entries of SLB. The entry number is determined by the instruction. A diagram of this shared structure is shown in Figure 5-8.

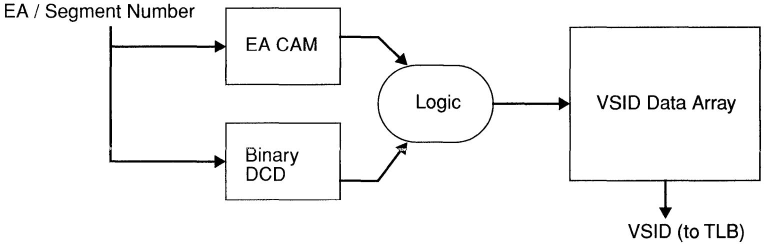


Figure 5-8. Shared Segment Registers and SLB Structure

The SLB supports both **mtsr** and normal hardware refills simultaneously, which allows complex combinations of both STE- and segment-register based translations. In these hybrid modes, the software must maintain SLB consistency. In particular, the LRU policy of the SLB becomes critical because hardware refill can overwrite the segments. The SLB internally maintains a pointer to determine which cache block is to be written on the next hardware refill. Following any write operation, the pointer is moved just below the written line. The **mtsr** instructions ignore the refill pointer and write at the offset indicated by the instruction.

In addition the following resources, defined as optional by the PowerPC architecture, are implemented on the 620.

- ASR[V]—Address space register valid bit (ASR[63]) is used on the 620 to disable segment table search operations. If this bit is cleared, any translation that misses in the SLB causes in a segment fault.
- MSR[ISF]—The MSR[ISF] is copied to MSR[SF] when the processor takes an exception.

- Segment register instructions that are required by the 32-bit architecture that are used by the 64-bit bridge, **mtsr**, **mtsrin**, **mfsr**, and **mfsrin**.
- Additional segment register double instructions that are used only by the 64-bit bridge facility:
  - **mtsrd** (Move to Segment Double)—This instruction is a 64-bit extension of the 32-bit **mtsr** instruction. The enhanced instruction allows larger virtual addresses to be loaded into the SLB.
  - **mtsrind** (Move to Segment Register Double Indirect)—This instruction is a 64-bit extension of the 32-bit **mtsrin** instruction. The enhanced instruction allows larger virtual addresses to be loaded into the SLB.

5.5.2 Guidelines

This section provides a very general overview for adapting a 32-bit operating system to run on the 620. In particular, it describes differences between a 32-bit based processor and 620 that are important to a operating system.

- PTE format—The 64-bit version of PTE has a different format from the 32-bit PTE. Sections of the operating system that create and reference PTE data must reflect the new format.
- SDR1—The 64-bit version of SDR1 has a different format from the 32-bit version. References to this register must use the new 64-bit format.
- BATs—Since the 620 updates BATs as they are defined for 64-bit addressing, software must ensure that the upper 32 bits are cleared in the GPR before issuing a **mtspr**[BAT] instruction.
- HID0, L2CR, L2SR, BUSCSR—Because most processors differ in how they use these processor configuration registers, the operating systems that access these registers will need modification.

This list cannot consider every operating system in detail, and should not be considered complete.

5.6 Direct-Store Interface Address Translation

The 620 implements the optional direct-store interface as it is defined by the PowerPC architecture. That is, if T = 1 for the selected segment descriptor and there are no BAT hits, the access maps to the direct-store interface, invoking a specific bus protocol for accessing some special-purpose I/O devices. Direct-store segments are provided for POWER compatibility and is not implemented on all PowerPC processors. As the direct-store interface is present only for compatibility with existing I/O devices that used this interface and the direct-store interface protocol is not optimized for performance, its use is discouraged. Applications that require low latency load/store access to external address space should use memory-mapped I/O, rather than the direct-store interface.

Note that the 620 implements EPATs that provide effective-to-physical address translations for recent memory accesses. The EPATs also contain information that indicates whether the T bit is set for the segment descriptor that corresponds to the memory access. Therefore whether an access is to a direct-store device can be determined without having to locate the segment descriptor.

5

Chapter 6 Instruction Timing

This chapter describes instruction prefetch and execution through all of the execution units of the PowerPC 620 microprocessor. It also provides examples of instruction sequences showing concurrent execution and various register dependencies to illustrate timing interactions.

6.1 Terminology and Conventions

Terminology and conventions used in this chapter are described as follows:

- Stage—An element in the pipeline at which certain actions are performed, such as decoding the instruction, performing an arithmetic operation, and writing back the results. A stage typically takes a cycle to perform its operation; however, some stages are repeated (a double-precision floating-point multiply, for example). When this occurs, an instruction immediately following it in the pipeline is forced to stall in its cycle.

  In some cases, an instruction may also occupy more than one stage simultaneously—for example, instructions may complete and write back their results in the same cycle.

  After an instruction is fetched, it can always be defined as being in one or more stages.

- Pipeline—In the context of instruction timing, the term pipeline refers to the interconnection of the stages. The events necessary to process an instruction are broken into several cycle-length tasks to allow work to be performed on several instructions simultaneously—analogous to an assembly line. As an instruction is processed, it passes from one stage to the next. When it does, the stage becomes available for the next instruction.

  Although an individual instruction may take many cycles to complete (the number of cycles is called instruction latency), pipelining makes it possible to overlap the processing so that the throughput (number of instructions completed per cycle) is greater than if pipelining were not implemented.

6

- Superscalar—A superscalar processor is one that can issue multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can be in the same stage at the same time. In the 620 these instructions can leave the execute stage out of order but must leave the other stages in order.
- Branch prediction—The process of guessing whether a branch will be taken. Such predictions can be correct or incorrect; the term predicted as it is used here does not imply that the prediction is correct (successful). The PowerPC architecture defines a means for static branch prediction, which is part of the instruction encoding. The 620 also implements dynamic branch prediction, where there are levels of probability assigned to a particular instruction depending on the history of that instruction, which is recorded in the branch history table (BHT).
- Branch resolution—The determination of whether a branch is taken or not taken. A branch is said to be resolved when it can exactly be determined which path it will take. If the branch is resolved as predicted, speculatively executed instructions can be completed. If the branch is not resolved as predicted, instructions on the mispredicted path are purged from the instruction pipeline and are replaced with the instructions from the nonpredicted path.
- Program order—The original order in which program instructions are provided to the instruction queue from the cache.
- Stall—An occurrence when an instruction cannot proceed to the next stage.
- Latency—The number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction. If the execution time varies dependent on the data operands, then a best-case/worst-case range is given for instructions that do not update architectural registers, such as store instructions. The execution time means the time it takes to execute this instruction back-to-back. Latency also includes the serialization penalty.
- Throughput—A measure of the number of instructions that are processed per cycle. For example, a series of double-precision floating-point multiply instructions has a throughput of one instruction per clock cycle. Throughput also includes the serialization penalty.
- Reservation station—A buffer between the dispatch and execute stages that allows instructions to be dispatched even though the operands required for execution may not yet be available. In the 620, each execution unit has a two-entry reservation station. The 620 implements two types of reservation stations. The integer units implement out-of-order execution units so integer instructions can be executed out of order within individual integer units and among the three units. The load/store unit also supports out-of-order retrieval of instructions from its reservation stations. The reservation stations of the floating-point execution unit are in-order reservation stations—that is, all instructions must pass through the floating-point unit in program order with respect to other like instructions.

- Rename buffer—Temporary buffers used by instructions that have not completed and as write-back buffers for those that have.
- Finish—The term indicates the final cycle of execution. In this cycle, the completion buffer is updated to indicate that the instruction has finished executing.
- Completion—Completion occurs when an instruction is removed from the completion buffer. When an instruction completes we can be sure that this instruction and all previous instructions will cause no exceptions. In some situations, an instruction can finish and complete in the same cycle.
- Write-back—Write-back (in the context of instruction handling) occurs when a result is written from the rename registers into the architectural registers (typically the GPRs and FPRs). Results are written back at completion time or are moved into the write-back buffer. Results in the write-back buffer cannot be flushed. If an exception occurs, these buffers must write back before the exception is taken.

6.2 Instruction Timing Overview

The 620 has been designed to maximize instruction throughput and minimize average instruction execution latency. For many of the instructions in the 620, this can be simplified to include only the execute phase for a particular instruction. Note that the number of additional cycles required by data access instructions depends on whether the access hits in the L1 cache in which case there is a single cycle required for the cache access. If the access misses in the L1 cache, the number of additional cycles required is affected by the L2 cache access latency, processor-to-bus clock ratios, and other factors pertaining to memory access.

In keeping with this definition, most integer instructions have a latency of one clock cycle (for example, results for these instructions are ready for use on the next clock cycle after issue). Other instructions, such as the integer multiply, require more than one clock cycle to finish execution.

Figure 6-1 provides a detailed block diagram—showing the additional data paths that contribute to the improved efficiency in instruction execution and more clearly shows the relationships between execution units and their associated register files.

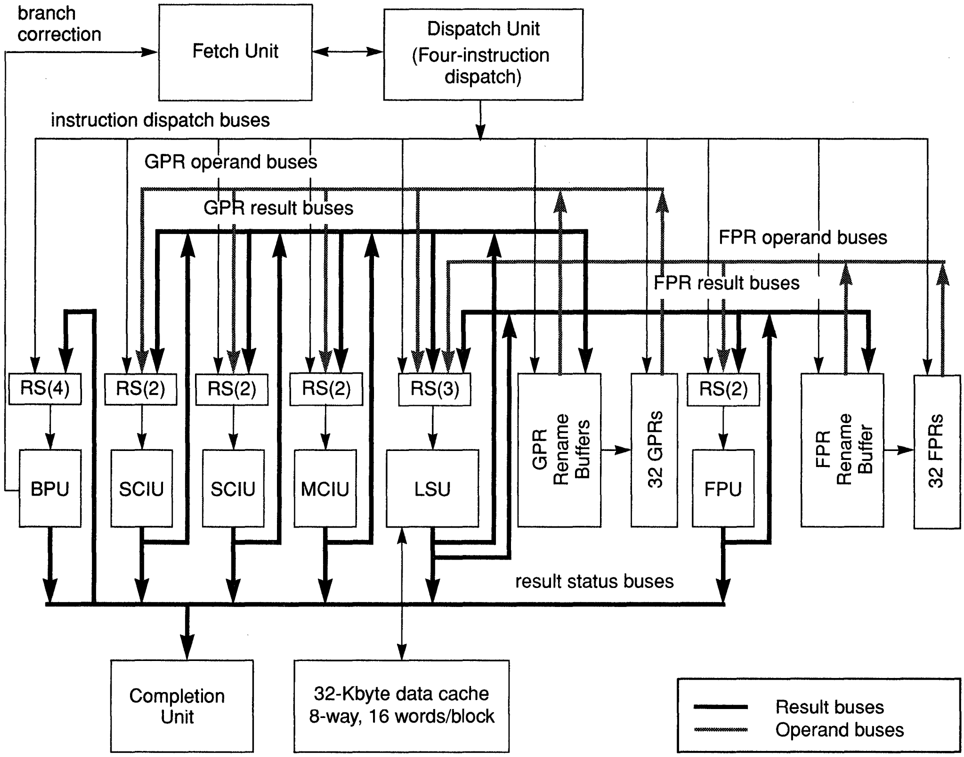


Figure 6-1. PowerPC 620 Microprocessor Block Diagram Showing Data Paths

As shown in Table 6-1, effective throughput of more than one instruction per clock cycle can be realized by the many performance features in the 620 including multiple execution units that operate independently and in parallel, pipelining, superscalar instruction issue, dynamic branch prediction, the implementation of two reservation stations for each execution unit to avoid additional latency due to stalls in individual pipelines, and result buses that forward results to dependent instructions instead of requiring those instructions to wait until results become available in the architected registers.

Although it is not shown in Figure 6-1, the LSU and FPU are pipelined.

The 620's completion buffer can retire four instructions every clock cycle. In general, instruction processing is accomplished in five stages—fetch stage, dispatch stage, execute stage, completion stage, and write-back stage. The instruction fetch stage includes the clock cycles necessary to request instructions from the on-chip cache as well as the time it takes the on-chip cache to respond to that request. In the complete stage, as many as four instructions per cycle are completed in program order. In the write-back stage, results are returned to the register file. Instructions are fetched and executed concurrently with the execution and write-back of previous instructions producing an overlap period between instructions. The details of these operations are explained in the following paragraphs.

6.2.1 Pipeline Structures

The currently proposed master instruction pipeline of the 620 has five stages. Each instruction executed by the machine will flow through at least these stages. Some instructions (for example, loads and stores) flow through additional pipeline stages as shown in Figure 6-3.

The five basic stages of the master instruction pipeline are:

- Fetch (IF)
- Dispatch (DS)
- Execute (E)
- Completion (C)
- Write-back (W)

These stages are shown in Figure 6-2. Some instructions occupy multiple stages simultaneously and some individual execution units, such as the FPU and MCIU, have multiple execution stages.

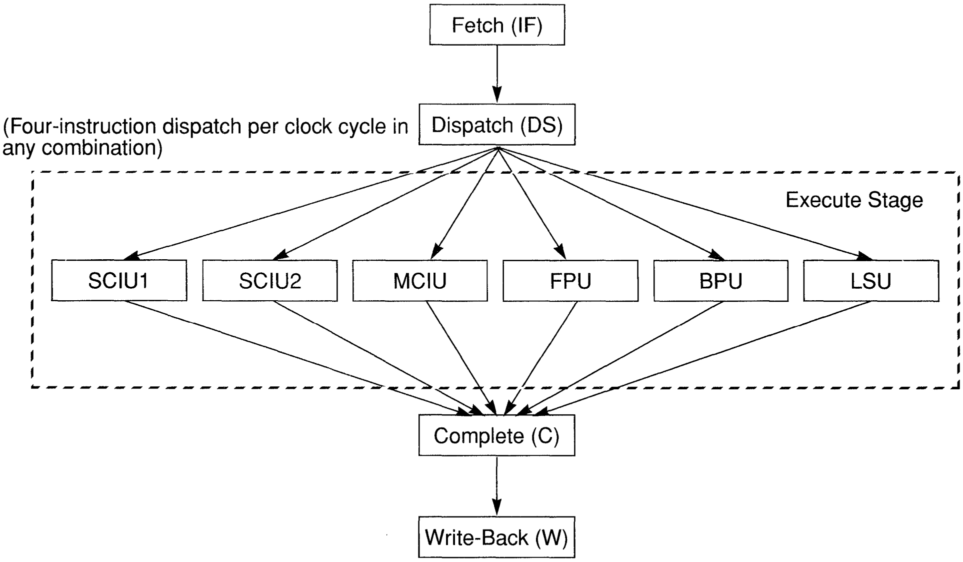
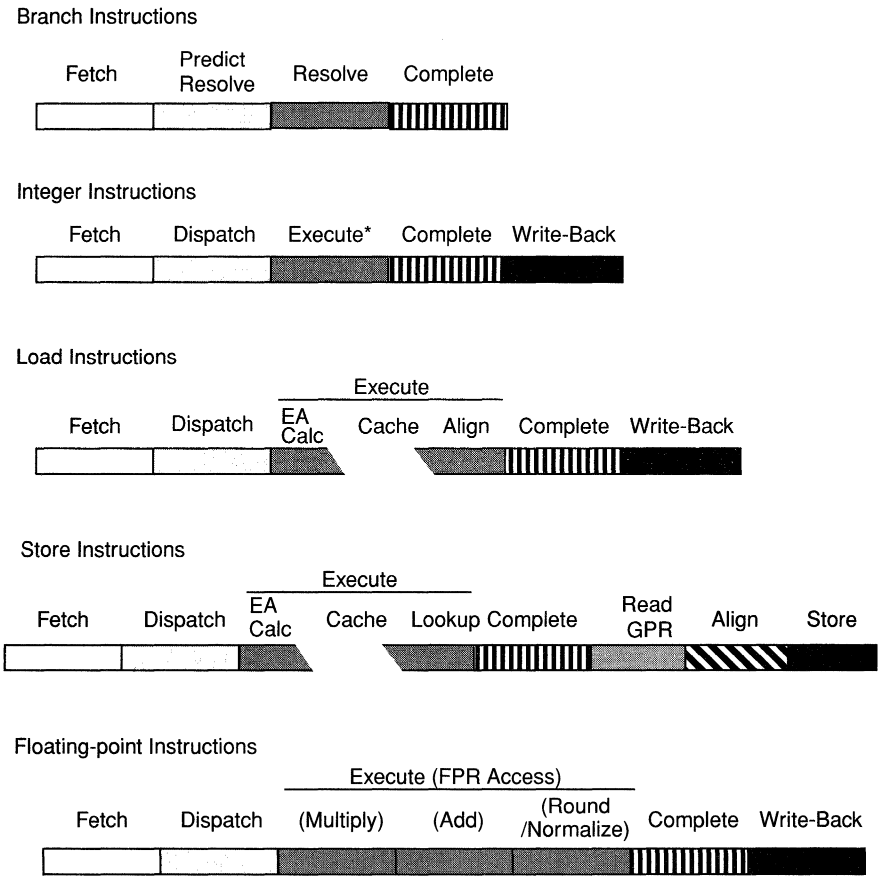


Figure 6-2. Pipeline Diagram

Pipelines for typical instructions for each of the execution units are shown in Figure 6-3. Note that this figure does not accurately reflect the latencies for all instructions that pass through each of the pipelines. The division of instructions into branch, integer, load/store, and floating-point instructions indicates the execution unit in which the instructions execute. For example, **mtspr** instructions, which are not thought of as integer instructions from a functional perspective, are considered with integer instructions here because they execute in the MCIU.

Note that in many circumstances, complete and write-back can occur in the same cycle. Also, integer multiply, integer divide, move to/from SPR, store, and load instructions that miss in the cache can occupy both the final stage of execute (finish) and complete (and write-back) simultaneously.



* Note that several integer instructions that execute in the MCIU have multiple execute stages.

Figure 6-3. Master Instruction Pipeline

A description of each of the five stages of the master instruction pipeline is provided in the following sections.

6.2.1.1 Fetch Stage

The fetch pipeline stage primarily involves accessing instructions from the instruction cache and determining where the next instruction fetch should occur. The instructions fetched from the cache are either latched into an instruction buffer or the dispatch buffer for subsequent consideration by the dispatch pipeline stage. Both branch history table (BHT) and branch target address cache (BTAC) are accessed in the fetch stage to determine where the next instruction fetch should occur.

6.2.1.2 Dispatch Stage

The dispatch pipeline stage is responsible for decoding the instructions in the dispatch buffer, and allocating execution resources to the instructions. Instructions are eligible to be dispatched if they get all of their required resources; otherwise, they are held in the dispatch buffer until the resources become available. Resources are allocated to the instructions in program order. The source operands of the instructions are read from the register file or rename buffers, and are dispatched with the instruction to the execution units. The target registers are renamed, and the rename-register tags are sent to the execution units too. At the end of the dispatch pipeline stage, the dispatched instructions and their operands are sent and latched into reservation stations or execution unit input latches.

6.2.1.3 Execute Stage

The functionality of the execute pipeline stage is executing an instruction from the reservation stations or from instructions just arriving from dispatch. An instruction becomes eligible for execution when all its source operands are available. The execution unit selects the oldest instruction to execute if there is more than one instruction ready for execution. Integer and load/store units can retrieve instructions from the reservation stations out-of-order, but branch and floating-point units can only execute the oldest instruction in the reservation stations. At the end of execute stage, execution unit will write the results into the appropriate rename buffer entry, and notify the completion stage that the instruction has finished execution. In the cases an exception occurs due to the instruction, the execution unit will report the exception to complete pipeline stage and continue executing next instructions from the reservation stations.

6.2.1.4 Complete Stage

The complete pipeline stage is responsible for maintaining the correct architectural machine state. It considers four instructions residing in the completion buffer and uses the information about the status of instructions provided by the dispatch and execute stages. If the instructions being considered meet the constraints on completion, their results are scheduled to be written back from the rename buffer(s) to the architectural register file(s). If the completion logic detects an instruction containing exception status or a branch has been mispredicted, all following instructions will be cancelled, their execution results in the rename buffers will be discarded, and the correct instruction stream will be fetched.

6.2.1.5 Write-back Stage

The write-back pipeline stage is relatively straightforward. It acts on the write-back select information generated by the complete pipeline stage and writes the appropriate Rename Buffer entries into the appropriate architectural register file(s). Updating of many other architectural registers (CTR, LK, CR, etc.) is performed at this time as well.

6.3 Instruction Scheduling Guidelines

Since instructions are dispatched in program order, the performance of the 620 can be improved by scheduling instructions appropriately to avoid resource conflicts and promote parallel utilization of execution units.

6.3.1 Instruction Dispatch Rules

The following list provides limitations on instruction dispatch that should be kept in mind in order to avoid dispatch stalls:

- At most, four instructions can be dispatched per cycle.
- Since instructions are dispatched in program order, an instruction cannot be dispatched unless all preceding instructions in the dispatch buffer are dispatched.
- One instruction can be dispatched per functional unit:
  - — The branch unit executes all branch and condition register logical instructions.
  - — The two single-cycle integer units are identical. Either can execute any integer arithmetic, logical, shift/rotate, or trap instructions. They also handle **mtcrf** instructions that update only one field.
  - — The multi-cycle integer unit executes all integer multiply, divide, and move to/from instructions (except single-field **mtcrf**). It also executes **cntlz** instructions.
  - — The load/store unit executes load, store, and cache control instructions.
  - — The floating-point unit executes all floating-point instructions, including move to/from FPSCR.
- Each instruction must have an entry in the 16-entry reorder buffer (that is, the completion buffer). The dispatch unit stalls when the reorder buffer is full. Reorder buffer entries become available in the cycle after the instruction has completed. Reorder buffer entries are assigned and released in pairs.
- An instruction that modifies a GPR is assigned at least one of the eight positions in the GPR rename buffer. Load with update instructions get two positions since they update two registers. When the GPR rename buffer is full, the dispatch unit stalls when it encounters the first instruction that needs an entry. A rename buffer entry becomes available for reassignment in the cycle that the previous results are written back to the GPR, which is always the cycle after the instruction has completed.

- Any instruction that modifies an FPR or is dispatched to the floating-point unit will be assigned one entry in the eight-entry FPR rename buffer. When the FPR rename buffer is full, dispatch will stall on the next instruction which requires an FPR rename. A rename buffer entry becomes available for reassignment one cycle after the instruction is completed.
- CR renaming is implemented in the 16-entry reorder buffer, thereby removing the possibility that an instruction will stall solely due to the lack of a CR rename.
- Each execution unit (not including branch) has a reservation station with a minimum of two entries (the load/store unit has three entries). The reservation station holds instructions until they are ready for execution. Instructions can only be dispatched to a unit if its reservation station is guaranteed to have at least one entry available. (Instructions may be removed from the reservation station and executed out-of-order in all units except the floating-point unit. When multiple instructions in the reservation station are available for execution, the oldest instruction will be selected.)
- Only one branch instruction can dispatch per cycle. There are no restrictions on the dispatch of instructions after a branch, unless the branch unit determines that there may have been a prefetch error. In this case, the branch unit will halt the dispatch of the 'wrong-path' instructions after the branch and begin fetching instructions from what it knows or suspects to be the correct path.
- Only one instruction which may update the count register (CTR) may be pending completion at one time. A second update-CTR instruction will not dispatch until the first completes. In addition, any branch instruction which uses the CTR will not dispatch while a move-to-CTR instruction is pending.
- There are also interlock mechanisms between instructions which update the Link Register (LR). In general, two LR renames are supported. This allows, at most, two instructions which update the LR to be pending completion. There are additional constraints regarding the LR. A move-to-LR will not dispatch if there is a previous move-to-LR pending execution. A move-from-LR will not dispatch if there is a previous move-to-LR or move-from-LR pending execution. A branch with LK bit set will not dispatch if there is a move-to-LR pending execution.
- The 620 can handle as many as four branch instructions in the execute stage. The dispatch stalls on the first instruction after the fourth branch until the first branch executes. The 620 does not restrict the number of branches which have finished execution, but have not completed.
- An instruction may not be dispatched if a serialization mode is in effect for the instruction:
  - When in single-step trace mode, branch trace mode, or single instruction mode, an instruction will not be dispatched until the previous instruction has completed.
  - An instruction will not be dispatched if there is a previous Single-Instruction serialized instruction which has not finished execution.
  - An instruction will not be dispatched if a soft-stop breakpoint event occurred on the previous instruction.

- No instructions are dispatched during a 'global cancel' or if a 'halt' command from the COP is being processed.
- An instruction in the last slot of the dispatch buffer will not dispatch if the instruction contains an **r**B field.

6.4 Instruction Serialization Modes

Some instructions, such as **mfspr** and most **mtspr** instructions, extended arithmetic instructions that require the carry bit, and condition register instructions, require serialization to execute correctly. For this reason, the 620 implements a simple serialization mechanism that allows such instructions to be dispatched properly but delays execution until they can be executed safely. When all previous instructions have completed and updated their results to the architectural states, the serialized instruction is executed by directly reading and updated in the architectural states. If the instruction target is a GPR, FPR, or the CR, the register is renamed to allow later nondependent instructions to execute.

Store instructions are dispatched to the LSU where they are translated and checked for exception conditions. If no exception conditions are present, the instruction is passed to the store queue where it waits for all previous instructions to complete before it can be completed. Direct-memory accesses are handled in the same way to ensure that exceptions are precise. Serialization modes are described in the following sections.

6.4.1 Single Instruction Serialization

During the first cycle in which the serializing instruction is present in the dispatch buffer the instruction will be decoded. At this point the instruction will be identified as a serializing instruction. If the instruction belongs to single instruction serialization class, the serializing instruction will be dispatched as usual and all subsequent instructions will be prevented from dispatching in this cycle. The drain time period begins at this point.

During the drain time period, all subsequent instructions will remain in the dispatch buffer. While in the dispatch buffer, they may shift positions within the buffer but they may not be dispatched. This time period will continue until all instructions prior to the serializing instruction have completed execution and all architecture facilities are updated. At this time, the single-instruction-mode time period begins.

At the beginning of the single-instruction-mode time period, all source operands for the serializing instruction are guaranteed to be available from the appropriate architectural facility. During the execution cycle any architectural facilities which are updated by the serializing instruction are updated directly; all rename buffers and rename registers are bypassed. After the instruction finishes execution, the unit in which the instruction is executing will send a finish signal to the completion and dispatch blocks. The serializing instruction is guaranteed to be completed by the completion logic in the cycle immediately after it finishes because none of the completion constraints will prevent it from completing, as the machine is empty of all other activity. The serializing instruction will not pass through the write-back pipeline stage, as all of its destination writes will have completed during the execution pipeline stage.

Upon receiving the finish signal for the serializing instruction, the dispatch mechanism will begin dispatching the instructions following the serializing instruction normally. This will constitute the end of the single-instruction-mode time period.

Instructions causing single instruction serialization include:

- Instructions which update GPRs in non-rename mode, such as load multiples and load string instructions
- Instructions which update the entire condition register, such as **mtcrf**
- Instructions defined by the architecture to have context synchronizing behavior
- Cache operations which modify or invalidate the content of a cache line

6.4.2 Execution Serialization

The occurrence of an execution serialization instruction has no effect on the dispatching and execution of any following instructions. The only difference between an execution serialization instruction and a non-serialization instruction is that the execution serialization instruction cannot be executed until it is the oldest instruction in the system. In another words, the instruction will be dispatched into a reservation station, but cannot be issued to execution until the completion block informs the execution unit to execute the instruction.

Instructions causing execution serialization include:

- Instructions which read CA and OV bits
- Instructions which write the whole XER (except SO bit)
- Instructions which store multiple data items to memory
- Instructions which read SPRs and write MSR
- Instructions which modify SPRs except CTR/LR
- MMU operations which modify the contents of MMU
- Instructions which modify reservation
- Load access to T = 1 and cache-inhibited space
- Instructions which may alter the operating mode of the FPU
- Floating-point instructions with bit Rc = 1
- Load/store instructions which require extra data alignment cycles, such as load-byte-reverse instructions, and load/store crossing the double-word boundary.

6.4.3 Refetch Serialization

The occurrence of refetch serialization instruction prevents the 620 from dispatching following instructions, and it will cause the 620 to cancel all outstanding instructions in the processor when the serialization instruction is completed and refetch instructions after the serialization instructions.

Instructions causing execution serialization include:

- **isync**, **rfi**
- Instructions which change the SO bit in the XER
- Store instructions which detect the out-of-order execution of load instructions accessing the same doubleword.

6.4.4 Other Serialization Modes

This section provides additional serialization modes implemented on the 620.

- Single Instruction Step Trace Mode—The 620 will serialize all instructions when in single-step trace mode, MSR[SE] = 1.
- Branch Trace Mode—The 620 will serialize all instructions when in Branch Trace mode, MSR[BE] = 1.
- Single Instruction Dispatch Mode—The 620 will serialize all instructions when in Single Instruction dispatch mode, HID0[24] = 0.

6.5 Instruction Execution Timing

Table 6-1 shows the latency and throughput of each instruction (sorted alphabetically by mnemonic).

General assumptions used to derive the numbers in this table:

- Data and instruction accesses hit in the L1 caches.
- Floating point operations do not involve zeros, NaNs, or infinity.
- References to memory are aligned.

Notations used in the table:

- Execution Unit
  - — IPU: Instruction Processing Unit
  - — IFU: Instruction Flow Unit
  - — LSU: Load/Store Unit
  - — SCIU: Single-cycle Integer Unit
  - — MCIU: Multi-cycle Integer Unit
  - — FPU: Floating-Point Unit
  - — CRLU: Condition Register Logical Unit

- Latency—Latency is defined as the number of processor cycles from when the instruction begins execution to when the execution result is available to dependent instructions. If the execution time varies dependent on the data operands, then a best-case/worst-case range is given. For instructions that do not update architectural registers, such as store instructions, the execution time means the time it takes to execute this instruction back to back. Latency also includes the serialization penalty.
- Throughput—Throughput is defined as the number of processor cycles required per instruction for a series of independent instructions. Throughput also includes the serialization penalty.
- Serialization Scheme
  - — Single_Inst_Ser: The instruction causes Single Instruction Serialization.
  - — Exec_Ser: The instruction causes Execution Serialization.
  - — Refetch: The instruction causes Refetch Serialization.
  - — Non-Pipelined: The instruction cannot be executed in a pipelined fashion.

Table 6-1. Instruction Execution Timing Sorted by Mnemonic

Instruction	Execution Unit	Latency	Throughput	Serialization
add	SCIU	1	1	—
addc	SCIU	1	1	—
adde	SCIU	2	1	Exec_Ser
addi	SCIU	1	1	—
addic	SCIU	1	1	—
addic.	SCIU	1	1	—
addis	SCIU	1	1	—
addme	SCIU	2	1	Exec_Ser
addze	SCIU	2	1	Exec_Ser
and	SCIU	1	1	—
andc	SCIU	1	1	—
andi.	SCIU	1	1	—
andis.	SCIU	1	1	—
b	IPU	0	1	—
bc	IPU	0	1	—
bcctr	IPU	0	1	—
bclr	IPU	0	1	—
cmp	SCIU	1	1	—
cmpi	SCIU	1	1	—

6

Table 6-1. Instruction Execution Timing Sorted by Mnemonic (Continued)

Instruction	Execution Unit	Latency	Throughput	Serialization
cmpl	SCIU	1	1	—
cmpli	SCIU	1	1	—
cntlzd	MCIU	2	1	—
cntlzw	MCIU	2	1	—
crand	CRLU	2	3	Exec_Ser
crandc	CRLU	2	3	Exec_Ser
creqv	CRLU	2	3	Exec_Ser
crnand	CRLU	2	3	Exec_Ser
crnor	CRLU	2	3	Exec_Ser
cror	CRLU	2	3	Exec_Ser
crorc	CRLU	2	3	Exec_Ser
crxor	CRLU	2	3	Exec_Ser
dcbf	LSU	15	15	Single_Inst_Ser
dcbi	LSU	30	30	Single_Inst_Ser
dcbst	LSU	15	1	Exec_Ser
dcbt	LSU	2	1	—
dcbtst	LSU	2	1	—
dcbz	LSU	7	7	Single_Inst_Ser
divd	MCIU	37	36	Non-Pipelined
divdu	MCIU	37	36	Non-Pipelined
divw	MCIU	37	36	Non-Pipelined
divwu	MCIU	37	36	Non-Pipelined
eciwx	LSU	26		—
ecowx	LSU	3		—
eieio	LSU	22	Not Applic.	Exec_Ser
eqv	SCIU	1	1	—
extsb	SCIU	1	1	—
extsh	SCIU	1	1	—
extsw	SCIU	1	1	—
fabs	FPU	3	1	—
FP Instructions with Rc=1	FPU	4	1	Exec_Ser

Table 6-1. Instruction Execution Timing Sorted by Mnemonic (Continued)

Instruction	Execution Unit	Latency	Throughput	Serialization
fadd	FPU	3	1	—
fadds	FPU	3	1	—
fcfid	FPU	3	1	—
fcmpo	FPU	3	1	—
fcmpu	FPU	3	1	—
fctid	FPU	3	1	—
fctidz	FPU	3	1	—
fctiw	FPU	3	1	—
fctiwz	FPU	3	1	—
fdiv	FPU	18 - 25	18 - 25	Non-Pipelined
fdivs	FPU	18 - 25	18 - 25	Non-Pipelined
fmadd	FPU	3	1	—
fmadds	FPU	3	1	—
fmr	FPU	3	1	—
fmsub	FPU	3	1	—
fmsubs	FPU	3	1	—
fmul	FPU	3	1	—
fmuls	FPU	3	1	—
fnabs	FPU	3	1	—
fneg	FPU	3	1	—
fnmadd	FPU	3	1	—
fnmadds	FPU	3	1	—
fnmsub	FPU	3	1	—
fnmsubs	FPU	3	1	—
fres	FPU	3	1	—
frsp	FPU	3	1	—
frsqrte	FPU	3	1	—
fsqrt	FPU	21 - 30	21 - 30	Non-Pipelined
fsel	FPU	3	1	—
fsqrts	FPU	21 - 30	21 - 30	Non-Pipelined
fsub	FPU	3	1	—
fsubs	FPU	3	1	—

Table 6-1. Instruction Execution Timing Sorted by Mnemonic (Continued)

Instruction	Execution Unit	Latency	Throughput	Serialization
icbi	LSU	27		Exec_Ser
isync	IFU	6		Refetch
lbz	LSU	2	1	—
lbzu	LSU	2	1	—
lbzux	LSU	2	1	—
lbzx	LSU	2	1	—
ld	LSU	2	1	Exec_Ser if misaligned
ldarx	LSU	3	Not Applic.	Exec_Ser
ldu	LSU	2	1	Exec_Ser if misaligned
ldux	LSU	2	1	Exec_Ser if misaligned
ldx	LSU	2	1	Exec_Ser if misaligned
lfd	LSU	2	1	Exec_Ser if misaligned
lfdu	LSU	2	1	Exec_Ser if misaligned
lfdux	LSU	2	1	Exec_Ser if misaligned
lfdx	LSU	2	1	Exec_Ser if misaligned
lfs	LSU	2	1	Exec_Ser if misaligned
lfsu	LSU	2	1	Exec_Ser if misaligned
lfsux	LSU	2	1	Exec_Ser if misaligned
lfsx	LSU	2	1	Exec_Ser if misaligned
lha	LSU	2	1	Exec_Ser if misaligned
lhau	LSU	2	1	Exec_Ser if misaligned
lhaux	LSU	2	1	Exec_Ser if misaligned
lhax	LSU	2	1	Exec_Ser if misaligned
lhbrx	LSU	5	1	Exec_Ser
lhz	LSU	2	1	Exec_Ser if misaligned
lhzu	LSU	2	1	Exec_Ser if misaligned
lhzux	LSU	2	1	Exec_Ser if misaligned
lhzx	LSU	2	1	Exec_Ser if misaligned
lmw	LSU	1 + # reg	1 + # reg	Single_Inst_Ser
lswi	LSU	1 + # reg	1 + # reg	Single_Inst_Ser
lswx	LSU	1 + # reg	1 + # reg	Single_Inst_Ser
lwa	LSU	2	1	Exec_Ser if misaligned

Table 6-1. Instruction Execution Timing Sorted by Mnemonic (Continued)

Instruction	Execution Unit	Latency	Throughput	Serialization
lwarx	LSU	3	Not Applic.	Exec_Ser
lwaux	LSU	2	1.	Exec_Ser if misaligned
lwax	LSU	2	1	Exec_Ser if misaligned
lwbrx	LSU	5	5	Exec_Ser
lwz	LSU	2	1	Exec_Ser if misaligned
lwzu	LSU	2	1	Exec_Ser if misaligned
lwzux	LSU	2	1	Exec_Ser if misaligned
lwzx	LSU	2	1	Exec_Ser if misaligned
mcrf	CRLU	2	3	Exec_Ser
mcrfs	FPU	4	1	Exec_Ser
mcrxr	MCIU	4	4	Exec_Ser
mfcr	MCIU	4	4	Exec_Ser
mffs	FPU	4	1	Exec_Ser
mfmsr	MCIU	5	5	Exec_Ser
mftb 268	MCIU	5	5	Exec_Ser
mftb 269	MCIU	4	4	Exec_Ser
mfspr	MCIU	4 - 16		Some are
mfsr	MCIU	9+	9	Exec_Ser, can be stalled by DCMMU
mfsrin	MCIU	9+	9	Exec_Ser, can be stalled by DCMMU
mtcrf (single field)	SCIU	2	1	
mtcrf (multiple fields)	MCIU	4	4	Single_Inst_Ser
mtfsb0	FPU	3	3	Exec_Ser
mtfsb1	FPU	3	3	Exec_Ser
mtfsf	FPU	4	1	Exec_Ser
mtfsfi	FPU	4	1	Exec_Ser
mtmsr	MCIU	4	4	Exec_Ser
mtmsrd	MCIU	4	4	Exec_Ser
mtspr LR/CTR	MCIU	4	4	

Table 6-1. Instruction Execution Timing Sorted by Mnemonic (Continued)

Instruction	Execution Unit	Latency	Throughput	Serialization
mtspr (others)	MCIU	4 - 16		Exec_Ser
mtsr	MCIU	7+	7	Exec_Ser, can be stalled by DCMMU
mtsrin	MCIU	7+	7	Exec_Ser, can be stalled by DCMMU
mulhd	MCIU	3 - 9	1	10 bits per iteration
mulhdu	MCIU	3 - 9	1	10 bits per iteration
mulhw	MCIU	3 - 6	1+	10 bits per iteration
mulhwu	MCIU	3 - 6	1+	10 bits per iteration
mulld	MCIU	3 - 9	1+	10 bits per iteration
mulli	MCIU	3 - 4	1+	10 bits per iteration
mullw	MCIU	3 - 6	1+	10 bits per iteration
nand	SCIU	1	1	—
neg	SCIU	1	1	—
nor	SCIU	1	1	—
or	SCIU	1	1	—
orc	SCIU	1	1	—
ori	SCIU	1	1	—
oris	SCIU	1	1	—
rfi	IFU	5		Refetch
rfid	IFU	5		Refetch
rldcl	SCIU	1	1	—
rldcr	SCIU	1	1	—
rldic	SCIU	1	1	—
rldicl	SCIU	1	1	—
rldicr	SCIU	1	1	—
rldimi	SCIU	1	1	—
rlwimi	SCIU	1	1	—
rlwinm	SCIU	1	1	—
rlwnm	SCIU	1	1	—
sc	IFU	5		Refetch
slbia	LSU	2		Exec_Ser

Table 6-1. Instruction Execution Timing Sorted by Mnemonic (Continued)

Instruction	Execution Unit	Latency	Throughput	Serialization
slbie	LSU	10		Exec_Ser
sld	SCIU	1	1	—
slw	SCIU	1	1	—
srad	SCIU	1	1	—
sradi	SCIU	1	1	—
sraw	SCIU	1	1	—
srawi	SCIU	1	1	—
srd	SCIU	1	1	—
srw	SCIU	1	1	—
stb	LSU	1	1	
stbu	LSU	2	1	
stbux	LSU	2	1	—
stbx	LSU	2	1	—
std	LSU	2	1	Exec_Ser if misaligned
stdcx.	LSU	3		Exec_Ser
stdu	LSU	2	1	Exec_Ser if misaligned
stdux	LSU	2	1	Exec_Ser if misaligned
stdx	LSU	2	1	Exec_Ser if misaligned
stfd	LSU	2	1	Exec_Ser if misaligned
stfdu	LSU	2	1	Exec_Ser if misaligned
stfdux	LSU	2	1	Exec_Ser if misaligned
stfdx	LSU	2	1	Exec_Ser if misaligned
stfiwx	LSU	2	1	Exec_Ser if misaligned
stfs	LSU	2	1	Exec_Ser if misaligned
stfsu	LSU	2	1	Exec_Ser if misaligned
stfsux	LSU	2	1	Exec_Ser if misaligned
stfsx	LSU	2	1	Exec_Ser if misaligned
sth	LSU	2	1	Exec_Ser if misaligned
sthbrx	LSU	2	1	Exec_Ser if misaligned
sthu	LSU	2	1	Exec_Ser if misaligned
sthux	LSU	2	1	Exec_Ser if misaligned
sthx	LSU	2	1	Exec_Ser if misaligned

Table 6-1. Instruction Execution Timing Sorted by Mnemonic (Continued)

Instruction	Execution Unit	Latency	Throughput	Serialization
stmw	LSU	1 + # reg	1 + # reg	Exec_Ser
stswi	LSU	1 + # reg	1 + # reg	Exec_Ser
stswx	LSU	1 + # reg	1 + # reg	Exec_Ser
stw	LSU	2	1	Exec_Ser if misaligned
stwbrx	LSU	2	1	Exec_Ser if misaligned
stwcx.	LSU	3		Exec_Ser
stwu	LSU	2	1	Exec_Ser if misaligned
stwux	LSU	2	1	Exec_Ser if misaligned
stwx	LSU	2	1	Exec_Ser if misaligned
subf	SCIU	1	1	—
subfc	SCIU	1	1	—
subfe	SCIU	2	1	Exec_Ser
subfic	SCIU	1	1	—
subfme	SCIU	2	1	Exec_Ser
subfze	SCIU	2	1	Exec_Ser
sync	LSU	26		Single_Inst_Ser
td	SCIU	1	1	—
tdi	SCIU	1	1	—
tlbie	LSU	22		Exec_Ser
tlbsync	LSU	26		Exec_Ser
tw	SCIU	1	1	—
twi	SCIU	1	1	—
xor	SCIU	1	1	—
xori	SCIU	1	1	—
xoris	SCIU	1	1	—

6.5.1 Performance of Load/Store Multiples and Strings Instructions

The following sections discuss the performance of load/store multiple and string instructions.

6.5.1.1 Load-Multiple Word and Doubleword from Cacheable Memory

The 620 will load one register per cycle with single-instruction serialization, which means load multiple cannot be executed until all previous instructions are completed, and no following instructions can be dispatched until load multiple is finished. The same mechanism is used to handle load string instruction whose starting address is word-aligned and byte count is multiple of four.

6.5.1.2 Store-Multiple Word to Cacheable Memory

The 620 will store one register per cycle with execution serialization, which means store multiple can be dispatched, but cannot be executed until it's the oldest instruction in the processor. All following instructions can be dispatched and executed as usual.

The same mechanism is used to handle store string instruction whose starting address is word-aligned and byte count is multiple of four.

6.5.1.3 Load-Multiple Word from Cache-Inhibited Memory

The 620 will generate one bus transaction every register. There is no data gathering in this case, and the transfers are not pipelined. Same serialization scheme is used as Load Multiple from cacheable memory.

The same mechanism is used to handle load string instruction whose starting address is word-aligned and byte count is multiple of four.

6.5.1.4 Store-Multiple Word to Cache-Inhibited Memory

620 will split a store-multiple instruction to cache-inhibit memory into multiple single-word stores, and Data Cache Unit will use the store-gathering mechanism to gather these single-word stores into doubleword packets and transfer these packets to the bus. Detailed description of the store mechanism is presented in the next section. In the ideal case where bus is always available and processor core can generate data fast enough, Same serialization is used as store multiple to cacheable memory.

The same mechanism is used to handle store string instruction whose starting address is word-aligned and byte count is multiple of four.

6.5.1.5 String Instructions that are not Treated as Load/Store Multiples

For this kind of string instructions, the 620 will access 4-bytes at a time if the access does not cross a doubleword boundary; otherwise, the access will be split into two separate accesses. The same mechanism is used to handle to/from cache-inhibited memory, and each access will generate a bus transaction; refer to Table 6-2.

Table 6-2. Performance of String Instructions not Treated as a Load/Store Multiple

# of bytes	Placement	Accesses
1	don't care	1
2–4	within_DW	1
2–4	cross_DW	2
5–8	within_DW	2
5–8	cross_DW, word_aligned	2
5–8	cross_DW, byte_aligned	3
9–12	word_aligned	3
9–12	within_two_DW	4
9–12	cross_two_DW	5

6.5.2 Store Gathering for Cache-Inhibited and Write-Through Stores

The 620 may gather consecutive stores in a quad word store buffer if they satisfy the following conditions:

- Have the same WI setting, and W = 1 or I = 1, and T = 0, and not atomic, and not **ecowx**.
- Have same operand size of either word or double word.

  String operation with word-aligned operand will be split into multiple store word, so they will be gathered too.
- Address of the following store is in the same quad word of the preceding store, and its word-offset is greater than the preceding store by one for store word, or by two for store double.

The store buffer will be flushed whenever a count of two words is reached during store words, or a count of four words during store doubles, or the succeeding store does not satisfy any of the above conditions, or there are no more stores in the completion store queue.Word stores can only be gathered in two word quantities because the 620 bus does not support three word transfers and there is no way of telling whether a fourth word will be available to obtain an acceptable store size (one quad word). Maximum throughput is one quad word every 12 processor cycles for cache-inhibited or write-through store doubles. Cache-inhibited or write-through store singles have the same maximum throughput, but only occupy half a quad word.

Gathering can be encouraged in the hardware by writing code which satisfies the above conditions, but no code sequence can guarantee that gathering will occur in the hardware. This variability is primarily caused by the fact that 620 can only gather based on stores that are available in the completed store queue. Since store will be processed as soon as they arrive in the completed store queue, gathering can be dramatically impacted by completed store queue utilization.

6.5.3 Cache-Inhibited or Write-Through Store Bandwidth

This section describes the system conditions necessary to obtain the cache-inhibited or write-through store maximum bandwidth of one quad word store every 14 processor cycles (PCLKs). Given the following conditions, the store bandwidth will be the same for cache-inhibited or write-through stores because these types of stores are strongly ordered with respect to memory (as opposed to burst stores which are not strong ordered). For write through stores, it is assumed that the L2 interface does not become the long path (that is, no non write through activity that could block write buffers to the L2). The necessary system conditions follow:

- The processor to bus ratio is 2:1.
- The master is address bus parked for the address-data transactions, receiving $\overline{\text{ABG}}$ the same cycle as the processor internal $\overline{\text{ABR}}$ goes active. See Section 8.3.10, “Bus Parking.” If the master can not be address bus parked, then one extra bus clock will be added to the arbitration time and if a combinatorial arbiter is not used, one additional cycle will be added for $\overline{\text{ABG}}$ beyond $\overline{\text{ABR}}$.
- The DBR and DBG occur in one bus clock due to a combinatorial arbiter.
- The BUSRESPTEN is the minimum value of two bus clocks (so ASTATOUT/IN have a tenure of two bus cycles). If the tenure is three bus clocks, then the extra bus cycle will be added to the bandwidth. See Section 8.4.6, “Address Status and Address Response Tenure (BUSRESPTEN[0–1]).”
- The address tenure is one bus clock. If the tenure of the address is more then one bus clock, then this may limit continuous store bandwidth and result in some idle bus clocks when write data could have been transferred. See Section 8.3.9, “Early Address Transfer Start (EATS).”

6.6 L1, L2, and Memory Load Latency

This section defines the latency of data and instruction memory loads. This analysis does not take into account additional latency due to a busy resource, either internal or external to the 620. Latency is expressed in terms of PCLKs, L2CLOCKs, and BUSCLKs, which are processor clocks, L2 clocks, and bus clocks. The PCLK is always the highest frequency clock and the L2CLOCK and BUSCLK are always an integer multiple of PCLKs, but not necessarily the same multiple (ratio). (Note: L2CLOCK = L2RATIOSR PCLKs and BUSCLK = BUSRATIO PCLKs.)

6.6.1 L1 Data Cache Latency

Table 6-3 outlines the latency for an L1 data cache access.

Table 6-3. L1 Data Cache Latency

#	Owner	Latency Description	Parameter	Value	Unit
1	LD/ST	Effective Address Calculation	L1EA	0.5	PCLK
2	DCache	L1 Cache Access	L1CA	1	PCLK
3	L1 Hit: Add 0 latency. L1 Miss: Add latency from Table 6-6.				
4	LD/ST	Data Align/Extend, Drive, and Setup	L1RD	0.5	PCLK

6.6.2 L2 Cache Latency

Table 6-5 outlines the latency for an L2 cache access.

Table 6-4. L2 Cache Latency

#	Owner	Latency Description	Parameter	Value	Unit
1	DCache	Tag Comparison	L1TC	0.5	PCLK
2	BIU	Address Muxed to L2 Interface	L2AM	1	PCLK
3	BIU	PCLK to L2CLOCK transition	L2CK	(see Note)	PCLK
4	BIU	SRAM External Address	L2PR	L2RATIOSR	PCLK
5	BIU	SRAM Array Access and External Data	L2SR	L2RATIOSR * L2RATIOSR	PCLK
6	BIU	ECC + Tag Compare	L2TC	1	PCLK
7	L2 Hit: Add 0 latency. L2 Miss: Add latency from Table 6-6.				
8	DCache	Data passes rising to falling edge	L2RF	0.5	PCLK

Note: Worst case transition from PCLK to L2CLOCK is (1 L2CLOCK – 1 PCLK). Best case is 0 PCLK.

6.6.3 L2 Cache Disabled Latency

When the L2 is disabled for a cacheable read operation, address collision detection is not hidden underneath the L2 cache access latency and becomes a sequential latency component. Table 6-5 outlines the latency for an L2-disabled cacheable access.

Table 6-5. L2 Disabled Latency

#	Owner	Latency Description	Parameter	Value	Unit
1	DCache	Tag Comparison	L1TC	0.5	PCLK
2	BIU	Address Muxed to L2 Interface	L2AM	1	PCLK
3	BIU	PCLK to L2CLOCK transition	L2CK	(see Note)	PCLK
4	BIU	Collision detection compare	L2CD1	1	PCLK
5	BIU	Collision detection grant	L2CD2	1	PCLK
6	Add latency from Table 6-6 starting with Parameter MIA.				
7	DCache	Data passes rising to falling edge	L2RF	0.5	PCLK

Note: Worst case transition from PCLK to L2CLOCK is (1 L2CLOCK – 1 PCLK). Best case is 0 PCLK.

6.6.4 Bus Read Latency

Table 6-6 defines the latency for a bus read.

Table 6-6. Bus Read Latency

#	Owner	Latency Description	Parameter	Value	Unit
1	BIU	L2 Coherency to Bus Interface Req.	MCR	1	PCLK
2	BIU	Internal arbitration for address bus	MIA	1	PCLK
3	BIU	PCLK to BUSCLK transition	MPB	0–1	BUSCLK[1]
4	XARB	Bus Arbitration	MAL	0–N	BUSCLK[2]
5	Bus	Address Transferred	MAT	1–2	BUSCLK[3]
6	Bus	Latency from address to response	MRL	BUSTLAR	BUSCLK
7	Bus	Snoop response transferred	MRT	2–3	BUSCLK[4]
8	Bus	Response logic and data bus arb	MAD	2	PCLK
9	Bus	No Return Data	MDL	1–N	BUSCLK
10	Bus	First return data valid	MDT	1	BUSCLK
11	BIU	Data passed to Bus load buffer	MDR	1	PCLK
12	DCache	Load bus arbitration	MLA	1	PCLK

Notes:

1. Worst case transition from PCLK to BUSCLK is (1 BUSCLK – 1 PCLK). Best case is 0 PCLKs.
2. Determined by external bus arbiter. 0 bus clock cycles for bus parked case, 1 bus clock cycle for combinatorial arbiter case, and 2 bus clock cycles for registered arbiter case.
3. Determined by EATS.
4. Determined by BUSRESPTEN.

6.6.5 Load Latency Example #1: IL1 or DL1 Miss and L2 Hit

This example illustrates the timing of both data and instruction L1 misses that hit in the L2. The data load and instruction timing is common for the timing parameters grouped as BIU and are different for the timing parameters grouped as Data Load and IFetch. The L2 is double-synchronous with L2RATIOSR equal to one PCLK. The address arrives at the L2 interface in time to be driven to the SRAMS. It takes one PCLK to drive the L2 address to the SRAMs. It takes one PCLK to access the SRAMS. It takes one PCLK to pass the data from the SRAMs to the processor. It takes one PCLK to check ECC and detect a hit. Figure 6-4 illustrates this example:

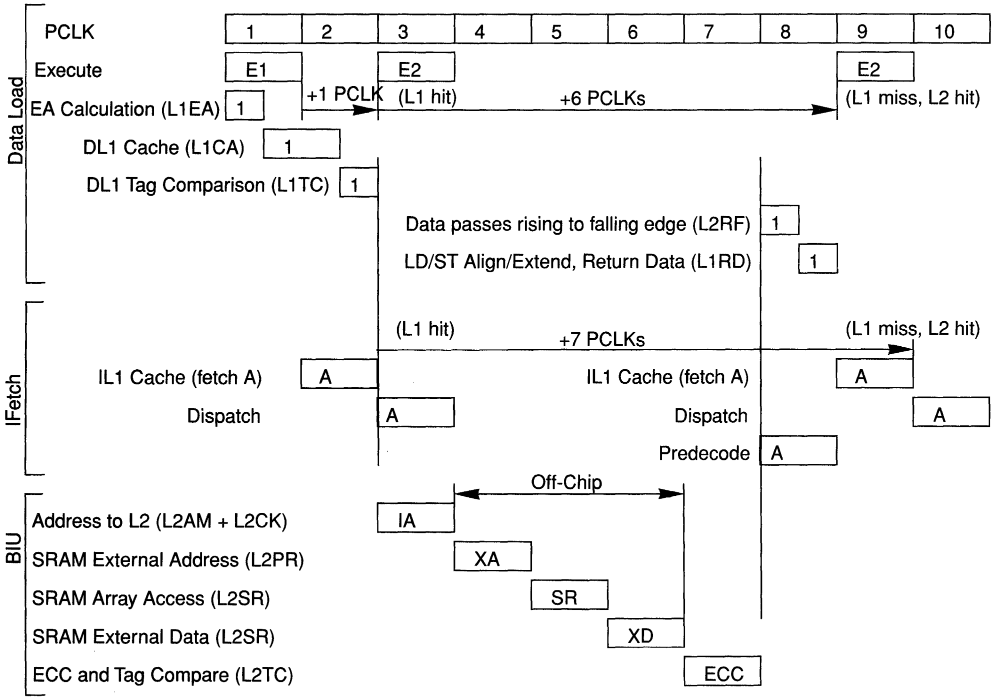


Figure 6-4. Load Latency Example #1: DL1 Miss and L2 Hit

6.6.6 Load Latency Example #2: DL1 and L2 Miss

The data L1 and L2 accesses miss and the L2 is enabled. The address arrives at the L2 interface in time to be driven to the SRAMs. It takes one PCLK to drive the L2 address to the SRAMs. It takes one PCLK to access the SRAMS. It takes one PCLK to pass the data from the SRAMs to the processor. It takes one PCLK to detect a miss and make a bus request. The bus clock ratio is 2:1. The address bus is granted in the same BUSCLK cycle the request is made. The address bus is one BUSCLK cycle in duration. Since data is not returned until after ARESPIN, BUSTLAR is three BUSCLKs and BUSRESPTEN is two BUSCLKs. Table 6-7 provides the calculation of the latency in PCLK cycles.

Table 6-7. Load Latency Example #2: DL1 and L2 Miss

#	Owner	Latency Description	Parameter	PCLKs
1	LD/ST	Effective address calculation	L1EA	0.5
2	DCache	L1 Cache Access	L1CA	1
3	DCache	Tag Comparison	L1TC	0.5
4	BIU	Address muxed to L2 interface	L2AM	1
5	BIU	SRAM External Address	L2PR	1
6	BIU	SRAM Array Access and External Data	L2SR	2
7	BIU	ECC + Tag compare (L2 miss)	L2TC	1
8	BIU	L2 Coherency to Bus Interface Req.	MCR	1
9	BIU	Internal arbitration for address bus	MIA	1
10	BIU	PCLK to BUSCLK transition	MPB	0
11	XARB	Addr Bus arbitration ($\overline{\text{ABR}}$ and $\overline{\text{ABG}}$ in same BUSCLK)	MAL	2
12	Bus	Address transferred	MAT	2
13	Bus	Latency from address to response	MRL	6
14	Bus	Snoop response transferred	MRT	4
15	Bus	Response logic and data bus arb	MAD	2
16	Bus	No Return Data	MDL	2
17	Bus	First return data valid	MDT	2
18	BIU	Data passed to bus load buffer	MDR	1
19	BIU	Load bus arbitration	MLA	1
20	DCache	Data passes rising to falling edge	L2RF	0.5
21	LD/ST	Align/Extend, Return data	L1RD	0.5
22	Total Latency in PCLKs			32

6.6.7 Load Latency Example #3: DL1 Miss and L2 Disabled

The data L1 access misses. The L2 is disabled. The bus clock ratio is 2:1. The address bus is parked. The address bus is one BUSCLK cycle in duration. BUSTLAR is three BUSCLK cycles (six PCLKs). The response tenure is two BUSCLK cycles (four PCLKs). Table 6-8 provides the calculation of the latency in PCLK cycles.

Table 6-8. Load Latency Example #3: DL1 Miss, L2 Disabled

#	Owner	Latency Description	Parameter	PCLKs
1	LD/ST	Effective address calculation	L1EA	0.5
2	DCache	L1 Cache Access	L1CA	1
3	DCache	Tag Comparison	L1TC	0.5
4	BIU	Address Muxed to L2 Interface	L2AM	1
5	BIU	Collision detection compare	L2CD1	1
6	BIU	Collision detection grant	L2CD2	1
7	BIU	Internal arbitration for address bus	MIA	1
8	BIU	PCLK to BUSCLK transition	MPB	0
9	XARB	Bus arbitration	MAL	0
10	Bus	Address transferred	MAT	2
11	Bus	Latency from address to response	MRL	6[1]
12	Bus	Snoop response transferred	MRT	4[2]
13	BIU	Response logic and data bus arb	MAD	2
14	Bus	No Return Data	MDL	2
15	Bus	First return data valid	MDT	2
16	BIU	Data/response passed to bus load buffer	MDR	1
17	BIU	Load bus arbitration	MLA	1
18	DCache	Data passes rising to falling edge	L2RF	0.5
19	LD/ST	Align/Extend, Return data	L1RD	0.5
20	Total Latency in PCLKs			27

Notes:

[1] BUSTLAR = 3, BUSRATIO = 2:1.

[2] BUSRESPTEN=2, BUSRATIO=2:1.

6.7 Snoop Push/Intervention Latency

This section defines the latency from the 620 snooping a bus operation that causes a push or intervention.

To calculate latency from the perspective of a master, start with Table 6-3. Use Table 6-5 if the L2 is enabled in the master or Table 6-5 if the L2 is disabled in the master. Table 6-9 provides push/intervention latency information for the L1 cache.

Table 6-9. The L1 Contains the Push/Intervention Data

#	Latency Description	Parameter	Value	Unit
1	Latency from address to response	MRL	BUSTLAR	BUSCLK
2	Snoop response transferred	MRT	2–3	BUSCLK
3	Response evaluated	BSARI	1	PCLK
4	Request and grant for L1 snoop interface	L1RG	1–N	PCLK[1]
5	Request and grant for bus store buffer	SBRG	1–N	PCLK[2]
6	Request and transfer first data from the L1 (assuming the L1 is idle).	L1D1	7	PCLK
7	The 2nd to 4th datums are passed to the bus store buffer.	L1D24	3	PCLK
8	Store buffer valid	BSBV	1	PCLK
9	Internal arbitration for data bus	MID	1	PCLK
10	PCLK to BUSCLK transition	MPB	0–N	PCLK

Notes:

1. Dependent on priority of other requestors.
2. Dependent on priority of other requestors and number of empty store buffers.

Table 6-10 provides push/intervention latency information for the L2 cache.

Table 6-10. The L2 Contains the Push/Intervention Data

#	Latency Description	Parameter	Value	Unit
1	Bus address sample to L2 status return. Refer to the BUSTLAR equation in Section 8.4.5.3, "L2 Cache and Coprocessor Mode Disabled."	BUSTLAR	equation result	PCLK
2	Start bus snoop state machine	BSSM	1	PCLK
3	State machine check of bus status.	BSASI	0–N	PCLK[1]
4	Request and grant for bus store buffer	SBRG	1–N	PCLK

Table 6-10. The L2 Contains the Push/Intervention Data (Continued)

#	Latency Description	Parameter	Value	Unit
5	Get the data from the L2. The latency from the assertion of the L2 request to the first data. Refer to the BUSTLAR equation in Section 8.4.5.3, "L2 Cache and Coprocessor Mode Disabled."	BUSTLAR	equation result	PCLK
6	Transfer first data to the bus store buffer.	L2D1	1	PCLK
7	The 2nd to 4th datums are passed to the bus store buffer.	L2D24	3 * L2RATIOSR	PCLK
8	Store buffer valid	BSBV	1	PCLK
9	Internal arbitration for data bus	MID	1	PCLK
10	PCLK to BUSCLK transition	MPB	0–N	PCLK

Note: Dependent on BUSRESPTEN value vs. L2 pipeline latency.

6.7.1 Push/Intervention Example #1: L2 Enabled and data in L1

6

The latency for this case is listed in Table 6-11. The L2 is enabled and the data is in the L1. BUSTLAR=5 and BUSRESPTEN=2.

Table 6-11. Push/Intervention Example #1: L2 Enabled and data in L1

#	Latency Description	Parameter	PCLKs
1	Latency from address to response	MRL	10
2	Snoop response transferred	MRT	4
3	Response evaluated	BSARI	1
4	Request and grant for L1 snoop interface	L1RG	1
5	Request and grant for bus store buffer	SBRG	1
6	Request and transfer first data from the L1 (assuming the L1 is idle).	L1D1	7
7	The 2nd to 4th datums are passed to the bus store buffer.	L1D24	3
8	Store buffer valid	BSBV	1
9	Internal arbitration for data bus	MID	1
10	PCLK to BUSCLK transition	MPB	1
Total latency in PCLKs			30

6.7.2 Push/Intervention Example #2: L2 Enabled and data in L2

The latency for this case is listed in Table 6-12. The L2 is enabled and the data is in the L2. The parameters for the BUSTLAR equation are BUS_TO_L2=1, L2RATIOSR=2 and L2_SS=1, BUSRESPTEN=2.

Table 6-12. Push/Intervention Example #2: L2 Enabled and data in L2

#	Latency Description	Parameter	PCLKs
1	Bus address sample to L2 status return.	BUSTLAR	5
2	Start bus snoop state machine.	BSSM	1
3	State machine check of bus status.	BSASI	1
4	Request and grant for bus store buffer.	SBRG	1
5	Get the data from the L2.	BUSTLAR	5
6	Transfer first data to the bus store buffer.	L2D1	1
7	The 2nd to 4th datums are passed to the bus store buffer.	L2D24	6
8	Store buffer valid	BSBV	1
9	Internal arbitration for data bus	MID	1
10	PCLK to BUSCLK transition	MPB	0
Total latency in PCLKs			22

Chapter 7
Signal Descriptions

This chapter describes the PowerPC 620 microprocessor's external signals. It contains a concise description of individual signals, showing behavior when the signal is asserted and negated and when the signal is an input and an output.

NOTE

> A bar over a signal name indicates that the signal is active low—for example, $\overline{\text{DBG}}$ (data bus grant) and $\overline{\text{EATS}}$ (early address transfer start). Active-low signals are referred to as asserted (active) when they are low and negated when they are high. Signals that are not active low, such as AP[0–2] (address bus parity signals) and DTAG[0–7] (data tag signals) are referred to as asserted when they are high and negated when they are low.

The 620's signals are grouped as follows:

- Address arbitration signals—The 620 uses these signals to arbitrate for address bus mastership.
- Address transfer start signal—This signal indicates that a bus master will end its transaction on the address bus.
- Address transfer signals—These signals, which consist of the address bus and address parity signals are used to transfer the address and to ensure the integrity of the transfer.
- Transfer attribute signals—These signals provide information about the type of transfer, the transfer size, and whether the transfer is a burst transaction.
- Address transfer response signals—These signals are used to acknowledge the end of the address phase of the transaction. They also indicate whether a condition exists that requires the address phase to be repeated.
- Data arbitration signals—The 620 uses these signals to arbitrate for data bus mastership.
- Data transfer signals—These signals, which consist of the data bus, the data parity signals, and data bus qualifier are used to transfer the data and to ensure the integrity of the transfer.

- Data transfer termination signals—Data termination signals are required after each data beat in a data transfer. In a single-beat transaction, the data termination signals also indicate the end of the tenure, while in burst accesses, the data termination signals apply to individual beats and indicate the end of the tenure only after the final data beat. They also indicate whether a condition exists that requires the data phase to be repeated.
- System status signals—These signals include the interrupt and reset signals.
- Processor state signals—These signals are used to indicate the state of the reservation coherency bit and checkstop.
- L2 cache interface— These signals include the L2 cache address, data, clocking, ECC, tagging, and enable signals.
- Miscellaneous signals—These signals are used in conjunction with such resources as secondary caches, time base, PLL configuration, and test and debug facilities.
- COP interface signals—The common on-chip processor (COP) unit is the master clock control unit and it provides a serial interface to the system for performing built-in self test (BIST) on all internal memory arrays.
- Clock signals—These signals determine the system clock frequency. These signals can also be used to synchronize multiprocessor systems.

7.1 Signal Configuration

Figure 7-1 illustrates the logical signal configuration of the 620, showing how the signals are grouped.

NOTE

A pinout showing actual pin numbers is included in the 620 hardware specifications.

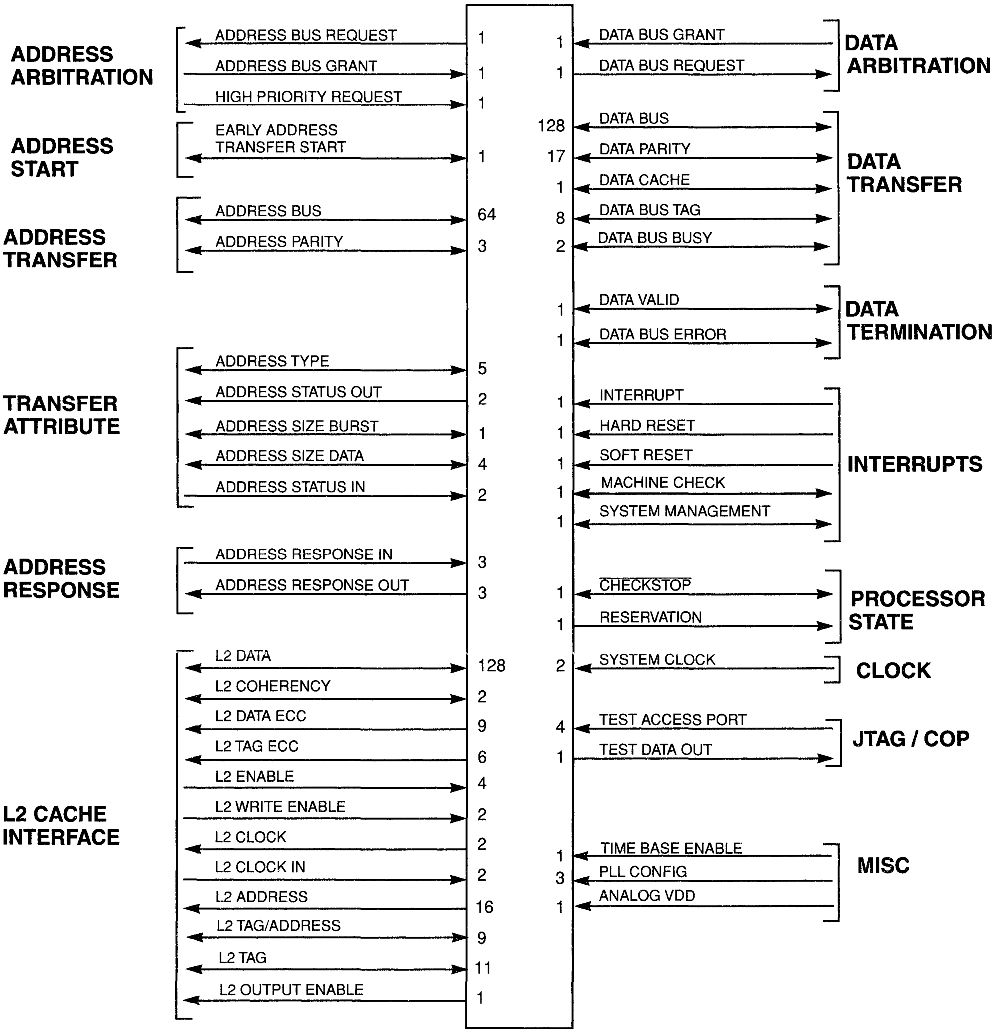


Figure 7-1. PowerPC 620 Microprocessor Signal Groups

7.2 Signal Descriptions

This section describes individual 620 signals, grouped according to Figure 7-1. Note that the following sections are intended to provide a quick summary of signal functions. Chapter 8, “System Interface Operation,” describes many of these signals in greater detail, both with respect to how individual signals function and how groups of signals interact.

7.2.1 Address Bus Arbitration Signals

The address arbitration signals are a collection of input and output signals the 620 uses to request the address bus and recognize when the request is granted. For a detailed description of how these signals interact, see Section 8.3, “Arbitration.”

7.2.1.1 Address Bus Request ($\overline{\text{ABR}}$)—Output

The address bus request ($\overline{\text{ABR}}$) signal is a point-to-point signal from the 620 to the arbiter. Following are the state meaning and timing comments for the $\overline{\text{ABR}}$ signal.

State Meaning Asserted—Indicates that the 620 is requesting mastership of the address bus.

The request is pending for as long as the request is asserted. This is called a level request implementation, as opposed to a pulsed request implementation, and only 1 request can be pending at any given time by a single 620.

Negated—Indicates that the 620 is not requesting the address bus.

Timing Comments Assertion—Occurs when an address bus transaction is needed and the 620 does not have a qualified address bus grant.

Negation—Occurs when the address bus grant is received, even if another transaction is pending.

7.2.1.2 Address Bus Grant ($\overline{\text{ABG}}$)—Input

The $\overline{\text{ABG}}$ is a point-to-point active low signal from the arbiter to the 620 and is asserted low. If $\overline{\text{ABG}}$ is asserted in a manner not compliant with the rules stated in Chapter 8, “System Interface Operation,” it may result in undefined behavior.

State Meaning Asserted—$\overline{\text{ABG}}$ is a pulsed grant, asserted for one bus clock cycle, that indicates the address bus is being granted.

When $\overline{\text{ABG}}$ is asserted, it indicates that the address bus can start driving the address bus on either the same edge that $\overline{\text{ABG}}$ is sampled asserted or 1 processor clock cycle later.

Negated—Indicates that the 620 is not the next potential address bus master.

Timing Comments Assertion—May occur at any time when $\overline{\text{ABR}}$ is asserted. $\overline{\text{ABG}}$ cannot be asserted more frequently than once every other cycle, with the exception of the sustained address bus parking case, described in Section 8.3.10.2, "Sustained Address Bus Parking."

Negation—May occur at any time to indicate the 620 cannot use the address bus.

7.2.1.3 High Priority Request ($\overline{\text{HPR}}$)—Output

The high-priority request (HPR) signal is an active low, output signal.

State Meaning Asserted—The $\overline{\text{HPR}}$ tells the arbiter that all high-priority bus operations and all non-high-priority operations in front of high priority bus operations should be treated as high priority bus operations. Section 8.3.4, "Internal Request Arbitration," for more information on low- and high-priority bus operations.

$\overline{\text{HPR}}$ will assert for a low priority request if the low-priority request was granted internally before the high-priority bus operation internally requested.

Negated—The $\overline{\text{HPR}}$ is negated when there are no outstanding high priority operations queued for arbitration onto the bus.

Timing Comments Assertion—The $\overline{\text{HPR}}$ is asserted when either $\overline{\text{ABR}}$ or $\overline{\text{DBR}}$ are asserted and if there is one or more high-priority bus operations that need, but have not yet been granted, the bus.

Negation—$\overline{\text{HPR}}$ will stay asserted until there are no outstanding high priority operations queued for arbitration onto the bus.

7.2.2 Address Transfer Start Signals

The address transfer start signal is an input signal that indicates that an address bus transfer has begun. The early address transfer start ($\overline{\text{EATS}}$) signal identifies the operation as a memory transaction. For detailed information about how $\overline{\text{EATS}}$ interacts with other signals, refer to Section 8.4, "Address Bus Transfer Protocol."

7.2.2.1 Early Address Transfer Start ($\overline{\text{EATS}}$)—Input/Output

$\overline{\text{EATS}}$ may be a unidirectional bused signal or it can be driven point-to-point with the same value.

State Meaning Asserted—$\overline{\text{EATS}}$ indicates to the master driving the address that it will disable its address drivers 1 cycle after sampling the assertion of $\overline{\text{EATS}}$. For all other devices monitoring the address bus, $\overline{\text{EATS}}$ indicates that the address transfer should be sampled 1 cycle after sampling the assertion of $\overline{\text{EATS}}$.

Negated— Indicates that no bus transaction is occurring.

Timing Comments Assertion—$\overline{\text{EATS}}$ can be asserted in the same cycle as $\overline{\text{ABG}}$, or on the cycle following $\overline{\text{ABG}}$. $\overline{\text{EATS}}$ cannot be delayed more than 1 bus clock cycle from $\overline{\text{ABG}}$. If $\overline{\text{EATS}}$ is asserted the same cycle as $\overline{\text{ABG}}$, then the address tenure is one bus clock cycle and a new address transfer can begin every 2 cycles.

If $\overline{\text{EATS}}$ is asserted the cycle following the assertion of $\overline{\text{ABG}}$ and BUSLAEN is deasserted, then the address tenure is 2 bus clocks. If $\overline{\text{EATS}}$ is asserted the cycle following the assertion of $\overline{\text{ABG}}$ and BUSLAEN is asserted, then the address tenure is (2 bus clocks – 1 processor clock). Refer to Section 8.3.9, "Early Address Transfer Start (EATS)," for more information.

Negation—Occurs one bus clock cycle after $\overline{\text{EATS}}$ is asserted.

7.2.3 Address Transfer Signals

The address transfer signals are used to transmit the address and to generate and monitor parity for the address transfer. For a detailed description of how these signals interact, refer to Section 8.4, "Address Bus Transfer Protocol."

7.2.3.1 Address Bus (A[0–63])

The address bus (A[0–63]) consists of 64 signals that are both input and output signals for memory operations.

7.2.3.1.1 Address Bus (A[0–63])—Output

Following are the state meaning and timing comments for the A[0–63] output signals.

State Meaning Asserted/Negated—Represents the physical address (physical address in the architecture specification) of the data to be transferred; see Section 8.9, "Address Bus."

Timing Comments Assertion/Negation—Occurs on the bus clock cycle after $\overline{\text{ABG}}$.

High Impedance—Occurs one bus clock cycle after $\overline{\text{EATS}}$ is negated.

7.2.3.1.2 Address Bus (A[0–63])—Input

Following are the state meaning and timing comments for the A[0–63] input signals.

State Meaning Asserted/Negated—Represents the physical address of a snoop operation.

Timing Comments Assertion/Negation—Occurs on the bus clock cycle after $\overline{\text{ABG}}$.

High Impedance—Occurs one bus clock cycle after $\overline{\text{EATS}}$ is negated.

7.2.3.2 Address Bus Parity (AP[0–2])

The address bus parity (AP[0–2]) signals are both input and output signals reflecting odd-byte parity.

7.2.3.2.1 Address Bus Parity (AP[0–2])—Output

Following are the state meaning and timing comments for the AP[0–2] output signal on the 620.

State Meaning Asserted/Negated—Represents odd parity for each of three sections of the address transaction. Odd parity means that an odd number of bits, including the parity bit, are driven high. The signal assignments correspond to the following:

AP0	A[0–31] (including the address tag)
AP1	A[32–63]
AP2	ATYPE[0–4], ASIZEDATA[0–3], $\overline{\text{ASIZEBURST}}$

For more information, see Section •, "Address Bus Parity—The address bus is parity protected by 3 bits, called AP[0–2]. Table 8-35 defines the coverage for each parity bit.."

Timing Comments Assertion/Negation—The same as A[0–63].

High Impedance—The same as A[0–63].

7.2.3.2.2 Address Bus Parity ($\overline{\text{AP}}$[0–2])—Input

Following are the state meaning and timing comments for the AP[0–2] input signal on the 620.

State Meaning Asserted/Negated—Represents odd parity for each of three bytes of the physical address for snooping. Detected even parity causes the processor to enter the checkstop state or take a machine check exception depending on whether address parity checking is enabled in the HID0 register and the condition of the MSR[ME] bit; see Section 2.1.2.3, "Hardware Implementation-Dependent Register 0 (HID0)."

Timing Comments Assertion/Negation—The same as A[0–63].

7.2.3.3 Address Bus Tag

The ATAG[0–7] is always aligned as A[8–15]. Refer to Section 8.9, "Address Bus," for more information.

7.2.4 Address Transfer Attribute Signals

The transfer attribute signals are a set of signals that further characterize the transfer—such as the size of the transfer, whether it is a read or write operation, and whether it is a burst or single-beat transfer. For a detailed description of how these signals interact, see Section 8.4, "Address Bus Transfer Protocol."

Note that some signal functions vary depending on whether the transaction is a memory access or an I/O access.

7.2.4.1 Address Type (ATYPE[0–4])

The address type (ATYPE[0–4]) signals consist of five input/output signals on the 620. For a complete description of ATYPE[0–4] signals and for address type encodings, see Table 7-1.

7.2.4.1.1 Address Type (ATYPE0–4])—Output

Following are the state meaning and timing comments for the ATYPE[0–4] output signals on the 620.

State Meaning Asserted/Negated—Indicates the type of transfer in progress.

Timing Comments Assertion/Negation/High Impedance—The same as A[0–63].

7.2.4.1.2 Address Type (ATYPE[0–4])—Input

Following are the state meaning and timing comments for the ATYPE[0–4] input signals on the 620.

State Meaning Asserted/Negated—Indicates the type of transfer in progress (see Table 7-1).

Timing Comments Assertion/Negation—The same as A[0–63].

Table 7-1. Address Type Encoding for PowerPC 620 Processor Bus Master

ATYPE[0–4]	Operation	Address/Data Type
00000	Clean	Address-Only
A0010	Write-With-Flush	Address-Data
00100	Flush	Address-Only
00110	Write-With-Kill	Address-Data
00110	Write-With-Clean	Address-Data
01000	SYNC	Address-Only
A1010	Read	A-Only/D-Only
01100	DKill	Address-Only
A1110	RWITM	A-Only/D-Only
10000	EIEIO	Address-Only
10100	External Control Out	Address-Data
11000	TLBIE	Address-Only
11100	External Control In	A-Only/D-Only
00001	LARX-Reserve	Address-Only
A0011	DClaim	Address-Only
01001	TLBSYNC	Address-Only
01101	IKill	Address-Only

Table 7-1. Address Type Encoding for PowerPC 620 Processor Bus Master (Continued)

ATYPE[0–4]	Operation	Address/Data Type
10001	PIO Load Immediate	A-Only/D-Only
10001	PIO Load Last	A-Only/D-Only
10001	PIO Store Immediate.	Address-Data
10001	PIO Store Last	Address-Data
10101	PIO Reply	Address-Only
11101	ReRun	Address-Only
11111	Null	Address-Only
00101 00111 01011 01111 10101 10101 10101 10110 10111 11001 11011	Reserved Treated as a Null operation by the 620 snooper. These codes are reserved for future PowerPC products. Use of these codes should be coordinated with PowerPC if forwards compatibility is desired.	

7.2.4.2 Address Size Data (ASIZEDATA[0–3])

The address bus signal ASIZEDATA[0–3] specifies the size of the data bus transfer. Refer to Section 8.4, "Address Bus Transfer Protocol."

State Meaning Asserted—If $\overline{\text{ASIZEBURST}}$ is deasserted, or a 1, then ASIZEDATA indicates the size of the data transfer in bytes. (Note that the 0 code is a 16 byte size transfer.)

If $\overline{\text{ASIZEBURST}}$ is asserted, or a 0, then ASIZEDATA indicates the size of the data transfer in full data bus width transfers. (Note that the 0 code is 16 bus width size transfers.) Refer to Table 7-2.

Timing Comments Assertion/Negation—The same as A[0–63].

7.2.4.3 Address Size Burst ($\overline{\text{ASIZEBURST}}$)

The address bus signal $\overline{\text{ASIZEBURST}}$ specifies the size of the data bus transfer. Refer to Section 8.4, "Address Bus Transfer Protocol."

State Meaning Asserted—If $\overline{\text{ASIZEBURST}}$ is deasserted, or a 1, then ASIZEDATA indicates the size of the data transfer in bytes. (Note that the 0 code is a 16 byte size transfer.)

If $\overline{\text{ASIZEBURST}}$ is asserted, or a 0, then ASIZEDATA indicates the size of the data transfer in full data bus width transfers. (Note that the 0 code is 16 bus width size transfers.)

The 620 data bus width is quad word sized, which means the unit for ASIZEDATA for burst mode is 16 bytes. Refer to Section 8.12, “Bus Operations,” for a definition of the $\overline{\text{ASIZEBURST}}$ and ASIZEDATA codes that are supported by the 620 for each bus operation. Refer to Table 7-2.

Table 7-2. ASIZEDATA[0–3] and $\overline{\text{ASIZEBURST}}$

$\overline{\text{ASIZEBURST}}$	ASIZEDATA[0–3]	Definition
1	0001–1111, 0000	Transfer size in bytes. 1–15, 16
0	0001–1111, 0000	Transfer size in full data bus width transfers. 1–15, 16

Timing Comments Assertion/Negation—The same as A[0–63].

7.2.4.4 Address Status (ASTATOUT[0–1] and ASTATIN[0–1])

The address status (AStat) consists of the unidirectional signals ASTATOUT[0–1] and ASTATIN[0–1]. For more information, refer to Section 8.4.8, “Address Status In/Out.”

7.2.5 Address Transfer Response Signals

The address transfer termination signals are used to indicate either that the address phase of the transaction has completed successfully or must be repeated, and when it should be terminated.

7.2.5.1 Address Response (ARESPOUT[0–2] and ARESPIN[0–2])

The address response (AResp) consists of the unidirectional signals ARESPOUT[0–2] and ARESPIN[0–2].

AResp provides coherency information (Null, Shared, Modified) and control information (Null, Retry, ReRun).

The function of ARESPOUT and ARESPIN is defined by Section 8.4.9, “Address Response In/Out,” Section 8.17.17, “Master Cache State Transitions Due to Instructions,” and Section 8.17.18, “Snooper Cache State Transitions Due to Bus Operations.”

For more information, refer to Section 8.4.9, “Address Response In/Out.”

7.2.6 L2 Cache Interface

The following sections briefly describe the function for each of the signals of the L2 cache interface.

7.2.6.1 L2 Data (L2DATA[0–127])

L2 cache read and write data signal. The L2 interface only supports 128 bit reads and writes. All cache block transfers are implemented by four L2 interface reads or writes.

7.2.6.2 L2 Address (L2ADDRESS[0–15])

The L2ADDRESS[0–15] signals are used to address the L2 cache SRAM devices. All of the address signals are connected to both the tag and data SRAMs, with the exception of the L2ADDRESS[14–15] signals, which are connected only to the data SRAMs. Following are the state meaning for the L2ADDRESS[0–15] output signals.

State Meaning Asserted/Negated—Represents the address of the data to be transferred to or from the L2 cache.

7.2.6.3 L2 Tag (L2TAG[0–10])

State Meaning Asserted/Negated—Represents the tag associated with the data driven onto the L2DATA[0–127] signals.

7.2.6.4 L2 Tag/Address (L2TAGADD[0–8]

The L2TAGADD[0–8] signals function as address or tag bits based on the L2 cache capacity configuration as described in Section 9.3.1.3.1, "The L2TAGADD Signal."

7.2.6.5 L2 Coherency State (L2COHERENCY[0–1])

The L2COHERENCY signal represents coherency associated with the data on L2DATA[0–127]. Table 7-3 shows the encoding for L2COHERENCY.

Table 7-3. Coherency Encoding

L2COHERENCY [0–1]	L2 State	Implied Data L1 State
00	Invalid	Invalid
01	Shared	Shared or Invalid
10	Exclusive	Shared or Invalid
11	Modified	Modified, Shared, or Invalid

7.2.6.6 L2 Data ECC (L2DATAECC[0–8])

The ECC bits that are generated using L2DATA[0–127]. For a description of the algorithm used, refer to Section 9.4.1, "The ECC Algorithm."

7.2.6.7 L2 Tag ECC (L2TAGECC[0–5])

The ECC bits that are generated using L2TAG[0–10], L2COHERENCY[0–1] and the signals in L2TAGADD[0–8] that are being used as tag. For a description of the algorithm used, refer to Section 9.4.1, "The ECC Algorithm."

7.2.6.8 L2 Enable Signals

The following sections describe the L2 Enable signals implemented on the 620.

7.2.6.8.1 L2 Data Enable (L2DATAENABLE[0–1])

State Meaning Asserted/Negated—When L2B2ENABLE, an L2 SPR bit, is disabled then L2DATAENABLE0 is the active low enable tied to the chip select inputs of all the Data SRAMs. L2DATAENABLE1 is deasserted. When L2B2ENABLE is enabled then L2DATAENABLE0 is the active low enable tied to the chip select inputs of all the bank 0 data SRAMs and is enabled by L2ADDRESS[0]=0.

L2DATAENABLE1 is the active low enable tied to the chip select inputs of all the bank 1 data SRAMs and is enabled by L2ADDRESS[0]=1. Refer to Section 9.3.1.7, "L2B2ENABLE Bit."

7.2.6.8.2 L2 Tag Enable (L2TAGENABLE)

State Meaning Asserted/Negated—L2TAGENABLE is the active low enable tied to the chip select inputs of all the tag SRAMs for all banks.

7.2.6.9 L2 Write Enable Signals

The following sections describe the L2 write enable signals.

7.2.6.9.1 L2 Write Data (L2WRITEDATA)

State Meaning Asserted/Negated—L2WRITEDATA is the active low write enable tied to the write enable inputs of all the data SRAMs.

7.2.6.9.2 L2 Write Tag (L2WRITETAG)

State Meaning Asserted/Negated—L2WRITETAG is the active low write enable tied to the write enable inputs of all the tag SRAMs.

7.2.7 L2 SRAM Output Enable (L2OUTPUTENABLE)

State Meaning Asserted/Negated—The L2 output enable signal is used only for a single bank of SRAMs that need an asynchronous output enable. Note that the asynchronous output enable is provided as a backup strategy. All SRAMs under consideration provide the internal synchronous generation of the SRAM output enable. The L2 output enable is not used if either double bank is enabled or the L2CP is enabled, both of which must have internally generated synchronous output enables. This signal is active low. If the SRAMs used do not require this signal, a resistive pull up to 3.3V should be attached to the signal.

7.2.7.1 L2 Clocks Out/In (L2CLOCK/L2CLOCK and L2CLOCKIN/L2CLOCKIN)

These signals output a differential CMOS clock that is used to clock the L2 cache SRAMs. The L2 interface assumes that the SRAMs are rising edge triggered off of the L2CLOCK. Single-ended clocking may be used, but is not recommended in high speed systems. If L2CLOCK is not being used, then a resistive pull-up to 3.3V should be attached to the signal.

During HRESET and CHECKSTOP the L2 output clocks are not three-stated during the assertion HRESET or CHECKSTOP, but will not function as a periodic clock. The L2 clock outputs during HRESET or CHECKSTOP will always be differential but will oscillate at an indeterminate frequency.

State Meaning Asserted/Negated—These input signals are used by the L2 clock phase-locked loop to lock the external L2 clock with the 620 internal processor clock. The system should guarantee that the delay from L2CLOCK to L2CLOCKIN should be the same as the delay from L2CLOCK to the clock signals on the L2 SRAMs. The L2 clock inputs can be configured to be single ended or differential by the L2CLKPECL SPR bit; refer to Section 9.3.1.6, "L2CLKPECL Bit."

7.2.8 Data Bus Arbitration Signals

Like the address bus arbitration signals, data bus arbitration signals maintain an orderly process for determining data bus mastership. For a detailed description on how these signals interact, see Section 8.3, "Arbitration."

7.2.8.1 Data Bus Grant (DBG)—Input

The data bus grant (DBG) signal is an input signal on the 620. Following are the state meaning and timing comments for the DBG signal.

State Meaning Asserted—DBG is a pulsed grant, asserted for one BUSCLK cycle that indicates the data bus is being granted. DBG can never be asserted more frequently than once every other cycle.

Negated—Indicates that the 620 is not the next potential data bus master.

Timing Comments Assertion—May occur at any time when DBR is asserted.

A data bus master may start driving the data bus when both of the following two conditions are satisfied:

- DBG is asserted.
- DBB is deasserted for two or more BUSCLKs.

If DBB was deasserted for two or more BUSCLKs when DBG is asserted, then the master will start driving the bus on the same edge that DBG is sampled asserted. The data bus grant is pending from the assertion of the data bus grant to the beginning of the data transfer. Refer to Section 8.3.8, “Pending Data Bus Grant Arbitration.”

Negation—May occur at any time to indicate the 620 cannot assume data bus mastership.

7.2.8.2 Data Bus Request ($\overline{\text{DBR}}$)

The address bus request ($\overline{\text{DBR}}$) signal is a point-to-point signal from the 620 to the arbiter. Following are the state meaning and timing comments for the $\overline{\text{DBR}}$ signal.

State Meaning Asserted—Indicates that the 620 is requesting mastership of the data bus.

The request is pending for as long as the request is asserted. This is called a level request implementation, as opposed to a pulsed request implementation, and only one request can be pending at any given time by a single 620.

Negated—Indicates that the 620 is not requesting the data bus.

Timing Comments Assertion—Occurs when a data bus transaction is needed and the 620 does not have a qualified data bus grant.

Negation—Occurs when the data bus grant is received, even if another transaction is pending.

7.2.9 Data Transfer Signals

Like the address transfer signals, the data transfer signals are used to transmit data and to generate and monitor parity for the data transfer. For a detailed description of how the data transfer signals interact, see Section 8.5, “Data Bus Transfer Protocol.”

7.2.9.1 Data Bus (DH[0–63], DL[0–63])

The data bus (DH[0–63] and DL[0–63]) consists of 128 signals that are both input and output on the 620. Following are the state meaning and timing comments for the data bus signals.

State Meaning The data bus has two halves—data bus high (DH) and data bus low (DL). See Table 7-4 for the data bus lane assignments.

Timing Comments The data bus is driven once for noncached transactions and four times for cache transactions (bursts).

Table 7-4. Data Bus Lane Assignments

Data Bus Signals	Byte Lane
DH[0–7]	0
DH[8–15]	1
DH[16–23]	2
DH[24–31]	3
DH[32–39]	4
DH[40–47]	5
DH[48–55]	6
DH[56–63]	7
DL[0–7]	8
DL[8–15]	9
DL[16–23]	10
DL[24–31]	11
DL[32–39]	12
DL[40–47]	13
DL[48–55]	14
DL[56–63]	15

7.2.9.1.1 Data Bus (DH[0–63], DL[0–63])—Output

Following are the state meaning and timing comments for the DH and DL output signals.

State Meaning Asserted/Negated—Represents the state of data during a data write. Byte lanes not selected for data transfer will not supply valid data.

Timing Comments Assertion/Negation—A data bus master may start driving the data bus when both of the following two conditions are satisfied:

- $\overline{\text{DBG}}$ was asserted.
- $\overline{\text{DBB}}$ was deasserted for two or more BUSCLKs.

If $\overline{\text{DBB}}$ was deasserted for two or more BUSCLKs when $\overline{\text{DBG}}$ is asserted, then the master will start driving the bus on the same edge that $\overline{\text{DBG}}$ is sampled asserted.

High Impedance—Occurs on the bus clock cycle after the final assertion of DVAL0 or DVAL1.

7.2.9.1.2 Data Bus (DH[0–63], DL[0–63])—Input

Following are the state meaning and timing comments for the DH and DL input signals.

State Meaning Asserted/Negated—Represents the state of data during a data read transaction.

Timing Comments Assertion/Negation—Data must be valid on the same bus clock cycle that $\overline{\text{DVAL0}}$ or $\overline{\text{DVAL1}}$ is asserted.

7.2.9.2 Data Bus Parity (DP[0–7])

The eight data bus parity (DP[0–7]) signals on the 620 are both output and input signals.

7.2.9.2.1 Data Bus Parity (DPH[0–7], DPL[0–7], DPCNTL)—Output

Following are the state meaning and timing comments for the DP output signals.

State Meaning Asserted/Negated—Represents odd parity for each of 17 bytes of data write transactions. Odd parity means that an odd number of bits, including the parity bit, are driven high. The signal assignments are listed in Table 7-5.

Timing Comments Assertion/Negation—The same as DH[0–63].

High Impedance—The same as DL[0–63].

Table 7-5. DP[0–7] Signal Assignments

Signal Name	Signal Assignments
DPH0	DH[0–7]
DPH1	DH[8–15]
DPH2	DH[16–23]
DPH3	DH[24–31]
DPH4	DH[32–39]
DPH5	DH[40–47]
DPH6	DH[48–55]
DPH7	DH[56–63]
DPL0	DL[0–7]
DPL1	DL[8–15]
DP2L	DL[16–23]
DPL3	DL[24–31]
DPL4	DL[32–39]
DPL5	DL[40–47]
DPL6	DL[48–55]
DPL7	DL[56–63]
DPCNTL	DTAG[0–7], $\overline{\text{DBB}}$, $\overline{\text{DCACHE}}$

7.2.9.2.2 Data Bus Parity (DPH[0–7], DPL[0–7], DPCNTL)—Input

Following are the state meaning and timing comments for the DP input signals.

State Meaning Asserted/Negated—Represents odd parity for each byte of read data. Detected even parity causes the processor to enter the checkstop state, or take a machine check exception depending on whether data parity checking is enabled in the HID0 register and the condition of the MSR[ME] bit; see Section 2.1.2.3, "Hardware Implementation-Dependent Register 0 (HID0)."

Timing Comments Assertion/Negation—The same as DL[0–31].

7.2.9.3 Data Cache ($\overline{\text{DCACHE}}$)

The function of the $\overline{\text{DCACHE}}$ signal is to distinguish cache (intervention) data from memory data. The $\overline{\text{DCACHE}}$ is both an input and output signal.

7.2.9.3.1 Data Cache ($\overline{\text{DCACHE}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{DCACHE}}$ output signal.

State Meaning Asserted—Data is being provided from the data cache.

Negated—Data is being provided from memory.

Timing Comments Asserted/Negated—$\overline{\text{DCACHE}}$ is valid only when $\overline{\text{DVAL}}$ is asserted. $\overline{\text{DCACHE}}$ is driven asserted only for the first data that is transferred and deasserted for all remaining data that are transferred.

7.2.9.3.2 Data Cache ($\overline{\text{DCACHE}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{DCACHE}}$ input signal.

State Meaning Asserted—Indicates that data is being provided from the data cache.

Negated—Data is being provided from memory.

Timing Comments Asserted/Negated—$\overline{\text{DCACHE}}$ is valid only when $\overline{\text{DVAL}}$ is asserted. $\overline{\text{DCACHE}}$ is sampled asserted only for the first data that is transferred.

7.2.9.4 Data Bus Tag (DTAG[0–7])

DTAG[0–7] is driven or sampled by the 620 during a read or write operation. The DTAG[0–7] signals correlate data bus transactions with the associated address tenure. Refer to Section 8.5.4, "Data Bus Tag."

State Meaning Asserted/Negated—A 620 with a read operation waiting for data will snoop the data bus tag and latch each data beat with a matching tag. The 620 does not support the interleaving of data tags with other bus master data tenures; when a transfer is initiated with a given data tag, all data beats of the transfer must be completed together.

Timing Comments Asserted/Negated—DTAG[0–7] is driven or sampled for the duration of the data bus tenure.

7.2.10 Data Valid ($\overline{\text{DVAL}}$[0–1])

The data bus master samples or drives $\overline{\text{DVAL}}$[0–1] to determine or indicate whether this cycle has valid data, thereby implementing flow control by the driving data bus master. A device receiving data from the data bus master has no means to control data flow. If a memory bus device does not have the required buffer space for the transfer then it retries the address. If a master has requested data, it must provide buffer space for the data requested.

7.2.10.0.1 Data Valid ($\overline{\text{DVAL}}$[0–1])—Output

Following are the state meaning and timing comments for the $\overline{\text{DVAL}}$[0–1] output signals.

State Meaning Asserted—Indicates that data is valid for this bus clock cycle.

Negated—Indicates that data is not valid this bus clock cycle, or that data flow control is being exercised. Note that $\overline{\text{DVAL}}$[0–1] can be deasserted during a data tenure when a bus master providing data needs to delay data transfer for one or more cycles (for example, when a correctable data error occurs), or when the bandwidth of the memory system is not equivalent to the data bus, and the $\overline{\text{DVAL}}$[0–1] signals are used to indicate when memory data is valid on the bus.

Timing Comments Asserted/Negated—$\overline{\text{DVAL}}$[0–1] is asserted by the 620 for all cycles during a data bus tenure. $\overline{\text{DVAL}}$[0–1] is asserted on the first bus cycle of a data tenure and held asserted until all contiguous data beats are completed.

The $\overline{\text{DVAL}}$[0–1] signals are driven as follows:

- $\overline{\text{DVAL}}$0 will be asserted if $\overline{\text{DVAL}}$0 has not been asserted by another bus master in the previous two bus clock cycles.
- $\overline{\text{DVAL}}$1 will be asserted if $\overline{\text{DVAL}}$0 has been asserted by another bus master in the two previous bus clock cycles. Note that bus devices other than the 620 have the option of not implementing a driver for $\overline{\text{DVAL}}$1, and may choose to wait one extra bus clock cycle in order to avoid $\overline{\text{DVAL}}$0 contention.
- The $\overline{\text{DVAL}}$[0–1] signals can switch between data tenures, but not within a data tenure.

7.2.10.0.2 Data Valid ($\overline{\text{DVAL}}$[0–1])—Input

Following are the state meaning and timing comments for the $\overline{\text{DVAL}}$[0–1] input signals.

State Meaning Asserted— Indicates that data is valid for this bus clock cycle. Data should be considered valid if either $\overline{\text{DVAL}}$0 or $\overline{\text{DVAL}}$1 are asserted during a data tenure.

Negated— Indicates that data is not valid for this bus clock cycle. If negation occurs during multiple-beat data transfer indicates that flow control is being exercised.

Timing Comments Asserted/Negated—$\overline{\text{DVAL}}$[0–1] is sampled for all cycles of a data bus tenure.

7.2.11 Data Transfer Termination Signals

Data termination signals are required after each data beat in a data transfer. Note that in a single-beat transaction, the data termination signals also indicate the end of the tenure, while in burst accesses, the data termination signals apply to individual beats and indicate the end of the tenure only after the final data beat.

7.2.11.1 Data Bus Busy ($\overline{\text{DBB}}$)

The $\overline{\text{DBB}}$ signal indicates that the data bus is performing a transfer operation.

7.2.11.1.1 Data Bus Busy ($\overline{\text{DBB}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{DBB}}$ output signal.

State Meaning Asserted—Multiple-beat data transfer is in progress.

Negated—Last beat of a multiple-beat data transfer is progress, or data bus is idle.

Timing Comments Asserted/Negated—$\overline{\text{DBB}}$ is asserted for all but the last data of the last transfer of a data bus tenure. $\overline{\text{DBB}}$ can also be thought to indicate that this is not the last BUSCLK of this data bus tenure.

7.2.11.1.2 Data Bus Busy ($\overline{\text{DBB}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{DBB}}$ input signal.

State Meaning Asserted—Multiple-beat data transfer is in progress.

Negated—Last beat of a multiple-beat data transfer is progress, or data bus is idle.

Timing Comments Asserted/Negated—$\overline{\text{DBB}}$ is sampled asserted for all but the last data of the last transfer of a data bus tenure.

7.2.12 Data Error ($\overline{\text{DERR}}$)

The $\overline{\text{DERR}}$ signal indicates that the current data transfer contains an error. $\overline{\text{DERR}}$ is an input and output signal.

7.2.12.0.1 Data Error ($\overline{\text{DERR}}$)—Output

Following are the state meaning and timing comments for the $\overline{\text{DERR}}$ output signal.

State Meaning Asserted— $\overline{\text{DERR}}$ is asserted for the following reasons:

- Snooper and push or intervention error—The 620 as a snooper for a push or intervention detects an L1 parity error while moving data from the L1 or detects an uncorrectable L2 error while moving data from the L2.
- Master and Replacement Copyback Error—The 620 for a replacement copyback detects an L1 parity error while moving data from the L1 or detects an uncorrectable L2 error while moving data from the L2.

Negated— Indicates that no data error occurred.

Timing Comments Asserted/Negated—$\overline{\text{DERR}}$ is driven by the data bus master two bus clock cycles following the assertion of $\overline{\text{DVAL}}$ to indicate whether the data transfer contains an error.

7.2.12.0.2 Data Error ($\overline{\text{DERR}}$)—Input

Following are the state meaning and timing comments for the $\overline{\text{DERR}}$ input signals.

State Meaning Asserted— The 620 will sample $\overline{\text{DERR}}$ asserted for the following reasons:

- The bus master providing the data has detected an error, e.g. 2-bit ECC memory error.
- A PIO load detected an error and the PIO master should expect a PIO load reply. Refer to Section 8.16, "PIO Load and Store Bus Operations."

Negated—Indicates that no data error occurred.

Timing Comments Asserted/Negated—$\overline{\text{DERR}}$ is sampled by the 620 two bus clock cycles following the assertion of $\overline{\text{DVAL}}$ to indicate whether the data transfer contains an error.

7.2.13 System Interrupt, Checkstop, and Reset Signals

Most of the system interrupt, checkstop, and reset signals are input signals that indicate when exceptions are received, when checkstop conditions have occurred, and when the 620 must be reset. The 620 generates the output signal, $\overline{\text{CHECKSTOP}}$, when it detects a checkstop condition.

7.2.13.1 Interrupt ($\overline{\text{INT}}$)—Input

The $\overline{\text{INT}}$ signal is input only. Following are the state meaning and timing comments for the $\overline{\text{INT}}$ signal.

State Meaning Asserted—The 620 initiates an interrupt if MSR[EE] is set; otherwise, the 620 ignores the interrupt. To guarantee that the 620 will take the external interrupt, the $\overline{\text{INT}}$ signal must be held active until the 620 takes the interrupt; otherwise, the 620 will take an external interrupt depending on whether the MSR[EE] bit was set while the $\overline{\text{INT}}$ signal was held active.

Negated—Indicates that normal operation should proceed.

Timing Comments Assertion/Negation—$\overline{\text{INT}}$ is level sensitive, and must be asserted until the interrupt is taken.

$\overline{\text{INT}}$ is synchronized through two registers clocked by the BUSCLK, so the $\overline{\text{INT}}$ signal can be treated asynchronously or synchronously.

If deterministic cycle sequencing is required (for example, in multiple processor systems operating in lock step), the $\overline{\text{INT}}$ signal should be asserted and negated synchronously with the SYSCLK signal.

7.2.13.2 System Management Interrupt ($\overline{\text{SMI}}$)—Input

The system management interrupt ($\overline{\text{SMI}}$) signal is input only. Following are the state meaning and timing comments for the $\overline{\text{SMI}}$ signal.

State Meaning Asserted—$\overline{\text{SMI}}$, when asserted and enabled by MSR[EE], will cause a system management interrupt.

If $\overline{\text{SMI}}$ is asserted and enabled when $\overline{\text{HRESET}}$ or $\overline{\text{SRESET}}$ is deasserted, the 620 will take the interrupt.

Negated—Indicates that normal operation should proceed.

Timing Comments Assertion—$\overline{\text{SMI}}$ is level sensitive, and must be asserted until the interrupt is taken.

$\overline{\text{SMI}}$ is synchronized through two registers clocked by the BUSCLK, so the $\overline{\text{SMI}}$ signal can be treated asynchronously or synchronously.

Negation—Should not occur until interrupt is taken.

If deterministic cycle sequencing is required (for example, in multiple processor systems operating in lock step), the $\overline{\text{SMI}}$ signal should be asserted and negated synchronously with the SYSCLK signal.

7.2.13.3 Machine Check Interrupt ($\overline{\text{MCP}}$)—Input

The machine check interrupt ($\overline{\text{MCP}}$) signal is input only on the 620. Following are the state meaning and timing comments for the $\overline{\text{MCP}}$ signal.

State Meaning Asserted—$\overline{\text{MCP}}$, when asserted and enabled by MSR[ME] and HID0[0], will cause a machine check interrupt.

If $\overline{\text{MCP}}$ is asserted and enabled when $\overline{\text{HRESET}}$ or $\overline{\text{SRESET}}$ is deasserted, the 620 will take the interrupt.

Negated—Indicates that normal operation should proceed.

Timing Comments Assertion—$\overline{\text{MCP}}$ is synchronized through 2 registers clocked by the BUSCLK, so the $\overline{\text{MCP}}$ signal can be treated asynchronously or synchronously.

Negation—May be negated two bus cycles after assertion.

If deterministic cycle sequencing is required (for example, in multiple processor systems operating in lock step), the $\overline{\text{MCP}}$ signal should be asserted and negated synchronously with the SYSCLK signal.

7.2.13.4 Reset Signals

There are two reset signals on the 620—hard reset ($\overline{\text{HRESET}}$) and soft reset ($\overline{\text{SRESET}}$). Descriptions of the reset signals are as follows:

7.2.13.4.1 Hard Reset ($\overline{\text{HRESET}}$)—Input

The hard reset ($\overline{\text{HRESET}}$) signal is input only and must be used at power-on to properly reset the processor. Following are the state meaning and timing comments for the $\overline{\text{HRESET}}$ signal.

State Meaning Asserted—Initiates a complete hard reset operation when this input transitions from asserted to negated. Causes a reset exception as described in Section 4.6.1, "System Reset Exception (0x00100)." Output drivers are released to high impedance after the assertion of $\overline{\text{HRESET}}$.

Negated—Indicates that normal operation should proceed.

Timing Comments Assertion—The minimum assertion at power-up occurs when the system must assert $\overline{\text{HRESET}}$ for the longer of the following two conditions:

- As long as it takes the weakly pulled bus signals $\overline{\text{DVAL}}$ and $\overline{\text{DBB}}$ to be pulled deasserted. See Section 8.5.5, "Data Bus Busy (DBB)," and Section 8.5.6, "Data Valid (DVAL[0–1])."

- It takes the BUSCLK PLL 2 ms to lock and scan reset state.

The minimum assertion at other times occurs in 1000 processor clocks, the amount of time that it takes for the reset state to be scanned.

$\overline{\text{HRESET}}$ is synchronized through two registers clocked by the BUSCLK, so the $\overline{\text{HRESET}}$ signal can be treated asynchronously or synchronously.

Negation—May occur any time after the minimum reset pulse width has been met.

If deterministic cycle sequencing is required (for example, in multiple processor systems operating in lock step), the $\overline{\text{HRESET}}$ signal should be asserted and negated synchronously with the BUSCLK signal. The $\overline{\text{HRESET}}$ signal has additional functionality in certain test modes.

7.2.13.4.2 Soft Reset ($\overline{\text{SRESET}}$)—Input

The soft reset ($\overline{\text{SRESET}}$) signal is input only. Following are the state meaning and timing comments for the $\overline{\text{SRESET}}$ signal.

State Meaning Asserted—The asserting or falling edge of $\overline{\text{SRESET}}$ tells the processor to cancel any outstanding operations and to vector to a base location to begin fetching instructions. A global cancel signal may internally reset portions of the logic, but the internal latches are not explicitly reset. No initiation of BIST is associated with soft reset.

Negated—Indicates that normal operation should proceed.

Timing Comments Assertion—May be asserted at any time.

$\overline{\text{SRESET}}$ is synchronized through two registers clocked by the BUSCLK, so the $\overline{\text{SRESET}}$ signal can be treated asynchronously or synchronously.

Negation—May be negated after one bus clock cycle if synchronous, or after two bus clocks if asynchronous.

If deterministic cycle sequencing is required (for example, in multiple processor systems operating in lock step), the $\overline{\text{SRESET}}$ signal should be asserted and negated synchronously with the SYSCLK signal. The $\overline{\text{SRESET}}$ signal has additional functionality in certain test modes.

7.2.14 Processor Configuration Signals

The signals described in this section provide inputs for controlling the 620's time base, signal drive capabilities, L2 cache access, and PLL configuration, along with descriptions of the output signals that indicate that a memory reservation has been set, and that the 620's internal clocking has stopped.

7.2.14.1 Checkstop ($\overline{\text{CHECKSTOP}}$)—Input/Output

The checkstop signal is input/output on the 620. Following are the state meaning and timing comments for the $\overline{\text{CHECKSTOP}}$ signal.

State Meaning Asserted—The $\overline{\text{CHECKSTOP}}$ signal can also be asserted by an external device. The 620 will respond by entering the checkstop state and stopping the internal processor clock.

The $\overline{\text{CHECKSTOP}}$ signal is an open-drain input/output that must be pulled high outside of the 620.

If the checkstop condition is detected as an input from an external device, then all outputs, except L2CLOCK/$\overline{\text{L2CLOCK}}$, and including the $\overline{\text{CHECKSTOP}}$ signal, will be three-stated. L2CLOCK/$\overline{\text{L2CLOCK}}$ will function normally. The 620 outputs to be three-stated will go three-stated asynchronously from detecting the checkstop condition.

If the checkstop condition is detected internally, then all outputs except the $\overline{\text{CHECKSTOP}}$ signal and L2CLOCK/$\overline{\text{L2CLOCK}}$ will be three-stated. L2CLOCK/$\overline{\text{L2CLOCK}}$ will function normally. The 620 outputs to be three-stated will go three-stated asynchronously from detecting the checkstop condition.

Negated—Indicates that normal operation should proceed.

Timing Comments Assertion—$\overline{\text{CHECKSTOP}}$, as an input, is synchronized through two registers clocked by the BUSCLK, so the $\overline{\text{CHECKSTOP}}$ signal can be treated asynchronously or synchronously. $\overline{\text{CHECKSTOP}}$, as an output, is asynchronous.

The $\overline{\text{CHECKSTOP}}$ signal as an output can not be BUSCLK synchronized because the internal PCLK is immediately halted by the check stop condition, potentially before the next BUSCLK and PCLK occurs.

Negation—Occurs upon assertion of $\overline{\text{HRESET}}$.

7.2.14.2 Reservation ($\overline{\text{RSRV}}$)—Output

The reservation ($\overline{\text{RSRV}}$) signal is output only on the 620. Following are the state meaning and timing comments for the $\overline{\text{RSRV}}$ signal.

State Meaning Asserted/Negated—Represents the state of the reservation coherency bit in the reservation address register that is used by the **lwarx** and **stwcx.** instructions.

Timing Comments Assertion/Negation—Occurs synchronously one bus clock cycle after the execution of an **lwarx** instruction that sets the internal reservation condition.

Note: The 620 does not require for a system to use this signal in order for **lwarx** and **stwcx.** instructions to work in a multi-level cache hierarchy.

7.2.15 JTAG Bus Signals

In accordance with the IEEE 1149.1 standard, the 620 supports the standard JTAG bus I/Os as shown in Table 7-6.

Table 7-6. JTAG Bus Signals

Signal Name	Function	Input/Output
JTAG_TCK	JTAG Clock —Used to control the internal TAP controller state transitions and used to clock data in and out during JTAG Boundary Scan.	Input
JTAG_TMS	Mode Select —This input in conjunction with transitions of JTAG_TCK causes state changes in the TAP.	Input
JTAG_TRST	Reset —Asynchronous input to reset TAP.	Input
JTAG_TDI	Serial Input —Source of all instructions and input data for the TAP.	Input
JTAG_TDO	Serial Output —Enabled during prescribed times to provide desired output of data from the TAP.	Output

7.2.16 Miscellaneous Signals

The following sections provide information on miscellaneous signals implemented on the 620.

7.2.16.1 Time Base Enable ($\overline{\text{TBENABLE}}$)—Input

The time base enable ($\overline{\text{TBENABLE}}$) signal is input only on the 620. Following are the state meanings and timing comments for the $\overline{\text{TBENABLE}}$ signal.

State Meaning Asserted—$\overline{\text{TBENABLE}}$, when asserted, enables the time base increment function. The unit for the 620 time base is the period of the BUSCLK.

Negated—Indicates the time base should stop clocking.

Timing Comments Assertion/Negation—May occur on any cycle. $\overline{\text{TBENABLE}}$ is synchronized through two registers clocked by the BUSCLK, so the $\overline{\text{TBENABLE}}$ signal can be treated asynchronously or synchronously.

7.2.16.2 Analog VDD (AVDD)—Input

The analog VDD signal is an input for supplying a stable voltage to the on-chip phase-locked loop clock generator. For more information about the electrical requirements of the AVDD input signal, refer to the 620 electrical specification.

7.2.17 Clock Signals

The clock signal inputs of the 620 determine the system clock frequency and provide a flexible clocking scheme that allows the processor to operate at an integer multiple of the system clock frequency. An analog voltage input signal is provided to supply stable power for the internal PLL clock generator. The 620 incorporates phase-locked-loop logic to generate the internal processor core and interface clocks with minimal skew relative to the BUSCLK input.

Refer to the 620 hardware specifications for exact timing relationships of the clock signals.

7.2.17.1 Bus Clock (BUSCLK and $\overline{\text{BUSCLK}}$)—Inputs

State Meaning Asserted/Negated—BUSCLK is the system BUSCLK input to the 620. An internal PLL is designed to synchronize the internal processor clock to the BUSCLK at the package input. Synchronization is rising edge to rising edge. Care should be taken to ensure that minimal noise is present on this input to avoid high frequency noise coupling into the package.

The BUSCLK input signals are configured for CMOS or GTL logic through the configuration of the BUSCLKGTL signal. When BUSCLKGTL is asserted the BUSCLK signals are configured for GTL drive levels. If single-ended GTL drive is being used, the $\overline{\text{BUSCLK}}$ signal should be connected to the GTL reverence voltage. If BUSCLKGTL is negated (selecting CMOS logic drive levels), $\overline{\text{BUSCLK}}$ should be weakly pulled low.

Timing Comments Duty cycle—Refer to the 620 hardware specifications for timing comments.

7.2.17.2 Phase-Locked Loop Bypass (PLLBypass)—Input

State Meaning Asserted/Negated—The PLLBypass signal, when asserted high, disables the internal PLL and puts the 620 into a low frequency test/emulation mode. For normal chip operation this signal should be weakly pulled low. For correct low frequency test/emulation operation an additional clock signal, LF_Test_Clk, must be provided.

7.2.17.3 Low Frequency Test Clock (LF_Test_Clk)—Input

State Meaning Asserted/Negated—The LF_Test_Clk (low frequency test clock) is an additional clock signal required in the low frequency test/emulation mode (PLLBypass active high). When PLLBypass is deasserted LF_Test_Clk should be weakly pulled low with a resistor, or driven actively high or low so as not to bring unnecessary switching noise into the chip.

Chapter 8
System Interface Operation

This chapter describes the PowerPC 620 microprocessor bus interface and its operation. It shows how the 620 signals, defined in Chapter 7, "Signal Descriptions," interact to perform address and data transfers.

8.1 Overview

There are three functional sections to the bus protocol—arbitration, address and data. Arbitration is described first to illustrate how address and data bus transfers are and are not related. The address bus is described next because any data transfer is directly or indirectly resultant from an address bus transfer. The data bus is the last of the three to be described because it depends on both arbitration and the address bus.

Figure 8-1 and Figure 8-2 show a high-level block diagram of the 620 bus interface unit implementation for the internal data and address paths that connect from the L1 caches to the L2 and system bus interfaces. The L2 controller is included in these diagrams because it is part of the bus interface unit logic design.

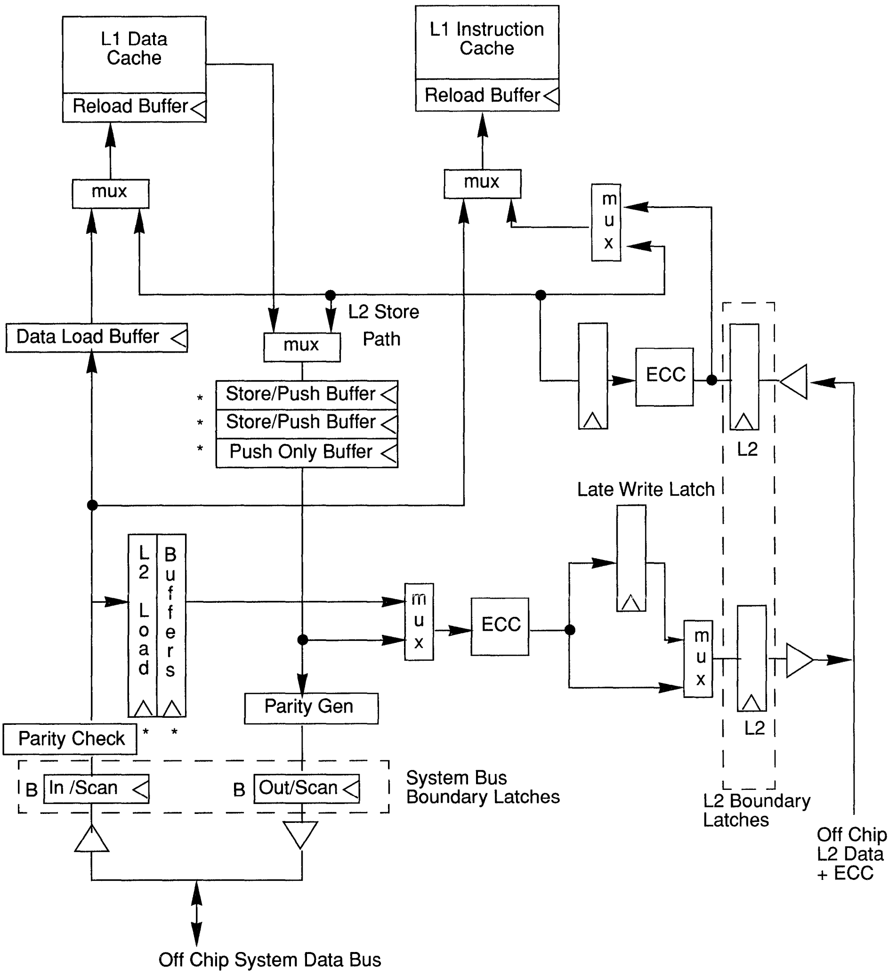
As noted in Figure 8-1, the internal data paths are 16 bytes or 128 bits wide. All of the data buffers are implemented as four 16-byte registers or one 64-byte cache line per buffer (note that the 620 cache line size is fixed at 64 bytes). Each buffer can be written to in four-beat bursts or in individual bytes for non-burst transactions.

If DX mode (64-bit wide data bus option) is used, then eight beats of 64 bits of system data is required to fill the data load buffer. In this mode, two 64-bit quantities will be collected before each 128-bit wide register is written. Likewise, outgoing stores to the system data bus are sent out as eight beats of 64 bits to form the store back buffers.

As shown in Figure 8-1, there is one reload buffer per L1 cache, one data load buffer, two L2 load buffers and three store back buffers. The store back buffers are used for system bus stores from either the L1 cache or the L2 cache and for L2 stores from the L1 data cache. All three store back buffers can be used for cache line pushes, but only two of the three buffers can be used for store data.

On the L2 interface, ECC generation and checking is performed on the L2 data. ECC checking on cache lines going to the L1 instruction cache are optional. Another feature of the L2 interface shown here is the Late Write latch. If the Late Write mode is selected in the L2 configuration register (L2CR), then outgoing data to the L2 will be written one L2 clock cycle later than the L2 address (without this mode, address and data are written to the L2 in the same L2 clock cycle).

Parity generation and checking is provided on the system bus interface. Boundary scan latches as well as some internal logic are clocked by the L2 interface clock for the L2 and by the system bus clock on the system bus. Internally, most of the logic runs off of the internal processor clock. An analog PLL controls the system bus to processor clock ratios and a digital PLL on the L2 interface controls the processor to L2 clock ratio.



Notes:

1. * = Cache line size buffers (64 Byte). All other latches are 16 Byte size. All internal data paths are 16 Byte wide to/from caches.
2. Symbol "L2" clocked by L2 interface. "B" clocked by system bus interface. All other latches clocked by processor clock.

Figure 8-1. PowerPC 620 Microprocessor Data Path

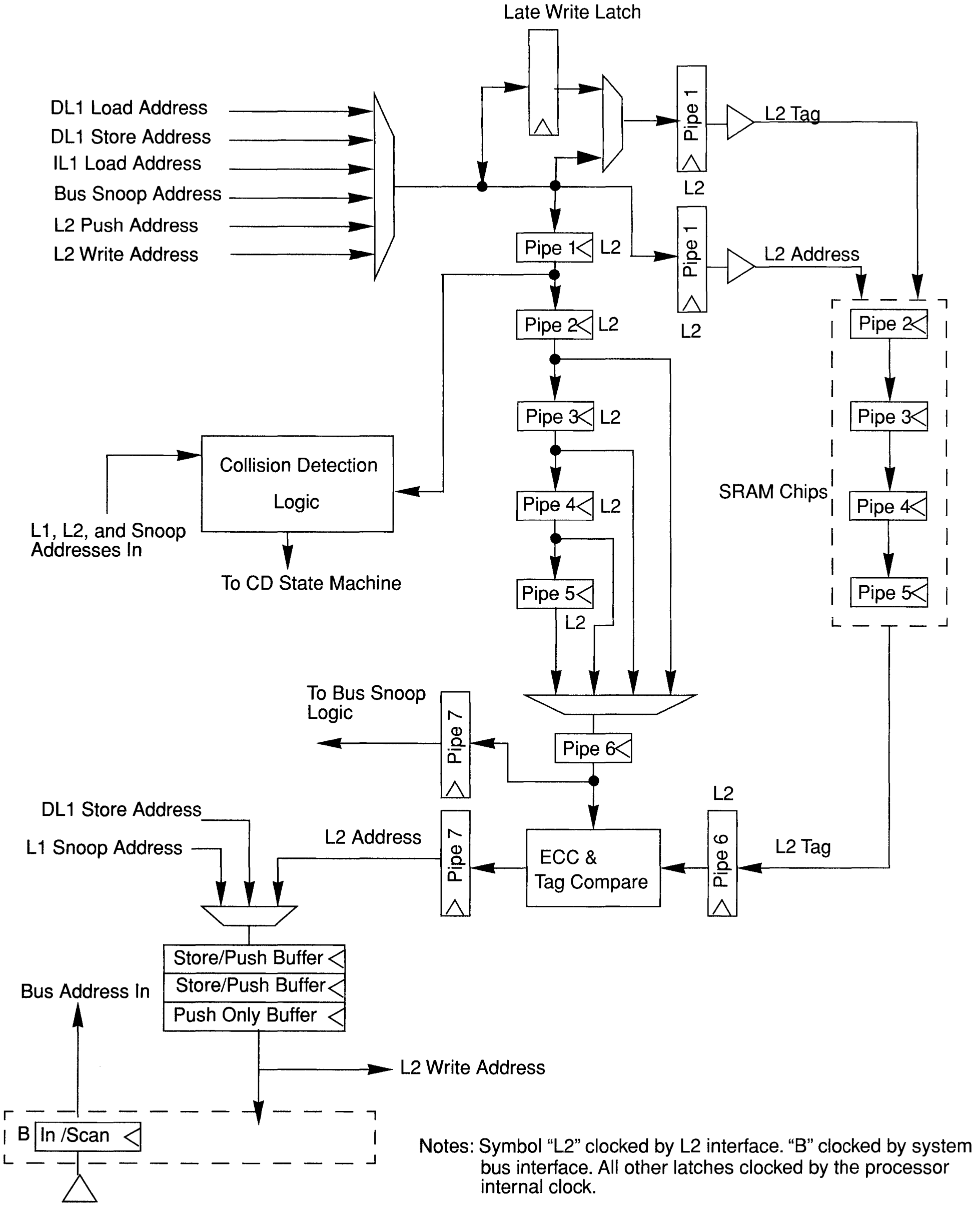
Figure 8-2 shows the 620 high-level address paths for the system bus interface and the L2 interface.

Several levels of muxes exist to select which address source will get to read from or write to the L2. Pipeline stages 3, 4 and 5 can be bypassed depending or whether single-, double-, triple- or quad-synchronous SRAMs are used.

The processor-to-L2 clock ratios are selected by the parameter L2RATIOSR in the L2CR. The L2RATIOSR configures the SRAM access time in terms of number of processor clocks where an L2RATIOSR value of two would indicate a ratio of two processor clocks to one L2 clock. SRAMs with appropriate access times would be selected accordingly.

Pipeline stages 1 and 6 also double up as boundary scan latches for tag and address going off of the 620.

Not shown in detail here, but a very important topic for the 620, is the collision detection logic that controls sequencing of incoming and outgoing bus transactions to the system bus and L2 interface in time to prevent collisions from occurring between different transactions in progress. For more detail on this topic, refer to Section 8.18, "Address Collision Detection and Handling (CD)."



Notes: Symbol "L2" clocked by L2 interface. "B" clocked by system bus interface. All other latches clocked by the processor internal clock.

Figure 8-2. PowerPC 620 Microprocessor Address Path

8.1.1 Timing Diagrams

The following sets of timing diagrams are designed to illustrate the 620 bus protocol. In all of the timing diagrams, BUSTLAR=3 and BUSRESPTEN=2. The bus time latency address to response (BUSTLAR) and BUSRESPTEN are set at boot time and remain fixed system parameters thereafter. System implementations using the bus protocol could choose different values based on their needs.

All data is clocked on the rising edge of BUSCLK. Figure 8-3 shows an example of two burst reads with data returning out of order. Masters A and B each request the address bus for two cycles. These request signals are really separate signals on each master, but are shown on one line here to save space. An $\overline{\text{ABG}}$ will be granted by the arbiter to each of the 620 masters. The master must hold the $\overline{\text{ABR}}$ asserted until it has sampled the valid $\overline{\text{ABG}}$.

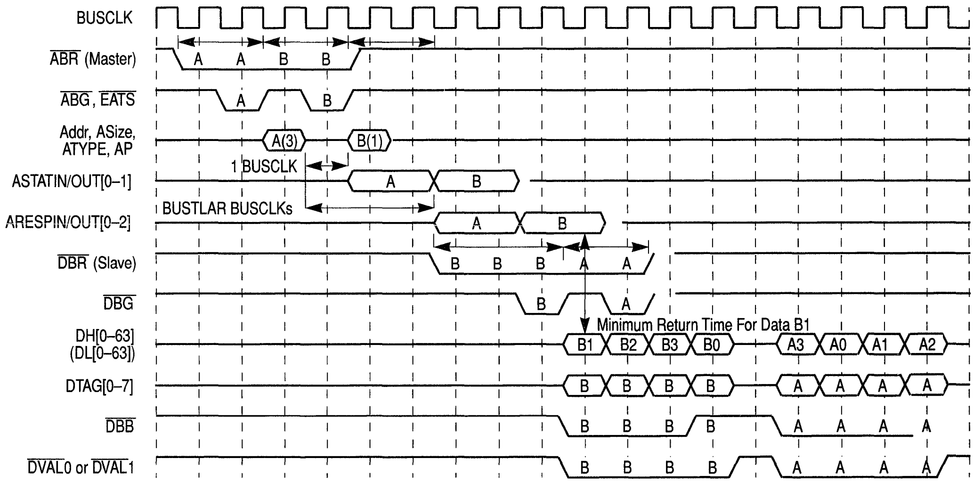


Figure 8-3. Two Burst Reads—A and B with Data Returning Out of Order

In this example, the maximum address bus bandwidth is used, which is one valid address every other bus clock. The 620 will expect read data from the memory controller in critical quad-word first. The notation A(3) means that master A is asking for the third quad-word of the burst read (cache line read) first. Master B is asking for the first quad-word of the cache line. Quad-words are numbered 0–3 for the four beat burst transaction.

Address duration in bus cycles is determined by when the arbiter drives $\overline{\text{EATS}}$ to all devices. In this example, $\overline{\text{EATS}}$ is driven in the same cycle as $\overline{\text{ABG}}$. The master will stop driving the address on the bus one cycle after $\overline{\text{EATS}}$ is sampled. From the time that the address bus is sampled to the beginning of when ASTATOUT is driven is fixed at one cycle in the 620 bus protocol. BUSRESPTEN = 2 indicates that both ASTATOUT/IN and ARESPOUT/IN have a total duration of two bus cycles. BUSTLAR is the snoop response latency from when the address is sampled to the beginning of when ARESPOUT is driven.

The $\overline{\text{DBR}}$ and $\overline{\text{DBG}}$ are unique to each device requesting the data bus. They follow the same rules as the $\overline{\text{ABR}}$ and $\overline{\text{ABG}}$ signals.

Once the data bus has been granted, the device supplying the data will begin driving the data bus in the next cycle. Data is supplied critical quad-word first. $\overline{\text{DBB}}$ is asserted for all but the last quad-word, while the $\overline{\text{DVAL}}$ signals are asserted for the whole tenure on the data bus. Because of the tagged transactions, memory controllers may supply data out of order from the original requests (in this case, B is back before A).

This example shows the fastest that data could be sampled by the master. Data can be sampled no earlier than the cycle when ARESPIN is sampled due to bus tag allocation/deallocation rules.

Figure 8-4 is an example of three address-data burst write transactions from the same or multiple masters (A, B, C). The address bus transaction is the same as for read operations, but the $\overline{\text{DBR}}$ goes out in the same cycle as the $\overline{\text{ABR}}$. The earliest that the bus arbiter can issue the $\overline{\text{DBG}}$ is in the same cycle as $\overline{\text{EATS}}$. For 620 bus writes, the quad-words do not get written critical quad-word first (even if the address specifies a quad-word other than zero).

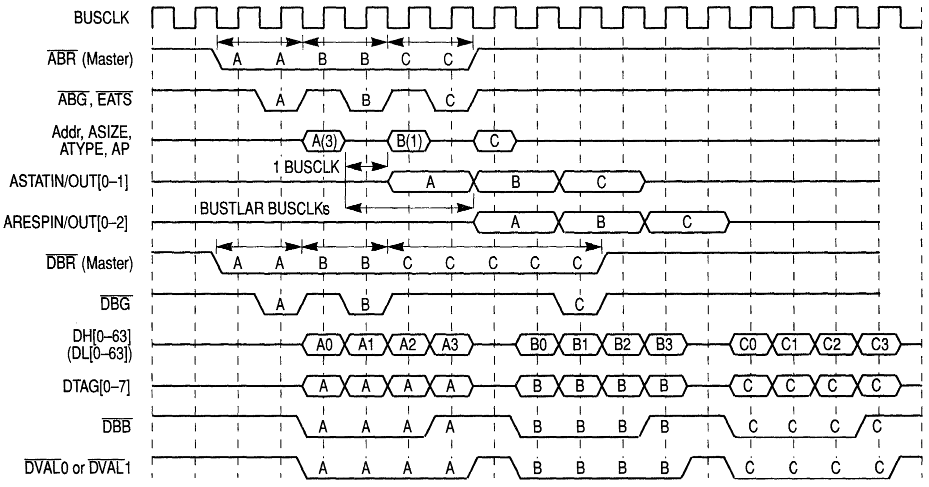


Figure 8-4. Three Burst Write Operations—$\overline{\text{DBG}}$ Given Earliest Cycle Possible (Same Cycle as $\overline{\text{EATS}}$)

Figure 8-5 is an example of a snooper supplying intervention data to a 620 master using the intervention protocol. Intervention is a mode which can be used when a snooper responds modified when a master tries to do a read-with-intent-to-modify (RWITM) or a normal bus read operation. The earliest that data can be sampled by the master is the same as for all reads (same cycle as ARESPIN is sampled). The $\overline{\text{DCACHE}}$ signal tells the master and the memory controller that the data is coming from a snooper.

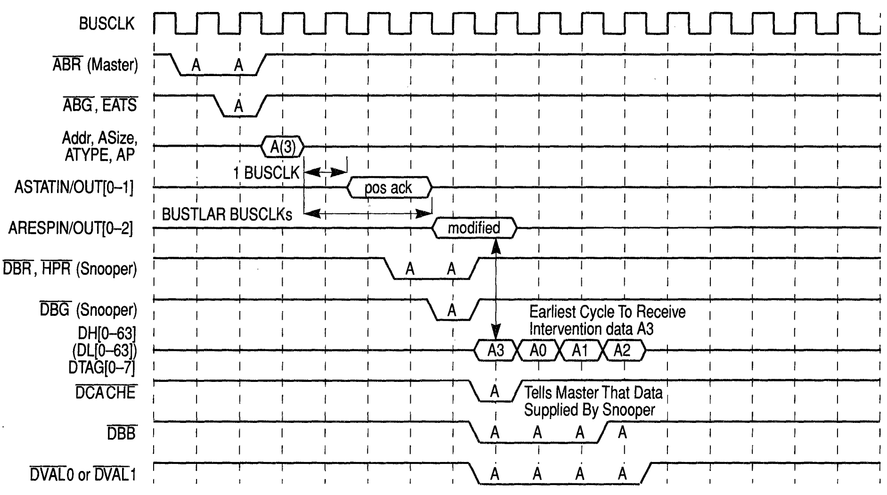


Figure 8-5. Intervention: Memory Read with a Snoop Hit

Figure 8-6 is an example of how a bus adapter could extend the snoop response time beyond the BUSTLAR value. This is done by the bus adapter responding with a snoop response of the Rerun operation. When the bus adapter is ready to provide the snoop response to the master, it issues ReRun with the same tag as was used by the master. This tells the master to try to start this bus operation again. This time, the bus adapter responds with a valid snoop response.

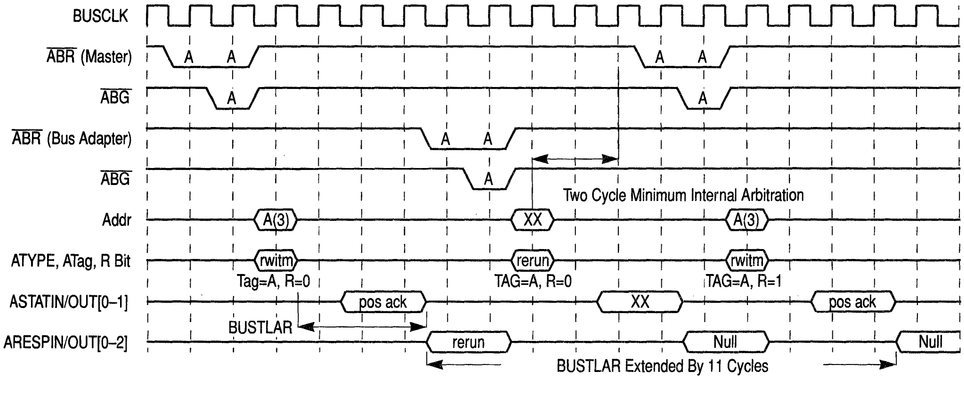


Figure 8-6. Rerun Mechanism (Non-Sync/TLBSync): Extending Snoop Response

Figure 8-7 is a special case of burst reads where data is supplied to several masters from memory using one $\overline{\text{DBG}}$. Memory may bundle data into one data bus tenure so that one dead bus cycle does not have to be inserted between data bus tenures. Masters sampling the data know which data to sample because the data bus tag changes every four beats.

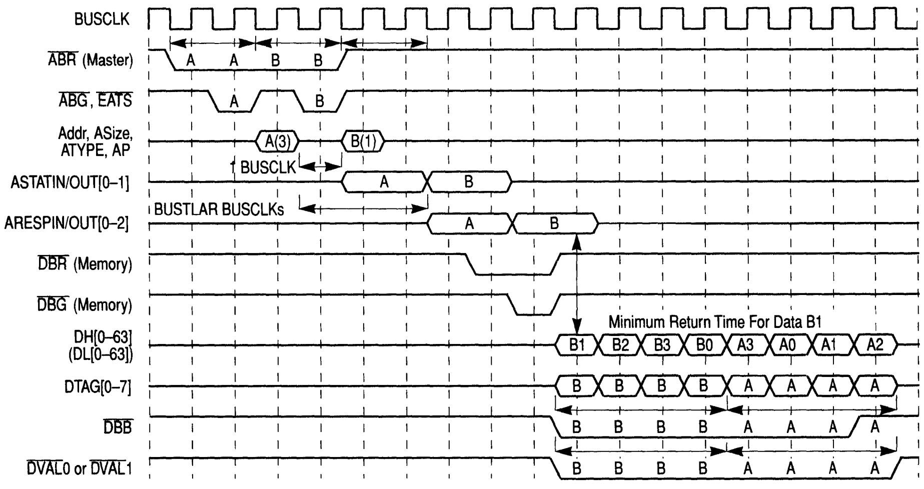


Figure 8-7. Special Case of Burst Read: Memory Supplies Data to Multiple Masters in One Data Bus Tenure

8.2 Processor Interface Termination

The following sections provide information on the 620 processor interface termination.

- Output-only signals—When $\overline{\text{HRESET}}$ is asserted the 620 will three-state all signals, including output-only signals. In order to prevent these signals from floating asserted when $\overline{\text{HRESET}}$ is asserted they must be pulled to the high or deasserted state.
- $\overline{\text{DVAL}}$, $\overline{\text{DBB}}$ and $\overline{\text{DCACHE}}$ signals—The $\overline{\text{DVAL}}$, $\overline{\text{DBB}}$ and $\overline{\text{DCACHE}}$ signals are terminated high. The $\overline{\text{DVAL}}$ and $\overline{\text{DBB}}$ signals are not driven when the bus is idle. The $\overline{\text{DCACHE}}$ signal is not required to be driven by a data provider that will never assert $\overline{\text{DCACHE}}$.

8.3 Arbitration

The 620 bus utilizes a centralized, as opposed to a distributed, arbitration scheme. The address bus and data bus are arbitrated for independently, except for address-data (write) bus operations.

A bus device that wants bus ownership issues a request to the central arbiter, which grants ownership of the address and/or data bus. The arbitration algorithm, such as round-robin or fixed priority, is defined by the central arbiter.

8.3.1 Arbitration Requests

All bus operation types cause one or more of these arbitration request types, as defined in Table 8-16 under the column titled “Address-Data Type.”

- Address-Only—The address-only arbitration type is for an address bus only operation. In the case of bus operations, an address-only bus operation may cause one or more bus devices to respond with a data-only bus operation. The arbiter may arbitrate the address bus independent of the data bus for the address-only request type.
- Data-Only—The data-only arbitration type is always in response to an address-only bus read operation. The arbiter may arbitrate the data bus independent of the address bus for the data-only request type.
- Address-Data—The address-data arbitration type is for data store type operations that pair a store data bus operation with an address bus operation. The arbiter must not allow the data to be sampled prior to the address being sampled. This means that for some system configurations, the arbiter must delay the assertion of $\overline{\text{DBG}}$ relative to the $\overline{\text{ABG}}$ in response to address-data requests. The relationship of $\overline{\text{DBG}}$ to $\overline{\text{ABG}}$ is a function of $\overline{\text{ABG}}$-to-address-sampled latency and $\overline{\text{DBG}}$-to-data-sampled latency. See Section 8.3.9, “Early Address Transfer Start (EATS),” to determine address sample latency and Section 8.5.6, “Data Valid (DVAL[0–1]),” for a discussion of the data sample latency.

  Address-data requests may be used to simultaneously issue address-only and data-only bus operations that do not have the same bus tag. However, the 620 will not utilize this feature. The 620 will only use address-data requests for address-data bus operations. The 620 will arbitrate for address-only and data-only operations as separate requests.

8.3.2 High-Priority Bus Operations

The $\overline{\text{HPR}}$ signal asserts to indicate that the 620 contains one or more high-priority bus operations. See Section 8.3.4, “Internal Request Arbitration,” for the definition of low and high-priority data bus arbitration.

There are two types of high-priority bus operations, as described by the following:

- Pushes and intervention—An arbitration request on behalf of a snoop push or a data-only intervention. See Section 8.12.4, “Write-With-Kill,” Section 8.12.5, “Write-With-Clean,” and Section 8.7, “Intervention and Push Definition.”

  $\overline{\text{HPR}}$ may be left disconnected, which effectively groups high and low priority requests.
- Collisions against the store buffer—An arbitration request on behalf of a copy-back or store that has caused a bus operation to be retried due to collision detection. See Section 8.18.3.4, “Rule 4: Operations that Take CD Priority.”

$\overline{\text{HPR}}$ asserts when either $\overline{\text{ABR}}$ or $\overline{\text{DBR}}$ are asserted and there are one or more high-priority operations that need the bus. $\overline{\text{HPR}}$ asserts for a low-priority request if there are internally-queued, high-priority requests. $\overline{\text{HPR}}$ stays asserted until there are no outstanding high-priority requests. The $\overline{\text{HPR}}$ signal tells the arbiter that all high-priority bus operations and all non-high-priority operations in front of high-priority bus operations should be treated as high-priority bus operations.

For more information on $\overline{\text{HPR}}$, refer to Section 7.2.1.3, "High Priority Request (HPR)—Output."

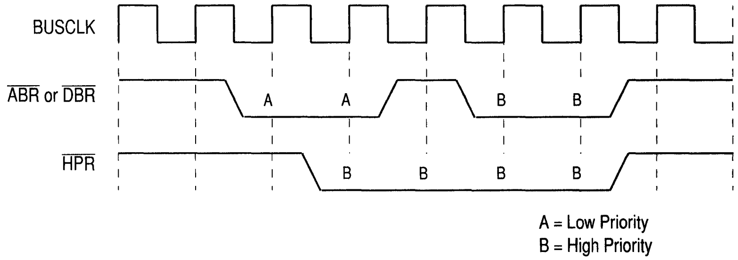


Figure 8-8. Assertion and Deassertion of $\overline{\text{HPR}}$

8.3.3 Withdrawing a Bus Request

The 620 will never withdraw a request before the appropriate grant is received (with the exception of a hard reset).

8.3.4 Internal Request Arbitration

Internal operations seeking access to the bus are grouped into four groups. Arbitration between groups is fixed priority, with the highest priority given to group 1. Arbitration within each group is round-robin.

1. High-priority store buffer operations

   There are three sources for high-priority store buffer operations.

   a) Bus snoop pushes. (Cast-outs caused by bus snoops)

   b) Bus snoop data-only intervention. See Section 8.7, "Intervention and Push Definition."

   c) Any low priority store buffer operation that causes a CD collision with either a core or bus operation will be turned from a low to a high priority store. See group 4.

2. Instruction load operations—Instruction fetches.
3. Data load and address-only operations—Data loads, tablewalk loads (I and D) and address-only operations.

4. Low-priority store buffer operations

   There are two sources for low-priority store buffer operations.

   a) Cast-outs caused by cache management instructions and data or instruction loads.

   b) Write-through or cache-inhibited stores.

8.3.5 External Request Arbitration

The external arbiter, that arbitrates between multiple 620s and other bus devices, implements an algorithm which may be the same or different from the 620 internal request arbitration.

While a fair arbitration algorithm, such as round-robin or snap-shot, applied to all bus requests will functionally work, performance will be less than optimal. The 620 supports an arbitration model that divides requests into three groups or tiers that have a strict arbitration priority with respect to each other and a fair arbitration priority within a group or tier. The three groups or tiers are listed as follows:

1. High priority address-only, address-data, or data-only
2. Low priority address-only
3. Low priority address-data or data-only.

8.3.6 Arbitration Grants

There are two grant signals, address bus grant ($\overline{\text{ABG}}$) and data bus grant ($\overline{\text{DBG}}$). $\overline{\text{ABG}}$ and $\overline{\text{DBG}}$ are point-to-point active-low signals from the arbiter to the bus device. The grants are not combined like the requests. $\overline{\text{ABG}}$ and $\overline{\text{DBG}}$ are treated by the 620 as independent grants for each bus.

$\overline{\text{ABG}}$ and $\overline{\text{DBG}}$ are pulsed grants, asserted for one BUSCLK cycle, indicating which buses are being granted. Except for the sustained address bus parking case, as described by Section 8.3.10.2, "Sustained Address Bus Parking," $\overline{\text{ABG}}$ cannot be asserted more frequently than once every other cycle. $\overline{\text{DBG}}$ can never be asserted more frequently than once every other cycle. Asserting $\overline{\text{ABG}}$ and $\overline{\text{DBG}}$ in violation with the rules stated in the following sections may result in undefined behavior. $\overline{\text{ABG}}$, when asserted, indicates that the address bus master can start driving the address bus on the same edge that $\overline{\text{ABG}}$ is sampled asserted.

A data bus master may start driving the data bus when both of the following two conditions are satisfied:

- $\overline{\text{DBG}}$ is asserted.
- $\overline{\text{DBB}}$ is deasserted for two or more BUSCLK cycles.

If $\overline{\text{DBB}}$ was deasserted for two or more BUSCLK cycles when $\overline{\text{DBG}}$ is asserted, then the master will start driving the bus on the same edge that $\overline{\text{DBG}}$ is sampled asserted. The data bus grant is pending from the assertion of the data bus grant to the beginning of the data transfer. See Section 8.3.8, "Pending Data Bus Grant Arbitration."

8.3.7 Address-Data Arbitration

Address-data arbitration is used for write or store operations where the data bus operation must occur with or after the address bus operation. This means that the arbiter must control the assertion of DBG with respect to ABG and EATS for address-data bus operations. See Section 8.3.9, "Early Address Transfer Start (EATS)."

If an address-data request is pending and only ABG is asserted, then the 620 deasserts ABR, keeping DBR, and possibly HPR, asserted. The 620 will not assert ABR for a new request until DBG is received for the DBR belonging to the address-data request. If the address bus operation for an address-data bus transaction is cancelled or retried and the DBR is still pending, then the 620 will wait for the pending data request to be granted before continuing; however, the 620 will not take data bus ownership (DBB will not be asserted). No DVALs will be asserted for the DBG in order to satisfy the rules for data bus tag deallocation. See Section 8.10, "Bus Tags." The 620 will not use the grants for an address-data request for address-only or data-only bus operations.

8.3.8 Pending Data Bus Grant Arbitration

The 620 supports zero or one pending DBGs. A description of each follows:

- Zero pending DBG—The arbiter does not issue DBG when DBB is asserted. This means that DBB for BUSCLK cycle 0 needs to be included into the equations that will produce DBG in BUSCLK cycle 1.
- One pending DBG—DBB is registered by the arbiter, the arbiter uses only the registered version of DBB and the arbiter asserts another DBG before knowing whether the previous DBG will take the bus for more than one BUSCLK. If the transfer is greater than one BUSCLK then the arbiter will hold off another DBG until the beginning of the data tenure for the pending DBG.

Pending DBG arbitration is described in Figure 8-9.

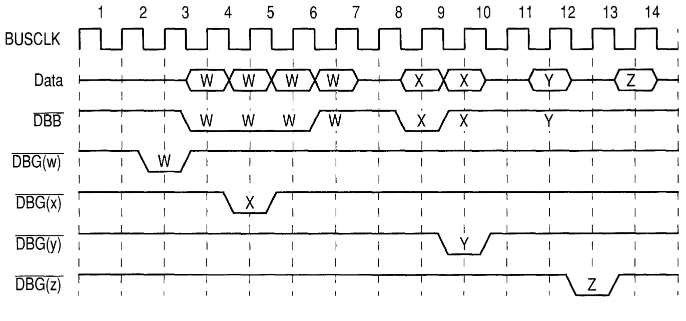


Figure 8-9. Pending DBG Arbitration

DBG(w) is the data bus grant for data bus operation W and so forth to Z. In cycle 2 DBB has been deasserted for two BUSCLKs, so W takes the bus in cycle 3. Although DBG for X is asserted in cycle 4 DBB is not deasserted for two BUSCLKs until cycle 7, and X starts in cycle 8. DBG for Y asserts in cycle 9 and DBB has not deasserted for two BUSCLKs until cycle 10, and Y starts in cycle 11. Z is like X in that DBB has been deasserted for two BUSCLKs in cycle 12, and Z begins in cycle 13.

Note that DBG cannot be asserted speculatively (no DBR) since the 620 does not support data bus parking.

8.3.9 Early Address Transfer Start (EATS)

The arbiter asserts the early address transfer start (EATS) signal to all bus participants for one BUSCLK cycle. EATS may be a unidirectional bussed signal or it can be driven point-to-point with the same value.

EATS indicates to the master driving the address that it will disable its address drivers one BUSCLK after sampling the assertion of EATS. For all other devices monitoring the address bus, EATS indicates that the address transfer should be sampled one BUSCLK after from sampling the assertion of EATS.

8.3.9.1 EATS Assertion Relative to ABG

EATS can be asserted in the same cycle as ABG or on the cycle following ABG. It is not permitted for EATS to be delayed more than one BUSCLK from ABG. If EATS is asserted in the same cycle as ABG, then the address tenure is one BUSCLK cycle and a new address transfer can begin every two BUSCLK cycles. If EATS is asserted in the cycle following the assertion of ABG, then the address tenure is two BUSCLK cycles.

8.3.9.2 EATS to DBG Minimum Latency for Address-Data Arbitration

The system arbiter controls the minimum latency from EATS to DBG which is useful for systems where the address must be seen a minimum number of BUSCLK cycles before the data. The arbiter may want a non-zero minimum latency from EATS to DBG for crossbar implementations. For example, if the arbiter guaranteed that the minimum latency from EATS to DBG was two BUSCLK cycles, then the address bus tag could be passed through the address controller to each data slice that needed to snoop the data bus tag. A system that requires a non-zero latency from EATS to DBG implies the need for a count function for each address-data operation that can have asserted an EATS and not yet asserted a DBG. The absolute minimum latency the 620 supports from EATS to DBG is zero BUSCLK cycles, which means they are both asserted simultaneously.

Figure 8-10 refers to a one BUSCLK address transfer time. The maximum bandwidth is one address every two BUSCLKs.

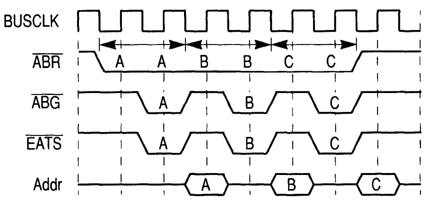


Figure 8-10. $\overline{\text{EATS}}$ Asserted Same Cycle as $\overline{\text{ABG}}$

Figure 8-11 refers to a two BUSCLK address transfer time. The maximum bandwidth is one address every three BUSCLKs. Note that operation D can be granted one BUSCLK sooner if C and D are the same device.

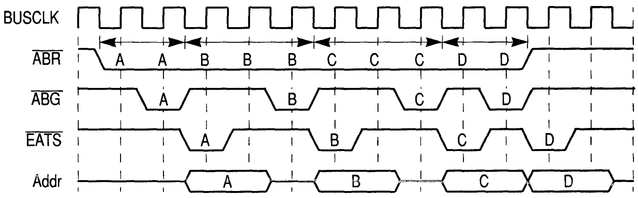


Figure 8-11. $\overline{\text{EATS}}$ Asserted One Cycle after $\overline{\text{ABG}}$

8.3.10 Bus Parking

Bus parking is a mechanism that can minimize the latency from internally requesting the bus to obtaining the bus.

The 620 bus protocol allows for bus parking for the address bus only. Bus parking is implemented by the arbiter, which pulses the appropriate grant to the parked master. The arbiter specifies the algorithm for bus parking.

If a master has a transaction to run that matches the parked grant, then the master can take the bus without asserting the external request, thus eliminating arbitration latency. Otherwise, the bus device will assert the request.

A parked $\overline{\text{ABG}}$ without a $\overline{\text{DBG}}$ may be used for address-only and address-data operations. If an $\overline{\text{ABG}}$ is speculatively asserted to the master and the master has an address-data transaction to run, the master may use the parked $\overline{\text{ABG}}$ for the address-data transaction and assert $\overline{\text{DBR}}$. The master will hold $\overline{\text{DBR}}$ asserted until $\overline{\text{DBG}}$ is asserted.

The 620 does not support data bus parking.

Unless otherwise noted, all following discussion and figures assume that $\overline{\text{EATS}}$ is asserted coincident with $\overline{\text{ABG}}$, producing a one cycle address tenure with two cycles from address to address.

8.3.10.1 Address Bus Parking

In reference to Figure 8-12, parked grant P_A occurs during the same cycle as internal request A (cycle 2), which enables the address for bus operation A to start on the next cycle (cycle 3). If a parked grant is not present at the same cycle as an internal request (cycle 4), then an external request is made (cycle 5). The grant may be a parked grant P_B and will enable the address for bus operation B to start on the next cycle (cycle 6). If the grant was a parked grant, then the arbiter may issue a grant that will effectively go unused (cycles 7 and 8).

If a bus device is issued a parked address grant and it does not need to use the address bus, then it must run a null address bus operation.

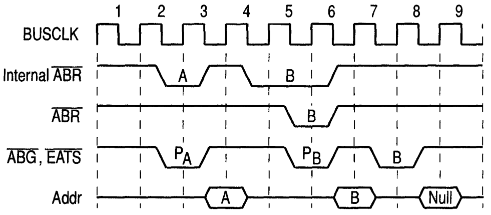


Figure 8-12. Address Bus Parking

8.3.10.2 Sustained Address Bus Parking

While the reduced latency associated with the statistical parking capability is adequate for most systems, there are some systems (such as uniprocessor systems) that could easily park the bus device in a sustained, non-statistical fashion if that capability existed. For these systems, memory latency could be further reduced if the arbiter could assert the address bus grant to a bus device for more than one bus clock cycle, such that if the bus device does have an operation to run, it is guaranteed to win the address bus immediately. With certain constraints, described below, it is permissible for the arbiter to park the 620 on the address bus in the "sustained" fashion.

Note: This description may contradict statements made elsewhere in this document. This is a special mode of operation; the rest of this document assumes the normal arbitration mode. If and only if the system designer meets all of the constraints outlined in this section, then the sustained address bus parking capability can be utilized.

System constraints:

- $\overline{\text{EATS}}$ and $\overline{\text{ABG}}$ are asserted together, and $\overline{\text{DBG}}$ must not be asserted. Only address tenures of one BUSCLK are supported for sustained address bus parking.

- The bus device will drive the address bus with a valid transaction if it has a pending internal request; otherwise, the bus device must drive the Null address bus operation in response to the parked address bus grant.
- All bus devices must be capable of decoding the Null type field and avoid filling their snoop buffers and other system resources with these Null operations.

If the preceding constraints are met, then the arbiter may assert both the address bus grant ($\overline{\text{ABG}}$) and early address transfer start ($\overline{\text{EATS}}$) to the bus device for multiple, consecutive bus clock cycles. In response to the parked address bus condition, the bus device will drive the address bus and associated signals starting with the bus clock following the assertion of $\overline{\text{ABG}}$ and $\overline{\text{EATS}}$. If the device has no pending internal request, the bus device will drive the Null code on the address type signals. The Null code will be driven on every bus clock cycle in which the bus device is parked and it has no pending internal request. This is illustrated in Figure 8-13.

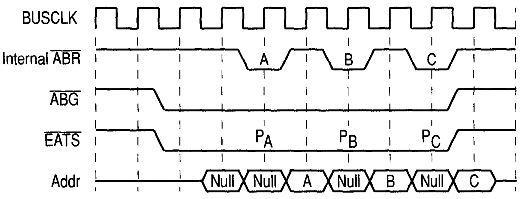


Figure 8-13. Sustained Address Bus Parking

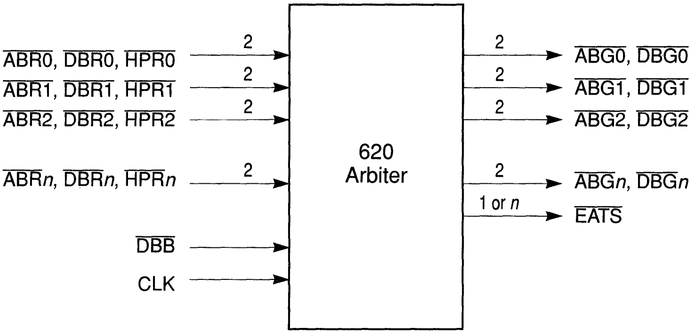
If the bus device has a pending internal request, it will present that transaction on the following bus clock cycle, without asserting the address bus request. If the pending internal request is for a store operation, and the data bus grant is not asserted, then the data bus request will be asserted with the address transaction on the following bus clock cycle. The transaction will be presented for a single bus clock and then removed. If the bus device is still parked, it will begin driving the Null code on the following bus cycle(s). If it is no longer parked, the bus device drivers will go back into a high-impedance state.

Special considerations:

- The data bus cannot be parked in a fashion similar to the sustained address bus.
- The arbiter knows when it can deassert the bus grant in the same manner as the normal mode; the 620 may have elected to run an address operation in the last cycle that address bus grant was sampled asserted. In this case, it will run a single clock cycle address operation in the next bus clock cycle (the first cycle with $\overline{\text{ABG}}$ deasserted).
- The 620 cannot run valid bus operations every BUSCLK in sustained address parking mode. The maximum address throughput is a new address every other BUSCLK cycle. Like statistical parking, this mode does not improve throughput; however, it can be used to lower memory latency.

8.3.11 Arbiter Block Diagram

Figure 8-14 is intended to illustrate the input and output signals of the 620 arbiter.



$\overline{ABR0}$, $\overline{DBR0}$, $\overline{HPR0}$
$\overline{ABR1}$, $\overline{DBR1}$, $\overline{HPR1}$
$\overline{ABR2}$, $\overline{DBR2}$, $\overline{HPR2}$
$\overline{ABRn}$, $\overline{DBRn}$, $\overline{HPRn}$
$\overline{DBB}$
CLK
2
2
2
2
620
Arbiter
2
2
2
2
1 or n
$\overline{ABG0}$, $\overline{DBG0}$
$\overline{ABG1}$, $\overline{DBG1}$
$\overline{ABG2}$, $\overline{DBG2}$
$\overline{ABGn}$, $\overline{DBGn}$
$\overline{EATS}$

Figure 8-14. Block Diagram of the Arbiter

8.4 Address Bus Transfer Protocol

The address bus transfer protocol specifies how address and address related status is passed between bus devices. Following are useful definitions:

- Address command—The address command transfers master information to all memories and snoopers. The information contained in the address command will vary from operation to operation but generally has an address, a tag and address attributes.
- Address status—The address status is a fixed, low-latency composite response from all bus devices for the purpose of providing positive acknowledge, flow control retry and address parity information.
- Address response—The address response is a “boot-time determined” latency composite response from all bus devices for the purpose of providing coherency information.

8.4.1 Address Transfer Example

Figure 8-15 is an example of three address transfers, labeled A, B and C.

- BUSTLAR is three BUSCLK cycles, ARESPOUT is driven three BUSCLK cycles from the address being sampled.
- BUSRESPTEN is two BUSCLK cycles, ASTATIN and ARESPIN are sampled two BUSCLK cycles from being driven.

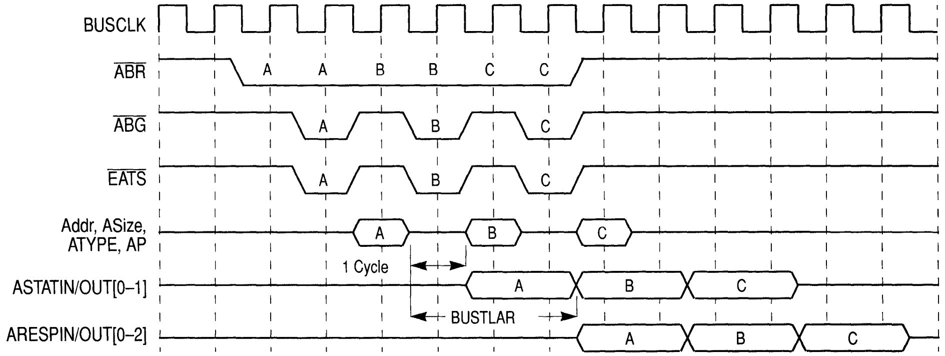


Figure 8-15. Address Transfer Protocol

8.4.2 Address Command

The address command consists of the following:

- Address, Address Bus Tag—Refer to Section 8.9, "Address Bus"
- ATYPE—Refer to Section 8.6, "Address Commands Definition"
- Size—Refer to Section 8.8, "ASIZEDATA[0–3] and ASIZEBURST Definition"
- Address Parity—Refer to Section 8.11, "Parity Protection"

8.4.3 Address Status and Address Response Signals

This section describes how the address status and address response signals interact.

- Address status—The address status (AStat) consists of the unidirectional signals ASTATOUT[0–1] and ASTATIN[0–1]. AStat provides positive acknowledge (PosAck), retry flow control (Retry) and address parity (AParErr) functionality for the address bus. Bus operations that do not need the functionality of the address response (AResp) will complete based on AStat.
- Address response—The address response (AResp) consists of the unidirectional signals ARESPOUT[0–2] and ARESPIN[0–2]. AResp provides coherency information (Null, Shared, Modified) and control information (Retry and ReRun).

8.4.3.1 Address Status and Address Response Communication

All ASTATOUT and ARESPOUT output signals from all bus participants are driven point-to-point to logic that then drives ASTATIN and ARESPIN inputs, either point-to-point or bused. The system must handle the priority encoding of all ASTATOUT and ARESPOUT outputs and generate the appropriate ASTATIN and ARESPIN inputs. The priority definition is defined in Table 8-9 and Table 8-11. With respect to Figure 8-16, a 620 is a bus device.

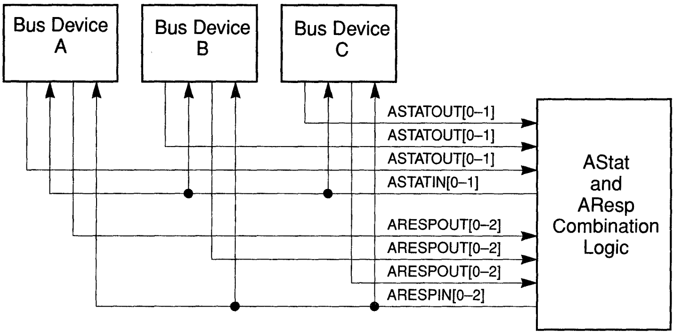


Figure 8-16. AStat and AResp Interconnection Diagram

8.4.3.2 Address Status and Address Response Validation

Table 8-1 defines when AStat and AResp are valid for each bus operation type. AStat-not-valid means that ASTATIN will be ignored. AResp-not-valid means that ARESPIN will be ignored. AResp-enabled operations that are AStat-aborted (AStat Retry, AParErr, or no PosAck and PosAck enabled) are AResp-not-valid. The I and M bits are the PTE IM bits.

Table 8-1. AStat and AResp Valid States

Bus Operation		AStat Valid Status	AResp Valid Status	Comment
Null		No	No	
PIO Reply, ReRun		Yes		AStat support is limited; see Table 8-10.
PIO Load + Store (Immed + Last), External Control In/Out, TLBIE, EIEIO, IKill				Only flow control is provided for these bus operations.
Read, Write with Flush-$\overline{A}$,	I=1 and M=0			AResp is ignored in order to provide high performance operation.
	I=0 or M=1		Yes	Coherency response is needed for these bus operations.
RWITM, Clean, Flush, DKill, DClaim, Write with Kill, Write with Clean, Write with Flush-A, LARX-Reserve, SYNC, TLBSYNC				

8.4.4 Address to ASTATOUT Latency (One BUSCLK)

The latency from the address sample point to the ASTATOUT drive point is always one BUSCLK. All bus devices must be able to compute ASTATOUT in one BUSCLK.

8.4.5 Address to ARESPOUT Latency (BUSTLAR[0–2])

The address to response latency (BUSTLAR) is defined to be from the address sample point to the ARESPOUT drive point. Table 8-2 shows the encoding definition for BUSTLAR.

BUSCSR[BUSTLAR] is configured by software at power up, before or at the same time BUSCSR[BUSSNPEN] is asserted and is not changed during system operation.

Table 8-2. BUSTLAR[0–2] Encoding Definition

BUSTLAR[0–2]	Latency from Address Sample to Response Drive
010–111, 000	2–7, 8 BUSCLK cycles
001	Reserved

The minimum BUSTLAR supported by the 620 is configuration-dependent and is defined in Table 8-3. The table values represent the cycles required by a 620 in the specified configuration to provide the response for a snooped bus operation. The PCLK:L2 ratio referenced in Table 8-3 refers to system variable L2RATIOSR which is described in Section 9.3.1.4, "L2RATIOSR Bit." The "X" entry in Table 8-3 corresponds to invalid combinations of PCLK:L2 and PCLK:BUSCLK ratios—the 620 does not support ratio combinations where the L2 is slower than the bus. Table 8-3 assumes that L2SINGSYNC and Coprocessor mode are both disabled. The maximum BUSTLAR supported by the 620 is eight BUSCLK cycles.

Table 8-3. BUSTLAR[0–2] Minimum Definition

PCLK:L2 Ratio	PCLK:BUSCLK Ratio 2:1	PCLK:BUSCLK Ratio 3:1	PCLK:BUSCLK Ratio 4:1	Unit
No L2	3	2	2	BUSCLK cycles
1:1	3	2	2	BUSCLK cycles
2:1	5	4	3	BUSCLK cycles
3:1	X	5	4	BUSCLK cycles

8.4.5.1 BUSTLAR Initialization Rules

If BUSTLAR is initialized to a value less than the specified minimum value, the 620 may generate an ARESPOUT Retry for all snooped operations. The 620 will ARESPOUT Retry if the cache snoops are not complete when ARESPOUT is driven.

8.4.5.2 Derivation of BUSTLAR Values

Table 8-4 describes the parameters used to generate BUSTLAR values. The L2RATIOSR parameter is described in Section 9.3.1.4, “L2RATIOSR Bit.” The L2SINGSYNC mode bit is described in Section 9.3.1.10, “L2SINGSYNC Bit.”

Table 8-4. BUSTLAR Parameters

Parameter	Definition	Latency (in PCLKs)
BUS_TO_L1	Number of cycles between a BUSCLK and the PCLK used to clock the L1 arbiter request.	3
BUS_TO_L2	Number of cycles between a BUSCLK and the subsequent PCLK edge used to clock the 620-internal address register for the L2 cache.	Refer to Table 8-5
L2_SR	SRAM array latency	Defined by L2RATIOSR
L2_REGS	Number of L2 registers	Defined by L2SINGSYNC
L1_DAT_VAL	Number of cycles to validate L1 Snoop Data	2.5
L2_DAT_VAL	Number of cycles to validate L2 Snoop Data	1
ARESP_GEN	Number of cycles to generate ARESPOUT	0.5

Table 8-5 lists the values of the BUS_TO_L2 parameter for the various configurations.

Table 8-5. BUS_TO_L2 Values

PCLK:L2 Ratio	PCLK:BUSCLK Ratio 2:1	PCLK:BUSCLK Ratio 3:1	PCLK:BUSCLK Ratio 4:1	Unit
1:1	1	1	1	PCLK
2:1	1	2	1	PCLK
3:1	X	3	3	PCLK

Table 8-6 lists the L2_REGS value associated with all combinations of L2SINGSYNC.

Table 8-6. L2_REGS Values

L2SINGSYNC	L2_REGS
0	2
1	1

8.4.5.3 L2 Cache and Coprocessor Mode Disabled

For a system with no L2 cache and Coprocessor mode disabled, BUSTLAR is generated by rounding up the result of the following equation:

$$\text{BUSTLAR} = (\text{BUS\_TO\_L1} + \text{L1\_DAT\_VAL} + \text{ARESP\_GEN}) / \text{BUSRATIO}$$

8.4.5.4 L2 Cache or Coprocessor Mode Enabled

For a system with either L2 cache or Coprocessor mode enabled, BUSTLAR is generated by rounding up the result of the following equation:

> BUSTLAR = (BUS_TO_L2 + L2_SR*(1+L2REGS) + L2_DAT_VAL + ARESP_GEN) / BUSRATIO

In addition, if the L2 is configured in Coprocessor mode, additional latency may be introduced due to high-impedance cycle injection. Table 8-7 illustrates the latency impact as a function of three consecutive accesses where the number of high-impedance cycles is shown for the third access. The 0 represents access to one read device and the 1 represents access to a second read device. See Section 9.2.3, "L2 Direct Connectivity to SRAMs."

Table 8-7. L2 Latency Additions Due to High-Impedance Cycle Injection

Access Pattern	Number of L2RATIOSR Multiples Added to Third Access
000	0
001	1
010	2
011	1
100	1
101	2
110	1
111	0

Note that the values in Table 8-3 for configurations with a L2 cache are somewhat conservative. To generate those values, the following assumptions were made:

- Double-synchronous SRAMS so L2_REGS=2.
- No high-impedance cycles.

Using these assumptions, BUSTLAR is generated by rounding up the result of the following equation:

BUSTLAR = (BUS_TO_L2 + L2_SR*3 + L2_DAT_VAL + ARESP_GEN) / BUSRATIO

8.4.6 Address Status and Address Response Tenure (BUSRESPTEN[0–1])

BUSRESPTEN[0–1] defines the latency from ASTATOUT and ARESPOUT being driven until ASTATIN and ARESPIN can be sampled. BUSRESPTEN is assigned by the hardware configuration mechanism. BUSRESPTEN is software readable by the BUSCSR SPR.

Table 8-8. BUSRESPTEN[0–1] Code Definition

BUSRESPTEN[0–1]	Definition
01	Reserved
10–11	2–3 BUSCLKs
00	Reserved

ASTATOUT and ARESPOUT are always driven valid for two BUSCLK cycles, independent of the configuration of BUSRESPTEN[0–1]. Figure 8-17 illustrates the relationship between ASTATOUT and ARESPOUT and ASTATIN and ARESPIN for each response tenure.

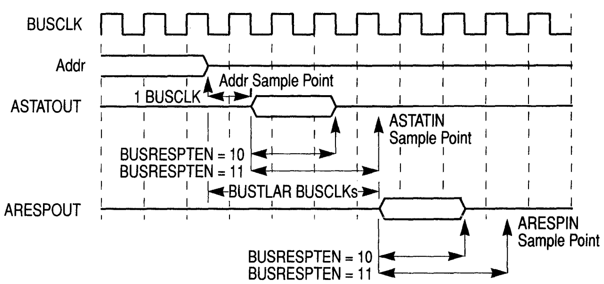


Figure 8-17. BUSRESPTEN[0–1] Timing Definition

When BUSRESPTEN = 3 a register stage is required in the AStat and AResp collection logic in order to pipeline the AStat/AResp collection.

The BUSRESPTEN configuration used is a function of the speed of the AStat/AResp collection logic. BUSRESPTEN[0–1] is readable from the BUSCSR SPR.

8.4.7 Snoop Pipeline Depth and Snoop Operation Processing

The 620 bus protocol specifies pipelined address responses. A bus device which supports pipelined address responses has an implementation-dependent snoop pipeline depth which corresponds to the number of addresses that a bus device is capable of storing while the address responses remain outstanding.

The optimal snoop pipeline depth is determined by the number of addresses that can be transmitted over the address bus for the period of time that it takes one address to be issued by the master, snooped by the snoopers and resolved by all bus devices.

If a bus device implements snoop logic that is less than the optimal depth, that bus device will cause retries due to an insufficient number of snoop buffers being available during saturated address bus conditions.

The optimal snoop pipeline depth is determined using the following assumptions:

1. The minimum tenure for an address is one BUSCLK.
2. The address bus is saturated so an address is snooped every other BUSCLK.
3. The response (snoop) latency is the maximum of eight BUSCLKs (BUSTLAR=8).
4. The response tenure is the maximum of three BUSCLKs (BUSRESPTEN=3).
5. The address response can be resolved by the snooper in 1 PCLK.

Figure 8-18 illustrates the snoop buffer utilization given the stated assumptions (BusRatio is 2:1). Under these conditions, once an address (referred to as Address 1) is snooped off the bus, it remains in a queue location for 11.5 BUSCLKs. In that duration another 5 addresses (Address 2–6) are sampled off the bus. Snoop Buffer 0 is emptied just prior to the Address 7 sample. These assumptions yield an optimal snoop pipeline depth of 6 locations.

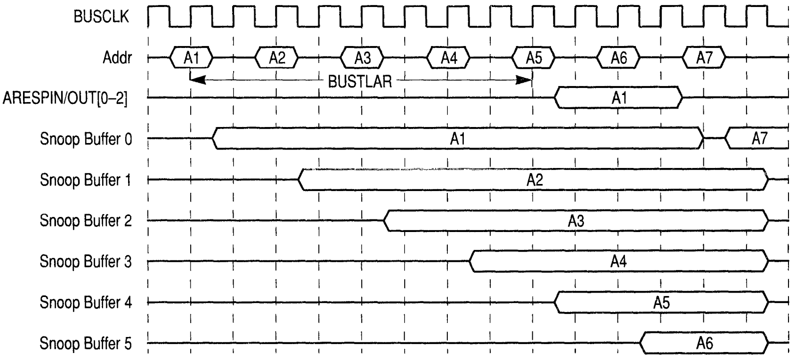


Figure 8-18. Optimal Snoop Pipeline Depth

The 620 snoop pipeline depth is determined using the following assumptions:

1. The minimum tenure for an address is one BUSCLK cycle.
2. The address bus is saturated so an address is driven onto the bus every other BUSCLK. However, it is assumed that only a subset of the bus addresses are actually snooped. This assumption is based on the fact that operations such as copybacks, pushes, SYNCs, TLBSYNCs, TLBIEs and IKills are sourced to the bus with the M-bit equal to 0. For this analysis, an average of 1 out of every 5 bus operations is assumed to be non-memory-coherent.
3. Given the minimum response latencies specified in Table 8-3, a reasonable response (snoop) latency is 6 BUSCLKs (BUSTLAR=6).
4. The maximum response tenure is three BUSCLKs (BUSRESPTEN=3).
5. An address response can be resolved by all bus devices in 1 PCLK.

Under these conditions, once an address (referred to as Address1) is snooped off the bus, it remains in a queue location for 9.5 BUSCLKs. During those 9.5 BUSCLKs, another four addresses (Address2–5) are sourced onto the Bus and 3 of these 4 addresses must be snooped. Thus, Address1–5 fill four snoop buffers and the Address1 snoop location empties prior to Address 6 being sampled. These assumptions yield a 620 snoop pipeline depth of four locations.

For both analyses, assumption 5 indicates that 1 PCLK is sufficient to resolve all bus operations. In the 620, this is true for all bus operations that miss the cache. For those operations that need additional processing (such as a push or intervention), additional logic is needed to complete the operation. The 620 incorporates two independent state machines to concurrently process these bus operation types.

Each of the state machines is capable of handling either a block move operation (push or intervention), a tag write, an IKill, a SYNC, a TLBIE or a TLBSYNC.

The 620 implements a 4-deep snoop pipeline and two independent state machines for additional processing.

8.4.8 Address Status In/Out

Table 8-9 defines the encoding and priority of address status out (ASTATOUT) and address status in (ASTATIN). Priority encoding 1 is highest and 4 is lowest.

Table 8-9. AStat: Code and Priority Definition

ASTATOUT[0–1] ASTATIN[0–1]	Priority	Definition	Reference Page
00	1	AParErr (Address Parity Error)	Refer to Section 8.4.11, "Address Status Address Parity Error (AStat AParErr)"
01	2	Retry	Refer to Section 8.4.17, "Address Status Out and Address Response Out Retry," and Section 8.4.18, "ASTATIN and ARESPIN Retry"
10	3	PosAck (Positive Acknowledge)	Refer to Section 8.4.10, "Address Status Acknowledge"
11	4	NoAck (No Acknowledge)	

Table 8-10 defines which AStat codes are defined for each bus operation type. Shaded is defined for ASTATOUT and ASTATIN.

Table 8-10. AStat Codes Enabled as a Function of Bus Operation Type

Bus Operation Type (AStat Enabled Only)	Status Type			
	NoAck	PosAck	AParErr	Retry
PIO Reply	▨		#1	#2
ReRun	▨		#1	
TLBSYNC, SYNC, TLBIE, EIEIO	▨		▨	▨
Flush, Clean, Write with Flush, Write with Kill, Write with Clean, Read, DKill, IKill, RWITM, DClaim, LARX-Reserve, PIO Load and Store operations, Ext Control Out and In	▨	▨	▨	▨

Notes:

1. The 620 will generate ASTATOUT AParErr if bad parity is detected, but will (if successfully decoded) execute the bus operation anyway. The master for these operations should not run the operation again.
2. The system bus specification allows a PIO master to flow control the PIO Reply from the PIO slave. However, the 620 will not flow control PIO Reply and will not drive ASTATOUT Retry. The 620 will ignore ASTATIN Retry for PIO Reply.

8.4.9 Address Response In/Out

Table 8-11 defines the encoding and priority of address response in (ARESPIN) and address response out (ARESPOUT). The encoding is defined such that if the ReRun responses are not used by a system, then ARESPIN[0] can be pulled high. Priority encoding 1 is highest and 5 is lowest.

The function of ARESPOUT and ARESPIN is defined by Section 8.4.19, "Function of ASTATOUT/ASTATIN and ARESPOUT/ARESPIN," Section 8.17.17, "Master Cache State Transitions Due to Instructions," and Section 8.17.18, "Snooper Cache State Transitions Due to Bus Operations."

Table 8-11. AResp Code and Priority Definition

ARESPOUT[0–2] ARESPIN[0–2]	Priority	Definition	Reference Page
000	—	Reserved[1]	Refer to Section 8.4.15, "Address Response Reserved"
001	—	Reserved[1]	
010	—	Reserved[1]	
011	3	ReRun	Refer to Section 8.4.14, "Address Response ReRun," and Section 8.13, "The ReRun Mechanism"
100	1	Retry	Refer to Section 8.4.17, "Address Status Out and Address Response Out Retry," and Section 8.4.18, "ASTATIN and ARESPIN Retry"
101	2	Modified	Refer to Section 8.4.16, "Address Response Modified"
110	4	Shared	Refer to Section 8.4.13, "Address Response Shared"
111	5	Null (Not Modified or Shared)	Refer to Section 8.4.12, "Address Response Null"

Note: Refer to Section 2.1.2.4, "Bus Status and Control Register (BUSCSR)," for information on the bus response error bit.

Table 8-12 defines which AResp codes are defined for each bus operation type. Shaded is defined for ARESPOUT and ARESPIN.

Table 8-12. AResp Codes Enabled as a Function of Bus Operation Type

Bus Operation Type (AResp Enabled Only)	Response Type				
	Null	Shared	Modified	Retry	ReRun
Read (I=0 or M=1), Clean	▨	▨	▨	▨	▨
RWITM, Flush	▨		▨	▨	▨
Write with Flush (I=0 or M=1), DKill, DClaim, Write with Kill, Write with Clean, TLBSYNC, SYNC, LARX-Reserve	▨			▨	▨

8.4.10 Address Status Acknowledge

The AStat positive acknowledge (PosAck) code indicates that the addressed slave accepted this address bus operation and that a higher priority AStat code, such as AParErr or Retry, did not occur.

- If positive acknowledge is enabled and the no acknowledge (NoAck) code is received from the AStat collection logic by the master, then the BUSPOSACKERR SPR bit is set. If BUSPOSACKEN is asserted then the 620 will take a machine-check exception. The master will abort the address and data portions of the operation. See Section 2.1.2.4, "Bus Status and Control Register (BUSCSR)," BUSPOSACKERR bit. The non-master bus devices will treat this condition as a retry but will not set BUSPOSACKERR or take a machine-check exception.

- If positive acknowledge is enabled and the PosAck code is received, then the operation has been accepted.
- If positive acknowledge is not enabled and the NoAck or PosAck codes are received, then the operation has been accepted.

```
Ack_Is_OK:=
((posack_disabled_op & (PosAck | NoAck))
| (posack_enabled_op & ((BUSPOSACKEN & PosAck)
| (^BUSPOSACKEN & (PosAck | NoAck)));
```

Table 8-13 defines for each 620 bus operation whether positive acknowledge is enabled and what slave bus device is responsible to give a positive acknowledge if positive acknowledge is enabled.

Table 8-13. PosAck Enabled as a Function of Bus Operation Type

620 Bus Operation	Enabled	Definition
Flush, Clean, Write with Flush, Write with Kill, Write with Clean, Read, DKill, IKill, RWITM, DClaim, LARX-Reserve	Yes	Positive acknowledge is given by the memory slave that decodes the real (T=0) memory address. Snoopers do not give positive acknowledge for these bus operations.
PIO Load Immediate + Last PIO Store Immediate + Last		Positive acknowledgment is given by the addressed BUID (T=1).
External Control Out, External Control In		Positive acknowledge is given by the device addressed by the RID.
TLBIE	No	Positive acknowledge is not supported for virtual address bus operations.
SYNC, EIEIO, TLBSYNC, Null, PIO Reply, ReRun		Positive acknowledge is not supported for addressless bus operations.
Reserved		By definition, because the 620 does not issue reserved bus operations, positive acknowledge is not supported by the 620 for reserved bus operations

8.4.11 Address Status Address Parity Error (AStat AParErr)

AStat AParErr indicates to the 620 that at least one bus device detected an address parity error. AStat AParErr is the highest priority AStat code. Section 8.11, "Parity Protection."

8.4.11.1 Bus Operation Abort

All devices will unconditionally abort the bus operation that receives ASTATIN AParErr, independent of whether parity checking is enabled for that device that receives ASTATIN AParErr, except for PIO Reply and ReRun. The master may, but does not have to, reissue the bus operation again.

8.4.11.2 PIO Reply and ReRun

All bus devices will complete and not abort if the PIO Reply and ReRun bus operations receive ASTATIN AParErr and will execute them as if no parity error was detected. The master will not reissue these bus operation if ASTATIN is AParErr.

8.4.12 Address Response Null

AResp Null indicates that one of the following conditions is true:

- This device does not cache data.
- This device does cache data but the block does not exist in this cache, which is referred to as the invalid state.
- This device does cache data, the block is not invalid and the null response is allowed (for example, Null for Clean on block marked E).

8.4.13 Address Response Shared

AResp Shared indicates that this cache will maintain a copy of the block marked shared. ARESPOUT Shared may be issued by a bus device to indicate that a copy of the addressed block may be cached in the shared state. The AResp collection logic must be able to handle receiving ARESPOUT shared and modified and prioritize modified as described in Table 8-11.

8.4.14 Address Response ReRun

AResp ReRun response indicates that the response is not available to one or more bus devices and that the bus operation will need to be run again. Refer to Section 8.13, "The ReRun Mechanism."

8.4.15 Address Response Reserved

No bus device should issue AResp Reserved. A 620 snooper will treat ARESPIN Reserved as a retry. A 620 master will treat ARESPIN Reserved as an abort, will initiate a machine check exception and will set the BUSRESPERR status bit. See Section 2.1.2.4, "Bus Status and Control Register (BUSCSR)."

8.4.16 Address Response Modified

AResp Modified indicates that a snooper or bus adapter has an exclusive modified copy of the addressed cache block and that the snooper or bus adapter will provide the exclusive modified copy of the cache block.

8.4.16.1 Modified vs ReRun and Shared

The Modified response dominates over the ReRun and Shared responses because the Modified response guarantees in a coherent cache-memory system that all other snoopers must have the addressed block marked Invalid.

8.4.16.2 ARESPOUT Modified Assertion

Note that ARESPOUT Modified is never asserted by the 620 as a master. ARESPOUT Modified will never be asserted by a snooper except for the following conditions.

- Read-Burst or RWITM and N = 1: The 620 as a snooper will assert ARESPOUT Modified for the Read-Burst or RWITM bus operations if the addressed block is modified (M) and bus intervention is enabled (N=1). If ARESPIN for this case is Modified (not Retry) then the 620 will use the intervention mechanism to supply the modified cache block. If ARESPIN for this case is Retry, then the 620 will not alter the cache state and will not push the block.
- Flush or Clean: The 620 as a snooper will assert ARESPOUT Modified for the Flush or Clean bus operations if the addressed block is modified (M). If ARESPIN for this case is Modified (not Retry), then the 620 will push the modified cache block and mark the cache I (Flush) or S/E (Clean). If ARESPIN for this case is Retry, then the 620 will not alter the cache state and will not push the block.

ARESPIN Modified is handled by the 620 in the following manner:

- Snooper: ARESPIN for a snooper only specifies Retry or not-retry. ARESPIN Modified is treated like a non-retry ARESPIN.
- Master and Read-Burst or RWITM: The 620 will wait for the snooper to provide the exclusive modified block via the intervention mechanism.
- Master and all other bus operations: The 620 will ignore the Modified response. There is no exception or state indicating that this condition occurred.

For more information, refer to:

- Section 8.12.3, “Read”
- Section 8.12.1, “RWITM (Read-With-Intent-To-Modify)”
- Section 8.17.14, “Modified (M)”
- Section 8.17.18, “Snooper Cache State Transitions Due to Bus Operations”
- Section 8.17.8, “Multi-Level Cache Definition”
- Section 8.7, “Intervention and Push Definition
- Section 8.10.3, “Bus Tag Allocation/Deallocation”

8.4.17 Address Status Out and Address Response Out Retry

The term ‘retry’, within the context of the address transfer protocol, is defined as follows:

- ASTATOUT Retry is asserted in general by bus devices for the following reasons:
  - — Memory and snooper: The memory or snooper does not have buffer space for the address or data operation.
  - — AResp disabled and ASTATOUT Retry: With the exception of TLBIE and IKILL, when AResp is disabled then only the addressed slave can assert ASTATOUT Retry; refer to Table 8-1. This enables a slave for an AResp-

disabled operation to proceed with the operation without waiting for ASTATIN.

- ASTATOUT Retry is asserted by the 620 for the following reasons:
  - — Master and CD previous adjacent: The master will ASTATOUT Retry itself if the CD previous adjacent condition is true. Section 8.18.3.2, "Rule 2: CD Previous Adjacent ASTATOUT Retry."
  - — Snooper and no snoop buffer available: The snooper does not have a snoop buffer for the bus operation.
  - — Snooper and no snoop state machine available for TI, SY, TS and IK: These operations require a snoop state machine immediately after being received and thus flow control using ASTATOUT Retry if a snoop state machine is not available.
  - — Snooper working on TI and a second TI is received: The 620 will only work on one TI (TLBIE) at a time; see Section 2.3.6.3.3, "Translation Lookaside Buffer Management Instructions."
- ARESPOUT Retry is asserted by the 620 for the following reasons:
  - — Master and lost collision detection—A master operation will retry itself if collision detection is lost. Section 8.18, "Address Collision Detection and Handling (CD)."
  - — Master and atomic: A master will retry itself if the reservation is lost.
  - — Snooper and push condition: A snooper will retry a bus operation if a push is needed.
  - — Snooper and CD: The snooper will retry a bus operation due to CD. See Section 8.18.3.4, "Rule 4: Operations that Take CD Priority," Section 8.18.3.5, "Rule 5: CD Based on Completion," and Section 8.18.3.6, "Rule 6: CD Between Snoop Buffers."
  - — Snooper and BUSTLAR: If BUSTLAR is set below the minimum value specified in Table 8-3.
  - — Snooper and no snoop state machine available: The snooper will retry a bus operation if a snoop state machine is needed and a snoop state machine is not available.
  - — Snooper and L2 uncorrectable error: The snooper will retry a bus operation if there is an uncorrectable error for the L2 access to determine the coherency state.

Refer to Section 8.10.3, "Bus Tag Allocation/Deallocation," for more information.

8.4.18 ASTATIN and ARESPIN Retry

A bus operation that gets AStat or AResp Retried does not have to be reissued to the bus again by the master with the same address, type or tag. Although the present implementation of the 620, with a few exceptions, will reissue a bus operation with the same address, type and tag, compatibility with future implementations is not guaranteed.

A bus operation that gets AStat or AResp Retried does not have to be the next bus operation from the master that was retried.

8.4.18.1 ASTATIN Retry for All Bus Devices

A bus operation that gets ASTATIN Retried will be aborted one BUSCLK from sampling ASTATIN Retry. Aborted means that the ASTATIN Retried bus operation will not cause any subsequent bus operations to be ASTATOUT or ARESPOUT Retried due to a Queue Full condition. See Section 8.22.1, "The Queue Full/Ping-Pong Deadlock."

8.4.18.2 ARESPIN Retry for All Bus Devices

A bus operation that gets ARESPIN Retried may take a variable amount of time to clear up queue resources. Note that this does not affect when the bus tags (Address/Data) are deallocated.

8.4.18.3 ASTATIN and ARESPIN Retry Master

The 620 as a master will internally rearbitrate following a Retry. The 620 may run the same operation again or may internally grant a different operation.

With the exception of Write-With-Flush, writes that aren't atomic may complete independent of ARESPIN Retry. For Write-With-Flush (either Atomic or Non-Atomic), the write is conditional based on the response. For more information refer to Section 8.12.6, "Write-With-Flush."

8.4.18.4 ASTATIN and ARESPIN Retry Snooper

The 620 handles ASTATIN Retry as described above. All bus operations that are AResp-enabled that receive ARESPIN Retry will be aborted by the 620 snooper and leave the cache state unchanged. For more information refer to Section 8.10, "Bus Tags."

8.4.19 Function of ASTATOUT/ASTATIN and ARESPOUT/ARESPIN

The purpose of this section is to define the function of ASTATOUT/IN and ARESPOUT/IN.

The following abbreviations are defined:

- The codes for the bus operations are defined in Table 8-16. "M" is the bus M-bit which is defined in Section 8.6.1.3, "Memory Coherent Address Attribute (M-Bit)."
- previous_adjacent_address_match is defined in Section 8.18.3.2, "Rule 2: CD Previous Adjacent ASTATOUT Retry."
- Ack_Is_OK is defined in Section 8.4.10, "Address Status Acknowledge."
- aresp_disabled_op is defined in Table 8-1.
- my_bus_operation indicates that this 620 master sourced the operation.
- address_parity_error_detected indicates incorrect address parity for snooped addresses.

- no_snoop_buffer_available indicates that all four snoop buffers are currently active.
- non_mc_snoop_op indicates one of the following—IKill, SYNC, TLBIE, TLBSYNC.
- no_snoop_state_machine indicates that the snooper has detected a snoop hit and that there is no snoop state machine available.
- cd_enabled_op and cd_disabled_op are defined in Section 8.18.3.3, “Rule 3: CD Disabled Bus Operations.”
  cd_is_not_ok:= (cd_enabled_op & ^cd_in);
- cistcx:= (WNB & atomic); STCX does not support write-through mode.
  stcx_succeed indicates whether the STCX instruction will succeed or fail.
  reservation is the LARX/STCX reservation.
- snoop_didnt_complete indicates that the 620 did not complete snooping into the L1, and optionally the L2, before generating ARESPOUT. Refer to Section 8.4.17, “Address Status Out and Address Response Out Retry.”
- push_retry_condition indicates that the operation will get AResp Retried and will cause a push or intervention; see Table 8-48.
- SY_or_TS_arent_done indicates that the 620 has not completed the execution of the SYNC or TLBSYNC bus snoop operation. This completion is dependent only on the state of the 620 and is not affected by other masters continuing to ReRun the SYNC or TLBSYNC.

Following are additional functions of AStat and AResp in the form of an “If-Then-Else” and “Switch statement” equations:

Master Address Status Out/In Definition

```
ASTATOUT:
        If (previous_adjacent_address_match)
         ASTATOUT:= Retry;
        Else
         ASTATOUT:= Null;
ASTATIN:
        AParErr:
         1. Set the BUSPARERR[0] bit.
         2. ARESPOUT/IN is not valid. The master bus operation is complete.
         3. If BUSPOSACKEN = 1, issue a machine-check exception or Checkstop if MSR(ME)=0.
        Retry:
         1. ARESPOUT/IN is not valid. The master bus operation is complete.
         2. Start master bus operation over again.
        ^Ack_Is_OK:
         1. Set the BUSPOSACKERR bit.
         2. ARESPOUT/IN is not valid. The master bus operation is complete.
         3. Issue a machine-check exception or Checkstop if MSR(ME)=0.
        Ack_Is_OK:
         If (aresp_disabled_op)
            ARESPOUT/IN is not valid. The master bus operation is complete.
         Else
            ARESPIN/OUT is valid. Refer to the ARESPOUT/IN figures for the master.
```

Snooper Address Status Out/In Definition

```
ASTATOUT:
          If (^my_bus_operation & address_parity_error_detected & EBA)
           ASTATOUT:= AParErr;
          Else
           If (BUSSNPEN & ^my_bus_operation & (no_snoop_buffer_available |

non_mc_snoop_op & no_snoop_hit_buffer))
               ASTATOUT:= Retry;
           Else
               ASTATOUT:= Null;
ASTATIN:
          If (Retry | AParErr | ^Ack_Is_OK):
           ARESPOUT/IN is not valid. The snoop bus operation is aborted.
           If (AParErr):
               Issue a machine-check exception or Checkstop if MSR(ME)=0.
          Else
           If (aresp_disabled_op)
               ARESPOUT/IN is not valid. The snoop bus operation is complete.
           Else
               ARESPIN/OUT is valid. Refer to the ARESPOUT/IN figures for the snooper.
```

Master Address Response Out/In Definition

```
ARESPOUT:
           If (cd_is_not_ok | (cistcx & ^reservation))
               ARESPOUT:= Retry;
           Else
               ARESPOUT:= Null;
ARESPIN:
           Retry:
               (cistcx & ^reservation):
                   1. stcx_succeed:= false;
                   2. The master operation is complete. The STCX instruction failed.
               (cd_is_not_ok):
                   (LR | DC):
                       The master operation gets retried back to the load/store unit. [see Note]
                   All Other Operations:
                       1. The master operation is NOT complete.
                       2. The master operation waits in the BIU until collision detection is passed.
                       3. Go back to the bus with the same bus operation.
               All other cases:
                   1. The master operation is NOT complete.
                   2. Go back to the bus with the same bus operation.
           ReRun:
               1. The master operation is NOT complete.
               2. The operation is "ReRun" according to Section 8.13, "The ReRun Mechanism."
           (Null | Modified | Shared):
               The master operation is complete.
               If (atomic)
                   stcx_succeed:= true;
               Else
                   stcx_succeed:= false;
```

Note: When Larx-Reserve and DClaim lose collision detection it may be due to a snoop operation that has changed the state of the L1 data cache. Because Larx-Reserve and DClaim are dependent on the state of the L1 data cache, unlike all other operations from the DCMMU, the conservative solution chosen is to retry the operation back to the load store unit and the operation starts over again.

Snooper Address Response Out Definition

```
ARESPOUT:
            If (cd_is_not_ok | snoop_didnt_complete | (^non_mc_snoop_op & no_snoop_hit_buffer) |
             push_retry_condition)
             ARESPOUT:= Retry;
            Else
             If (SY_or_TS_arent_done)
                 ARESPOUT:= ReRun;
             Else
                 ARESPOUT:= (Null, Modified, Shared); [see Note]
```

Note: The response is determined by Table 8-48.

The snooper $\overline{\text{ARESPIN}}$ definition is described in Table 8-14. Refer to Section 8.18, "Address Collision Detection and Handling (CD)," for more information.

Table 8-14. Snooper ARESPIN Definition

Operation(s)	ARESPIN	Definition
SY or TS (All operations other than SYNC and TLBSYNC)	Null, Shared, Modified, ReRun, Retry (Push)	1. If the snoop CD is "IN"[1] at ARESPIN and ARESPIN-Clean then force the CD for all master and snooper devices from IN to OUT for each SB[2] bit that is set. ARESPIN-Clean = (Null | Shared | Modified | Retry(Push) | (ReRun & E-State)); 2. Clear all SB bits for that snooper at ARESPIN. 3. If the snooper determines that a "cache state change is not needed" prior to ARESPIN then the snooper may drop the snoop address and inhibit subsequent collisions against that snoop operation. See Section 8.17.18, "Snooper Cache State Transitions Due to Bus Operations," for the definition of "cache state change is needed". 4. If a "cache state change is needed" then go from IN to INBUSY at ARESPIN and modify the cache and move data according to Table 8-48. Stay INBUSY until the cache modification is complete and then go OUT. 5. The snoop operation is done.
	Retry ($\overline{\text{Push}}$)	1. Clear all SB bits for that snoop buffer. 2. The snoop operation is aborted.
SYNC	Null	The snooper has completed the SYNC.
	ReRun	If ARESPOUT was ReRun, the snooper has not completed the SYNC. If ARESPOUT was not ReRun, the snooper has completed the SYNC.
	Retry	The snooper will abort the snooped SYNC. Section 2.3.4.7, "Memory Synchronization Instructions."
TLBSYNC	(Ignored)	TLBSYNC ignores ARESPIN. See Section 2.3.6.3.3, "Translation Lookaside Buffer Management Instructions."

Notes:

1. "OUT", "IN" and "INBUSY" are defined in Section 8.18.2, "CD States and State Transitions."
2. "SB"indicates whether there is a shared collision scoreboard bit asserted between this snoop buffer and a 620 master bus operation or another snoop buffer.

8.5 Data Bus Transfer Protocol

The latency from data bus grant to the beginning of the data bus transfer is not fixed and is dependent on the tenure of the data transfer already granted use of the data bus. The tenure of a data bus operation is variable and determined by the bus device that sources the data. The data transfer can be flow controlled by the producer, but not by the consumer. The data bus consists of the following signals:

- DH[0–63] and DL[0–63] (Data Bus)
- DTAG[0–7] (Data Bus Tag)
- $\overline{\text{DBB}}$ (Data Bus Busy)
- $\overline{\text{DVAL}}$[0–1] (Data Bus Valid)
- DPHigh[0–7], DPLow[0–7], DPCntl (Data Bus Parity)
- $\overline{\text{DERR}}$ (Data Bus Error)
- $\overline{\text{DCACHE}}$ (Data Cache) g

8.5.1 Data Bus Width and Interconnectivity

The 620 supports connectivity with quad-word width bus devices (non-DX mode) or double-word width bus devices (DX mode); refer to Section 8.5.10, "DX Mode (64-Bit Data Bus Width Mode)." The 620 does not support connectivity with word-width bus devices or to mixed double-word and quad-word devices. Connectivity to narrower bus widths is intended to be supported via a bus adapter that handles narrower width connectivity.

An exception is cache-inhibited and PIO bus devices that are allowed to connect to less than the full quad-word sized data bus as long as only the portion of the data bus that is connected to is addressed.

8.5.2 Data Bus (DL[0–63]/DH[0–63])

The data bus width is four words or 128-bits wide and is divided into two double-word parts. The least significant double-word is called DL[0–63]. The most significant double-word is called DH[0–63].

8.5.3 Data Sequence Ordering for Burst Operations

- All Burst Operations—The least significant 4 bits that point to the byte in quad-word (A[60–63]) should be ignored for all burst operations.
- All Burst Writes—All write burst bus operations, Write-With-Kill and Write-With-Clean, will be block aligned (the address points to quad-word 0). The sequence will appear as 0, 1, 2 and 3.
- All Burst Reads—The order that data is returned for burst read transfers, such as Read and RWITM, is called critical quad-word first. The byte address that caused the burst transfer points to a quad-word in a block and this is the first quad-word to

be transferred on the data bus. The sequence of quad-words that follows the quad-word pointed to by the address increments in a modulo-4 fashion that wraps around to the initial quad-word.

Table 8-15. Data Ordering for a Quad-Word Sized Bus

Quad-word Physical A[34–35]	Quad-word Ordering
0	0,1,2,3
1	1,2,3,0
2	2,3,0,1
3	3,0,1,2

8.5.4 Data Bus Tag

The data bus tag is 8 bits and is called DTAG[0–7]. Refer to Section 8.10, "Bus Tags," for more information.

8.5.4.1 Data Producer

The data producer for a read or write must drive the tag for all data transferred.

Bus devices such as the memory subsystem are allowed to produce multiple return data transfers during one data bus tenure. This means that the memory subsystem does not need to include a dead cycle between data transfers during the data bus tenure. Dead cycles are only required for handoffs between data bus tenures. The 620 need only see the change in data bus tag, qualified by one of the two $\overline{\text{DVAL}}$s, to recognize that a different data transfer is in progress. Note that data transfers cannot be interleaved— once a multi-beat transfer has started, it must complete before a different data bus tag can be presented. The 620 will not return multiple data transfers during one data bus tenure.

8

8.5.4.2 Data Consumer

The read data consumer will snoop the data bus tag and consume each data that tag matches a pending read that is waiting for data. The read data consumer can assume that read data for different read transfers will not be interleaved. The read data consumer must be able to consume zero or more data transfers for itself from a data bus tenure that includes zero or more data for other read data consumers. The consumer must not expect data after the deallocation of the data bus tag. Refer to Section 8.10.3, "Bus Tag Allocation/Deallocation."

8.5.4.3 Data Bus Tag Interleaving

The 620 bus does not support the interleaving of multiple transfers by the data producer. When a transfer is started then all data of that transfer must be transferred together allowing for producer flow control.

8.5.5 Data Bus Busy ($\overline{\text{DBB}}$)

$\overline{\text{DBB}}$ is asserted for all but the last data of the last transfer of a data bus tenure. $\overline{\text{DBB}}$ can also be thought to indicate that “this is not the last BUSCLK of this data bus tenure”. Figure 8-19 illustrates when $\overline{\text{DBB}}$ is asserted for 1, 2, 3 and *n* BUSCLK data bus tenures.

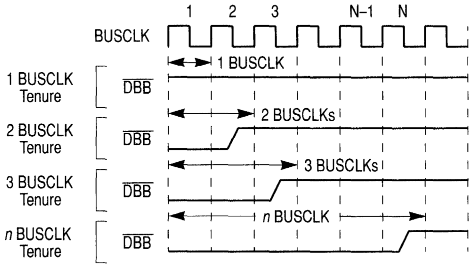


Figure 8-19. Definition of $\overline{\text{DBB}}$

8.5.6 Data Valid ($\overline{\text{DVAL}}$[0–1])

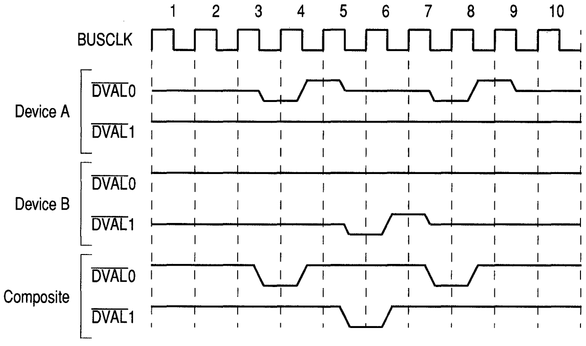
The data bus master, or the data producer, drives $\overline{\text{DVAL}}$[0–1] to indicate whether this cycle has valid data. This implements flow control by the data producer. The data consumer cannot flow control the data producer.

If a memory bus device does not have room for the transfer then it retries the address. If a master has requested data, it must have room for the return data.

The following information provides a definition for $\overline{\text{DVAL}}$[0–1].

- $\overline{\text{DVAL}}$[0–1] Producers
  - — $\overline{\text{DVAL0}}$ should be driven enabled and asserted if $\overline{\text{DVAL0}}$ was not asserted two cycles previously by another bus device.
  - — $\overline{\text{DVAL1}}$ should be driven enabled and asserted if $\overline{\text{DVAL0}}$ was asserted two cycles previously by another bus device. (Note: A bus device has the option of not implementing a driver for $\overline{\text{DVAL1}}$ and may choose to wait one extra bus cycle in order to avoid $\overline{\text{DVAL0}}$ contention. This is a cost/performance trade-off.)
  - — $\overline{\text{DVAL}}$[0–1] will switch between data bus tenures, but not within a data bus tenure.
- $\overline{\text{DVAL}}$[0–1] Consumers
  - — $\overline{\text{DVAL}}$ consumers should consider $\overline{\text{DVAL}}$ asserted if either $\overline{\text{DVAL0}}$ or $\overline{\text{DVAL1}}$ is asserted.
  - — A $\overline{\text{DVAL}}$ driver that has driven either $\overline{\text{DVAL}}$ signal asserted will drive that signal deasserted for one cycle before disabling.

Figure 8-20 provides the timing for DVAL[0–1].



Notes:

- Device A (cycle 3) drives DVAL0 because DVAL0 was not being used two cycles previously (cycle 1).
- Device B (cycle 5) drives DVAL1 because DVAL0 was being used two cycles previously (cycle 3).
- Device A (cycle 7) drives DVAL0 because DVAL0 was not being used two cycles previously (cycle 5).

Figure 8-20. Definition of DVAL[0–1]

8.5.6.1 Restrictions on Flow Control

The 620 as a data producer will not flow control data and will provide four contiguous data quad-words. The 620 as a data consumer, for memory or intervention (DCACHE) data, will always allow data to be flow controlled by the producer.

An exception to the previous rule is that a processor that is not allowed to flow control may hold off the beginning of a data transfer with DVAL, but not after the first data is transferred. It should be noted that this hurts data bus bandwidth and should be avoided. The 620 does not plan to hold off data transfers in this manner. The 620 will always assert DVAL on the first bus cycle of the data bus tenure.

8.5.6.2 Uses of Data Producer Flow Control

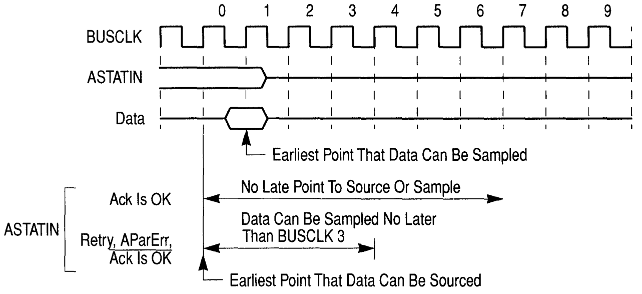
Data producer flow control can be used for many purposes, a few of which are listed as follows:

- A producer detects a correctable error for a transfer after the transfer has begun, and needs a cycle or more to correct the error and continue the data transfer.
- The bandwidth of memory may not be as high as the bus and the memory will want to get the critical data to the bus as quickly as possible and transfer the rest of the data as it becomes available.

8.5.7 Minimum Latency to Sample Read Data

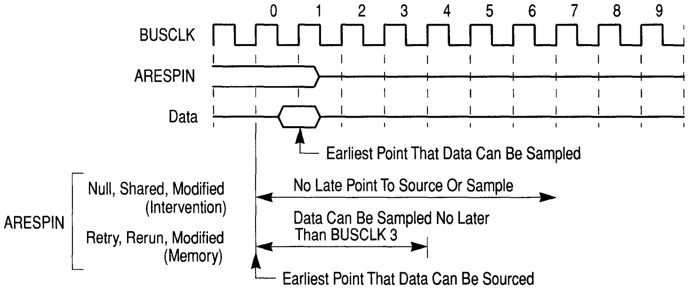
This section describes the minimum latency that data can be supplied for the Read, RWITM, PIO Load Immediate and PIO Load Last bus operations. Note that this rule does not state the minimum latency of $\overline{\text{DBR}}$ and $\overline{\text{DBG}}$; see Figure 8-21 and Figure 8-22.

If AResp is disabled then received data may be sampled by the 620 no sooner than sampling ASTATIN. If AResp is enabled then received data may be sampled by the 620 no sooner than sampling ARESPIN, refer to Section 8.4.3.2, "Address Status and Address Response Validation." Ack_Is_OK is defined in Section 8.4.10, "Address Status Acknowledge." Tag allocation and deallocation are defined in Section 8.10, "Bus Tags."



0 1 2 3 4 5 6 7 8 9
BUSCLK
ASTATIN
Data
Earliest Point That Data Can Be Sampled
ASTATIN
Ack Is OK
No Late Point To Source Or Sample
Retry, AParErr, $\overline{\text{Ack Is OK}}$
Data Can Be Sampled No Later Than BUSCLK 3
Earliest Point That Data Can Be Sourced

Figure 8-21. AResp-Disabled: Earliest Time to Sample Received Data



0 1 2 3 4 5 6 7 8 9
BUSCLK
ARESPIN
Data
Earliest Point That Data Can Be Sampled
ARESPIN
Null, Shared, Modified (Intervention)
No Late Point To Source Or Sample
Retry, Rerun, Modified (Memory)
Data Can Be Sampled No Later Than BUSCLK 3
Earliest Point That Data Can Be Sourced

Figure 8-22. AResp-Enabled: Earliest Time to Sample Received Data

8.5.8 Data Cache ($\overline{\text{DCACHE}}$) Signal

The function of the $\overline{\text{DCACHE}}$ signal is to distinguish cache (intervention) data from memory data. Refer to Section •, "Definition of Intervention—An Intervention is when a Read burst or RWITM bus operation is given the AResp modified response by a snooper that holds the modified data and the modified data is supplied by that snooper as a data-only transfer. Note that the minimum latency to sample intervention data is defined in Section 8.5.7, "Minimum Latency to Sample Read Data.."

8.5.8.1 An Intervention Data Producer—Cache

DCACHE is valid only when DVAL is asserted. DCACHE is driven asserted only for the first data that is transferred and deasserted for all remaining data that are transferred. DCACHE may only be driven asserted for a block transfer that consists of two or more data transfer cycles. The 620 only supports 64-byte sized blocks, requiring four data transfer cycles.

8.5.8.2 A Memory Data Producer—Memory

A device, such as memory, that does not care to drive DCACHE asserted is allowed to not implement either a driver to drive DCACHE or an input buffer to receive DCACHE, and may assume that DCACHE is deasserted for parity generation and checking. The bus N-bit must never be asserted by a bus device that does not implement the DCACHE input.

8.5.8.3 A Memory and Intervention Data Consumer (Read or RWITM Requestor)

The data consumer monitors the DCACHE signal for the first data transferred. The state of the DCACHE signal for the first data determines if the read will be serviced by an intervention. The data consumer may rely on the DCACHE signal to be deasserted for all remaining data transfers in the block. The timing diagram in Figure 8-23 is a block transfer from a cache.

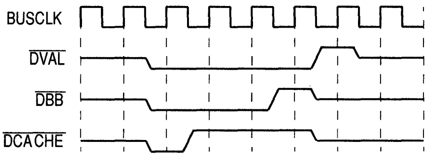


Figure 8-23. DCACHE Timing Diagram for an Intervention Data Transfer

8.5.9 Data Error (DERR)

DERR is driven by the data producer two BUSCLK cycles from when DVAL is asserted to indicate whether this data transfer contains an error. A device that does not care to drive DERR asserted must still drive DERR deasserted. The two BUSCLK cycle delay allows data crossbar or sliced implementations to pass DERR signals for each data slice to the address controller to produce the 620 DERR signal, see Figure 8-24.

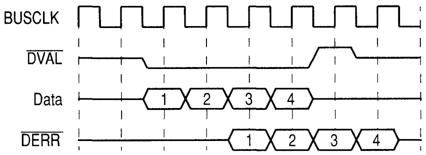


Figure 8-24. $\overline{\text{DERR}}$ Timing Diagram

- Memory Read or PIO Load Immediate—The assertion of $\overline{\text{DERR}}$ for these operations will not prevent the error data from being loaded into the register or caches. A $\overline{\text{DERR}}$ for a Memory Read or PIO Load Immediate bus operation will always set the SPR bit BUSDERR and will cause a machine-check exception if the SPR bit BUSDERREN is asserted. See Section 2.1.2.4, "Bus Status and Control Register (BUSCSR)," and Section 2.1.2.3, "Hardware Implementation-Dependent Register 0 (HID0)."
- PIO Load Last—A $\overline{\text{DERR}}$ for a PIO Load Last will cause a DSI exception. The data will not be loaded into the register. If a data parity occurs for a PIO load last error occurs then there is no guaranteed order between the DSI and MCI exceptions.

Note: The assertion of $\overline{\text{DERR}}$ associated with a data transfer will not affect the number of data transferred, which must be the number of data requested, independent of whether the data transferred are valid.

In general, bus devices will drive $\overline{\text{DERR}}$ asserted for the following reasons:

- The data producer is producing error data (for example, 2-bit ECC memory error).
- The PIO load detected an error and the PIO master should expect a PIO load reply. Section 8.16, "PIO Load and Store Bus Operations."

8

In specific, the 620 will drive $\overline{\text{DERR}}$ asserted for the following reasons:

- Snooper and push or intervention error—The 620 as a snooper for a push or intervention detects an L1 parity error while moving data from the L1 or detects an uncorrectable L2 error while moving data from the L2.
- Master and replacement copyback error—The 620 for a replacement copyback detects an L1 parity error while moving data from the L1 or detects an uncorrectable L2 error while moving data from the L2.

8.5.10 DX Mode (64-Bit Data Bus Width Mode)

The mode of operation is non-DX mode, unless otherwise specified. It is to be assumed that all deviations in behavior due to DX mode are defined in this section.

8.5.10.1 DX Mode Definition

The 620 data bus has a power-up hardware configuration mode that changes the data bus width from the native quad-word size to the special double-word size. Instead of passing quad-word data transfers on DH[0–63] and DL[0–63], double-word data transfers are passed only on DH[0–63]. BUSDX is software readable from the BUSCSR SPR. See Section 2.1.2.4, "Bus Status and Control Register (BUSCSR)." BUSDX is assigned by the hardware configuration mechanism. BUSDX is assigned the value of the BUSDX signal (for example, when BUSDX is a 1, then BUSDX is assigned a 1 and is asserted); see Section 8.23, "Hardware Configuration Mechanism (HCM)."

8.5.10.2 Data Low (DL[0–63])

DL[0–63] and the associated parity bits DPLow[0–7] will be disabled internally and tied to ground by the external pins. Parity checking is disabled for DL[0–63] and DPLow[0–7] for DX mode.

8.5.10.3 Double-Word Ordering

The double word ordering for DX-mode is the same as the quad-word ordering of non-DX mode. Additionally, each quad-word is broken up into two double-words with big-endian double-word 0 first and double-word 1 second. This is true even if the critical load data is in double-word 1.

8.5.10.4 Address Bus

The data bus DX mode does not affect the definition of the address bus. Size for burst transfers always refers to the number of quad-word transfers. Size for non-burst transfers always refers to the number of bytes transferred up to a full quad-word or two double-words. Non-burst operations will always transfer two double-words.

8.5.10.5 Block Read Data Latency

The 620 will gather double-word data into quad-words through the bus load buffer and will not use the combinatorial data forwarding path. This will add 1 PCLK of load latency.

8.5.10.6 $\overline{\text{DERR}}$ for PIO Load Last

The $\overline{\text{DERR}}$ signal for PIO Load Last is used to indicate if the PIO should cause a DSI exception. $\overline{\text{DERR}}$ should either be asserted or deasserted for both data transfers. All other bus operations, including PIO Load Immediate, can assert $\overline{\text{DERR}}$ on a per data basis.

Example 1: DX Mode Non-Burst Operation

Non-Burst transfers are always two double-word transfers, even if the non-burst size is one double-word or smaller and there is no valid data in one of the double-word transfers. Double-word 0 is always transferred first even if the address points to double-word 1 as shown by the following example. $\overline{\text{DBB}}$ is asserted for the first of the two double-word transfers.

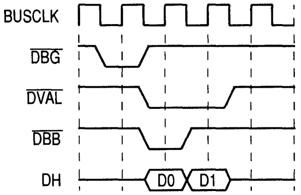


Figure 8-25. DX Mode: Non-Burst Operation: Addr=Double-Word 1

Example 2: DX Mode Burst Operation

Burst transfers are always eight double-word transfers. Double-word 0 is always transferred first even if the address points to double-word 1 as shown by the following example. $\overline{\text{DBB}}$ is asserted for the first seven of the eight double-word transfers.

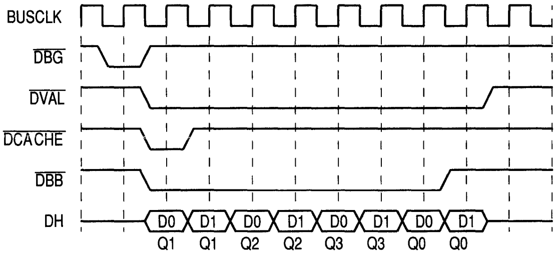


Figure 8-26. DX Mode: Burst Operation: Addr=Quad-Word1, Double-Word1

8.6 Address Commands Definition

The address command is defined by ATYPE[0–4] and A[0–7] according to Table 8-16. The following attributes are defined:

- 'X' is defined as deasserted or 0 for an output and a don't care as an input.
- '0' or '1' is defined as a logical 0 or 1 for both an input and an output.
- 'M'—Memory Coherent (Refer to Section 8.6.1.3, "Memory Coherent Address Attribute (M-Bit)"and Section 8.6.1, "WIM-Bit Definitions.")
- 'I'—Cache Inhibited (Refer to Section 8.6.1.2, "Cache-Inhibited Address Attribute (I-Bit).")
- 'A'—Atomic (Refer to Section 8.6.2, "Atomic Address Attribute (A-Bit).")
- 'N'—Intervention (Refer to Section 8.6.3, "Intervention Address Attribute (N-Bit)," Section 8.12.1, "RWITM (Read-With-Intent-To-Modify)," and Section 8.12.3, "Read.")
- 'K'—PIO Key (Refer to Section 8.9.3, "The PIO Request Address Format")

- 'E'—PIO Error (Refer to Section 8.9.4.1, "Error (E)")
- 'W'—Write Through (Refer to Section 8.6.1.1, "Write-Through Address Attribute (W-Bit)")
- 'S' and 'G'—The S and G bits are defined in Section 8.12.3, "Read."
- 'R'—ReRun (Refer to Section 8.13.3, "The R-Bit")

The terminology is defined as follows:

- **ATYPE** is the bus operation code. The encoding of the least significant 4 bits for all bus operations except for PIO bus operations should be the same as the 60x implementations.
- **Address** is the definition of the upper 8 bits for all address formats. Note that bit 2 is always defined as the bus M-bit. Note that bit [7] is the R-Bit for all AResp enabled operations. The "-" character indicates that the text defines the function of this bit. The "X" character indicates that this bit is undefined for the 620, driven as 0 as an output and ignored as an input.
- **620** indicates whether the 620 will issue or decode the specified bus operation. If the 620 **does not** issue or decode the specified bus operation, then this document will not define the function of the specified bus operation.
- **Operation** is the name of the bus operation. A definition under the same name can be found in Section 8.12, "Bus Operations."
- **Symbol/Code** is used throughout this document to denote the particular operation.
- **Address format** indicates the format of the bus operation, which are defined in Section 8.9, "Address Bus."
- **Address-Data Type** indicates whether the bus operation is address-only or address-data. An address-only bus operation that requests for a data-only bus operation is listed as address-only/data-only.
- **Page Num** provides a cross reference to a description of that bus operation.

Table 8-16. ATYPE[0–4] Definition

ATYPE [0–4]	A[0–7]	620	Operation	Symbol/ Code	Address Format	Address-Data Type	Page Num
00000	XXMXXXXR	Y	Clean	CL	Mem Req	Address-Only	73
A0010	WIMXXXXR	Y	Write-With-Flush	WNB	Mem Req	Address-Data	71
00100	XXMXXXXR	Y	Flush	FL	Mem Req	Address-Only	73
00110	WXM0XXXR	Y	Write-With-Kill	WBK	Mem Req	Address-Data	70
00110	WXM1XXXR	Y	Write-With-Clean	WBC	Mem Req	Address-Data	71
01000	XXMXXXXR	Y	SYNC	SY	Tag Only	Address-Only	73
A1010	NSMXGXXR	Y	Read	RB, RNB	Mem Req	A-Only/D-Only	69
01100	XXMXXXXR	Y	DKill	DK	Mem Req	Address-Only	72
A1110	NXMXXXXR	Y	RWITM	RWITM	Mem Req	A-Only/D-Only	68
10000	XXMXXXXX	Y	EIEIO	EI	Tag Only	Address-Only	73
10100	XXM-----	Y	External Control Out	XCO	ExtCon	Address-Data	74
11000	XXMXXXXX	Y	TLBIE	TI	Mem Req	Address-Only	73
11100	XXM-----	Y	External Control In	XCI	ExtCon	A-Only/D-Only	74
00001	XXMXXXXR	Y	LARX-Reserve	LR	Mem Req	Address-Only	69
A0011	XXMXXXXR	Y	DClaim	DC	Mem Req	Address-Only	72
01001	XXMXXXXR	Y	TLBSYNC	TS	Tag Only	Address-Only	73
01101	XXMXXXXX	Y	IKill	IK	Mem Req	Address-Only	72
10001	K0M0----	Y	PIO Load Immediate	PLI	PIO Req	A-Only/D-Only	82
10001	K0M1----	Y	PIO Load Last	PLL	PIO Req	A-Only/D-Only	82
10001	K1M0----	Y	PIO Store Immediate	PSI	PIO Req	Address-Data	82
10001	K1M1----	Y	PIO Store Last	PSL	PIO Req	Address-Data	82
10101	E1M1-XXX	Y	PIO reply	PR	PIO Rep	Address-Only	82
11101	XXMXXXXX	Y	ReRun	RR	Tag Only	Address-Only	75
11111	XXMXXXXX	Y	Null	Null	N/A	Address-Only	73
00101 00111 01011 01111 10101 10101 10101 10110 10111 11001 11011	XXMXXXXX XXMXXXXX XXMXXXXX XXMXXXXX X0M0XXXX X0M1XXXX X1M0XXXX XXMXXXXX XXMXXXXX XXMXXXXX XXMXXXXX	N	Reserved Treated as a Null operation by the 620 snooper. These codes are reserved for future PowerPC products. Use of these codes should be coordinated with PowerPC if forwards compatibility is desired.				

8.6.1 WIM-Bit Definitions

Table 8-17 defines the bus WIM-bits for all bus operations. The snooped column indicates YES if the bus snooper accepts a bus operation and NO if the bus snooper ignores a bus operation.

Operations that are marked M=0 in the bus M-Bit column must never appear on the bus M=1. There are known functional failures that will occur in the 620 if any M=0 bus operations appear as M=1.

Table 8-17. WIM-Bit Definition

Bus Operation	W-Bit Definition	I-Bit Definition	M-Bit Definition	Snoop Status
Clean	N/A	N/A	PTE M-Bit	Yes
Write-With-Flush	PTE W-Bit	PTE I-Bit	PTE M-Bit	Yes
Flush	N/A	N/A	PTE M-Bit	Yes
Write-With-Kill	page 70	N/A	page 70	Yes
Write-With-Clean	1	N/A	0	No
SYNC	N/A	N/A	0	Yes
Read	N/A	See Section 8.12.3, "Read."	PTE M-Bit	Yes
DKill	N/A	N/A	PTE M-Bit	Yes
RWITM	N/A	N/A	PTE M-Bit	Yes
EIEIO	N/A	N/A	0	No
External Control Out	N/A	N/A	0	No
TLBIE	N/A	N/A	0	Yes
External Control In	N/A	N/A	0	No
LARX-Reserve	N/A	N/A	0	No
DClaim	N/A	N/A	PTE M-Bit	Yes
TLBSYNC	N/A	N/A	0	Yes
IKill	N/A	N/A	PTE M-Bit	Yes
PIO Load Immediate	N/A	N/A	0	No
PIO Load Last	N/A	N/A	0	No
PIO Store Immediate	N/A	N/A	0	No
PIO Store Last	N/A	N/A	0	No
PIO reply	N/A	N/A	0	Yes
ReRun	N/A	N/A	0	Yes
Null	N/A	N/A	0	No
Reserved	N/A	N/A	0	No

8.6.1.1 Write-Through Address Attribute (W-Bit)

The write-through W-bit indicates whether a bus operation needs to be propagated at least one level or all levels towards main memory; see Table 8-18.

Table 8-18. Write-Through Address Attribute (W-Bit)

W	Definition
1	The bus operation must be propagated through all cache levels to main memory.
0	The bus operation must be propagated at least one cache level towards main memory.

8.6.1.2 Cache-Inhibited Address Attribute (I-Bit)

The cache inhibited I-bit indicates whether the bus operation is cache inhibited; see Table 8-19.

Table 8-19. Cache Inhibited Address Attribute (I-Bit)

I	Definition
0	Cache enabled
1	Cache inhibited

8.6.1.3 Memory Coherent Address Attribute (M-Bit)

The 620 source for this signal is specified for each bus operation. Section 8.6.1, "WIM-Bit Definitions." Since A[2] is defined as the M-bit for all ATYPE codes, a snooper can determine if a memory-coherent bus operation needs to be snooped exclusively by the M-bit.

Table 8-20 defines the significance of the M-bit for the two types of snoopers, snoopers for other L1 and L2 caches (horizontal) and the snooper for the L3 (vertical). Refer to Figure 8-32. Refer to Section 8.17.10, "Cache Coherency Protocol," for the definition of vertical and horizontal cache coherence.

Table 8-20. Memory Coherent Address Attribute (M-Bit)

L1,L2,L3	M	Definition
L1-L2	1	Memory coherent, must be snooped.
	0	Not memory coherent, may be snooped.
L3	X	Must be snooped independent of the M-Bit to preserve coherence between levels.

8.6.2 Atomic Address Attribute (A-Bit)

The A signal indicates whether this transaction was caused by a LARX or STCX instruction; see Table 8-21.

Table 8-21. Atomic Address Attribute (A-Bit)

A	Definition
1	Caused by a LARX or STCX instruction
0	Not caused by a LARX or STCX instruction

8.6.3 Intervention Address Attribute (N-Bit)

The definition of intervention is defined in Section 8.7, "Intervention and Push Definition."

Intervention address attribute N-bit is the enable for intervention and is defined as shown in Table 8-22.

Table 8-22. Intervention Address Attribute (N-Bit)

N	Definition
0	Intervention disabled
1	Intervention enabled

8.7 Intervention and Push Definition

When the 620 snoops a bus operation and determines that the addressed block is cached modified in either the L1 or L2, then the 620 will supply the modified block using either the intervention mechanism or the push mechanism. The following table defines which mechanism is used for all possible cases.

Table 8-23. Use of Intervention and Push on the PowerPC 620 Microprocessor

Bus Operation	Bus N-Bit	Intervention/Push
Read	0	Push (Write with Clean)
	1	Intervention
RWITM	0	Push (Write with Kill)
	1	Intervention
Write with Flush Flush	N/A	Push (Write with Kill)
Read Non-Burst Clean	N/A	Push (Write with Clean)

For more information, refer to Section 6.7, “Snoop Push/Intervention Latency,” Section 8.17.18, “Snooper Cache State Transitions Due to Bus Operations,” and Section 8.12, “Bus Operations.”

- Definition of Push—A push is when the modified data is written back to memory with either a Write with Kill or Write with Clean.
  - SYNC Bus Operation—A snooped SYNC bus operation will wait until all pushes have been completed on the bus. Section 2.3.4.7, “Memory Synchronization Instructions.” Note that the EIEIO bus operation is not snooped by the 620.
  - Collision Detection—All bus operations that are snooped and address match with an internal push operation, and are not CD-disabled, as defined in Section 8.18.3.3, “Rule 3: CD Disabled Bus Operations,” will be AResp Retried until that push has completed on the bus.
- Definition of Intervention—An Intervention is when a Read burst or RWITM bus operation is given the AResp modified response by a snooper that holds the modified data and the modified data is supplied by that snooper as a data-only transfer. Note that the minimum latency to sample intervention data is defined in Section 8.5.7, “Minimum Latency to Sample Read Data.
  - Read Burst—Intervention for a Read Burst bus operation is defined as a “cache to memory and cache” transfer, for which the cache supplies the data to the master and the memory captures the data as it is sent on the data bus. Therefore, the cache is the intervention data producer and the memory and Read requestor are both intervention data consumers.
  - RWITM—Intervention for a RWITM bus operation is defined as a “cache to cache” transfer, for which the cache supplies the data to the master. The memory does not capture the data as it is sent on the data bus. Therefore, the cache is the intervention data producer and the RWITM requestor is the intervention data consumer.
  - Intervention Data Producer (Snooper)—The bus architecture specifies that the snooper which produces the intervention data can supply the data such that it is sampled by the master no sooner than when ARESPIN is sampled. The 620 will not supply the modified data until determining that ARESPIN is Modified. The intervention data producer will request for a data-only operation and will indicate high-priority by asserting the $\overline{\text{HPR}}$ signal. The intervention data producer supplies the intervention data, using the same bus tag value for the data tag that was received from the address bus tag, and asserts the $\overline{\text{DCACHE}}$ signal. The $\overline{\text{DCACHE}}$ signal indicates to the intervention data consumer(s) that this data is the intervention data, as opposed to the memory data.
  - Intervention Data Consumer(s)—The bus architecture states that the intervention data can be sampled no sooner than when ARESPIN is sampled. The master that requested the Read or RWITM may receive both memory and intervention data and will use the $\overline{\text{DCACHE}}$ signal to distinguish between them.

For more information, refer to:

- Section 8.5.8, "Data Cache (DCACHE) Signal"
- Section 8.3.1, "Arbitration Requests"
- Section 8.3.2, "High-Priority Bus Operations"
- Section 8.10, "Bus Tags"
- Section 8.12.3, "Read"
- Section 8.12.1, "RWITM (Read-With-Intent-To-Modify)"

8.7.1 Bus Intervention Bit (N-Bit)

The intervention N-bit address attribute signal indicates that the master and the memory system support intervention. Each master that supports intervention must be configured to know whether the memory system supports intervention and should assert the N-bit only when the memory system supports intervention.

The intervention address attribute is defined in Section 8.6.3, "Intervention Address Attribute (N-Bit).

The 620 snooper looks at the bus N-bit to determine if intervention is enabled and not the 620 intervention enable bit. Even if the master and the memory system support intervention, the snooper that has the modified block may or may not support intervention. If the snooper does not support intervention, then the snooper behaves as if the bus intervention bit was deasserted or 0 by retrying the master and writing the block back to memory.

The 620 always supports intervention.

8.7.2 Intervention Enable Bit (BUSINTVEN)

The 620 as a master and a snooper supports intervention. The SPR mode bit BUSINTVEN indicates that memory, external to the 620, supports intervention. If BUSINTVEN is asserted, enabling intervention, then the 620 for Read and RWITM bus operations will assert the intervention N-bit, otherwise it will be deasserted.

8.7.3 Non-Block Sized or Cache Inhibited

Intervention is only supported for block-sized (64-byte) burst RWITM and Read transfers. It is the responsibility of the master for a Read that is intervention-enabled to ensure that the read is cache-enabled and block-sized. The 620 snooper will assume that intervention enabled read requests are cache enabled and block sized and reserves the option to ignore the size and bus I-bit.

8.8 ASIZEDATA[0–3] and ASIZEBURST Definition

The address bus signals ASIZEDATA[0–3] and ASIZEBURST specify the size of the data bus transfer. Refer to Section 8.4, "Address Bus Transfer Protocol."

Table 8-24 defines for each of the three address bus operation types whether the signals ASIZEDATA[0–3] and ASIZEBURST are defined. If these signals are not defined their value is not significant for that address bus operation transfer. Refer to Table 8-16 for the definition of Address-Data Type.

Table 8-24. ASIZEDATA[0–3] and ASIZEBURST Validation

Address-Data Type	Status	Bus Operations
Address-Data	Valid	WNB, WBK, WBC, XCO, PSI, PSL
Address-Only/Data-Only	Valid	RB, RNB, RWITM, XCI, PLI, PLL
Address-Only	Invalid	CL, FL, SY, DK, EI, TI, LR, DC, TS, IK, PR, RR, Null

ASIZEDATA[0–3] and ASIZEBURST are defined in Table 8-25. If ASIZEBURST is deasserted, then ASIZEDATA indicates the size of the data transfer in bytes. (Note that the 0 code is a 16 byte size transfer.) If ASIZEBURST is asserted, or a 0, then ASIZEDATA indicates the size of the data transfer in full data bus width transfers. (Note that the 0 code is 16 bus width size transfers.) The 620 data bus width is quad-word sized, which means the unit for ASIZEDATA for burst mode is 16 bytes. Refer to Section 8.12, "Bus Operations," for a definition of the ASIZEBURST and ASIZEDATA codes that are supported by the 620 for each bus operation.

Table 8-25. ASIZEDATA[0–3] and ASIZEBURST Definition

ASIZEBURST	ASIZEDATA[0–3]	Definition
1	0001-1111, 0000	Transfer size in bytes. 1–15, 16
0	0001-1111, 0000	Transfer size in full data bus width transfers. 1–15, 16

The following sub-sections discuss data alignment on the 620.

8.8.1 Supported Burst Data Sizes and Alignments

The only supported burst data size for the 620 as a master is 4, which specifies four bus-width or quad-word transfers (64 bytes). The four quad-word burst size is the address block size, the coherency block size and the transfer block size for the 620. The 620 will not issue any other burst transfer size.

8.8.2 Burst Reads

The least significant 4 bits, physical address[36–39], address the byte requested in a quad-word. These bits should be ignored by read data producers (memory and cache) and read data is supplied as aligned quad-words assuming the byte address is b'0000'.

8.8.3 Burst Writes

The least significant 4 bits, physical address[36–39], address byte 0 and are driven as b'0000'.

8.8.4 Non-Burst Data is Always Big-Endian Aligned

Non-Burst data on the data bus is always big-endian aligned on a quad-word sized data bus. This applies to all data transfers, PIO and memory. ("Big-Endian", as opposed to "Little-Endian", "MS or Left-Justified", or "LS or Right-Justified." The address for big-endian alignment always points to the most significant or "big end" byte of the data being addressed. Examples of big-endian transfers can be found in tables on Figure 8-27 and Figure 8-28.)

8.8.5 Supported Non-Burst Data Sizes and Alignments

Table 8-26 defines the supported non-burst data sizes and alignments as a master. Figure 8-27 defines the supported alignments for 1-, 2- and 3-byte sized transfers. Figure 8-28 defines the supported alignments for 4-, 8- and 16-byte sized transfers.

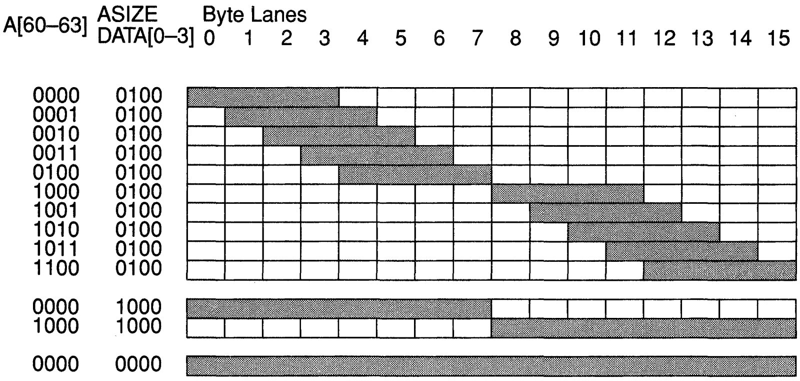
Table 8-26. Supported Data Sizes and Alignments

Size (Bytes)	Name (if any)	Supported		Definition
		Ordinary Segment T=0	Direct Store Segment T=1 (PIO)	
1	Byte	Yes		All byte alignments are supported.
2	Half-Word			All alignments that do not cross a double-word boundary are supported.
3	3-Byte			
4	Word			
8	Double-Word			Only double-word alignments are supported.
16	Quad-Word	Yes	No	Only quad-word aligned quad-word is supported.
5-7,9-15		No		Not Supported

Data alignments not specified by these tables are undefined.

	ASize	Byte Lanes															
A[60–63]	Data	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	[0–3]																
0000	0001	▨															
0001	0001		▨														
0010	0001			▨													
0011	0001				▨												
0100	0001					▨											
0101	0001						▨										
0110	0001							▨									
0111	0001								▨								
1000	0001									▨							
1001	0001										▨						
1010	0001											▨					
1011	0001												▨				
1100	0001													▨			
1101	0001														▨		
1110	0001															▨	
1111	0001																▨
0000	0010	▨	▨														
0001	0010		▨	▨													
0010	0010			▨	▨												
0011	0010				▨	▨											
0100	0010					▨	▨										
0101	0010						▨	▨									
0110	0010							▨	▨								
1000	0010									▨	▨						
1001	0010										▨	▨					
1010	0010											▨	▨				
1011	0010												▨	▨			
1100	0010													▨	▨		
1101	0010														▨	▨	
1110	0010															▨	▨
0000	0011	▨	▨	▨													
0001	0011		▨	▨	▨												
0010	0011			▨	▨	▨											
0011	0011				▨	▨	▨										
0100	0011					▨	▨	▨									
0101	0011						▨	▨	▨								
1000	0011									▨	▨	▨					
1001	0011										▨	▨	▨				
1010	0011											▨	▨	▨			
1011	0011												▨	▨	▨		
1100	0011													▨	▨	▨	
1101	0011														▨	▨	▨

Figure 8-27. Data Alignment for 1-, 2- and 3-byte Sized Transfers



A[60–63]	ASIZE DATA[0–3]	Byte Lanes 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0100	▨	▨	▨	▨												
0001	0100		▨	▨	▨	▨											
0010	0100			▨	▨	▨	▨										
0011	0100				▨	▨	▨	▨									
0100	0100					▨	▨	▨	▨								
1000	0100									▨	▨	▨	▨				
1001	0100										▨	▨	▨	▨			
1010	0100											▨	▨	▨	▨		
1011	0100												▨	▨	▨	▨	
1100	0100													▨	▨	▨	▨
0000	1000	▨	▨	▨	▨	▨	▨	▨	▨								
1000	1000									▨	▨	▨	▨	▨	▨	▨	▨
0000	0000	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨

Figure 8-28. Data Alignment for 4,- 8- and 16-byte Sized Transfers

8.9 Address Bus

The following subsections discuss the address bus (A[0–63]).

8.9.1 Address Formats

The format of A[0–63] varies according to the type of bus operation, with the exception of the ATAG[0–7], which is always aligned as A[8–15]. The following supported address formats are detailed in Figure 8-29:

- Memory Request
- PIO Request
- PIO Reply
- Tag Only
- External Control Request (ExtCon)

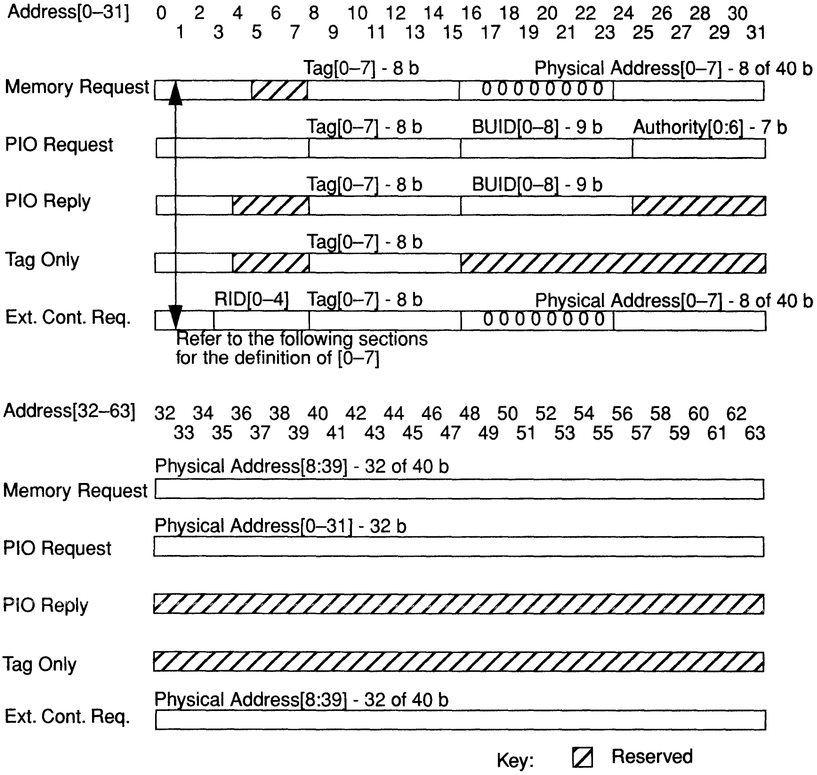

Address[0–31] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Memory Request
Tag[0–7] - 8 b
Physical Address[0–7] - 8 of 40 b
0 0 0 0 0 0 0 0
PIO Request
Tag[0–7] - 8 b
BUID[0–8] - 9 b
Authority[0:6] - 7 b
PIO Reply
Tag[0–7] - 8 b
BUID[0–8] - 9 b
Tag Only
Tag[0–7] - 8 b
Ext. Cont. Req.
RID[0–4]
Tag[0–7] - 8 b
Physical Address[0–7] - 8 of 40 b
0 0 0 0 0 0 0 0
Refer to the following sections for the definition of [0–7]
Address[32–63] 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
Memory Request
Physical Address[8:39] - 32 of 40 b
PIO Request
Physical Address[0–31] - 32 b
PIO Reply
Tag Only
Ext. Cont. Req.
Physical Address[8:39] - 32 of 40 b
Key: ▨ Reserved

Figure 8-29. The Format of A[0–63]

8.9.2 The Memory Request Address Format

The memory request address format is for normal segment accesses to the memory address space and is defined as shown in Table 8-27:

Table 8-27. Memory Request Address Format

A[0–63]	Width Bits	Definition
0–7	5	Refer to Section 8.6, “Address Commands Definition.”
8–15	8	Tag[0–7] Defined in Section 8.10, “Bus Tags.”
16–23	8	Output - B’00000000’ Input - B’XXXXXXXX’
24–63	40	Physical Address[0–39]

8.9.3 The PIO Request Address Format

The PIO request address format is for accesses to the I/O or direct address space and is defined in Table 8-28.

Table 8-28. PIO Request Address Format

A[0–63]	Width Bits	Addressing Mechanism			Definition
		STE	64-bit Segment Register	32-bit Segment Register	
[0]	1	Refer to Section 8.9.3, "The PIO Request Address Format"			Key (K)
[1–3]	3	Refer to Section 8.6, "Address Commands Definition"			ATYPE[1–3]
[4–7]	4	STE double-word 1 [34–36, 44]	SR[46–48, 56]	SR[14–16, 24]	Not part of this document.
[8–15]	8	Refer to Section 8.10, "Bus Tags"			Tag[0–7]
[16–24]	9	STE double-word 0 [60–61] || STE double-word 1 [25–31]	SR[35–43]	SR[3–11]	BUID[0–8]
[25–31]	7	STE double-word 1 [37–43]	SR[49–55]	SR[17–23]	Authority[0–6]
[32–35]	4	STE double-word 1 [48–51]	SR[60–63]	SR[28–31]	Physical Address[0–3]
[36–63]	28	EA[36–63]			Physical Address[4–31]

The K signal is derived from the K bits in the STE and the MSR bit specifying which K-bit to use according to the equation—Key = $(MSR[PR] \& K_P) \mid (\sim MSR[PR] \& K_S)$

8.9.4 The PIO Reply Address Format

The PIO Reply address format is defined as shown in Table 8-29:

Table 8-29. PIO Reply Address Format

A[0–63]	Width Bits	Definition
[0]	1	PIO Error (E-Bit) 1 - Error 0 - No Error
[1–3]	3	Refer to Section 8.6, "Address Commands Definition"
[4]	1	Type of Reply: 1 = load reply 0 = store reply
[5–7]	3	Reserved
[8–15]	8	Tag[0–7], Defined in Section 8.10, "Bus Tags."
[16–24]	9	BUID[0–8]
[25–31]	7	Reserved
[32–63]	32	Reserved

8.9.4.1 Error (E)

The E bit indicates that an error occurred during this PIO operation.

8.9.5 The Tag-Only Address Format

The Tag-Only address format is defined in Table 8-30. The tag-only address format consists of the master or sender tag and attribute bits. The bus operations that use the tag-only address format are EIEIO, SYNC, TLBSYNC and ReRun.

Table 8-30. Tag-Only Address Format

A[0–63]	width bits	Definition
[0–7]	4	Refer to Section 8.6, "Address Commands Definition"
[8–15]	8	Tag[0–7] Defined in Section 8.10, "Bus Tags."
[16–63]	48	Reserved

8.9.6 The External Control Request Address Format (ExtCon)

The external control request address format is defined in Table 8-31:

Table 8-31. External Control Request Address Format

A[0–63]	Width Bits	Definition
[0–2]	3	Refer to Section 8.6, "Address Commands Definition"
[3–7]	5	RID[0–4] (from SPR EAR[27–31]. Refer to Section 2.1.1, "Register Set.")
[8–15]	8	Tag[0–7] Defined in Section 8.10, "Bus Tags."
[16–23]	8	Output - B'00000000' Input - B'XXXXXXXX'
[24–63]	40	Physical Address[0–39]

8

8.10 Bus Tags

Explicit bus tagging has been added to the address and data buses so that data bus transfers do not have to occur in the same order as operations on the address bus. This capability can increase bus bandwidth for any system in which access latencies can vary.

Address bus tagging is found in Section 8.9, "Address Bus," and data bus tagging is found in Section 8.5.4, "Data Bus Tag."

8.10.1 Bus Tag Usage

Bus tags are used for the following purposes.

- A memory identifies the data for a write by comparing the address bus tag and the data bus tag.
- A memory identifies the data for a Read intervention by comparing the address bus tag and the data bus tag.
- The master identifies the data for a read by comparing the address bus tag and the data bus tag.
- The bus device identifies address-only operations that match the bus tag of a pending operation (for example, PIO reply).

Table 8-32 defines the usage of the address and data bus tags. Out means that the tag is produced, In means that the tag is consumed, snoop means that the tag is snooped and N/A means that the bus tag does not have significance for this case.

Table 8-32. Bus Tag Usage

Bus Operation	Tag	Master	Memory	Snooper	PIO Slave
All Reads	A-Tag	Out	In	In	In
	D-Tag	Snoop	Out/In[1]	Out[2]	N/A
All Writes	A-Tag	Out	In	In	In
	D-Tag	Out	Snoop	N/A	Snoop
PIO Reply	A-Tag	Snoop	N/A	N/A	Out
	D-Tag	N/A	N/A	N/A	N/A

Notes:

1. Optional - Only for intervention
2. Optional - Only for intervention

8.10.2 Bus Tag Format

The bus tag should be thought of as the address for a bus device. Just like memory is addressed by the memory address, bus masters are identified or addressed by the bus tags.

At boot time, each bus device is assigned an aligned power of two sized block of bus tag space. This allows for bus tag values to be assigned as needed, instead of allocating a maximum to all bus devices. Not all devices will need the same number of bus tags.

Each bus device that generates bus tags has the notion of a bus tag base address and a bus tag block size. Bus tag blocks can be any size as long as they are in the 256 bus tag space and are unique.

Masters will use the bus tag as a bus device ID and a transfer ID. The master accepts its entire bus tag block and then interprets the transfer ID.

Memory and Snoopers only think of the bus tag as a value, which is compared and presented back to the master.

The 620 bus tag format is defined in Table 8-33.

Table 8-33. Bus Tag[0–7] Format

Bus Tag	bits	Definition
[0–4]	5	BUSPID[0–4]
[5–7]	3	Transaction ID[0–2] (8 bus tags)

- BUSPID[0–4]—The 5-bit bus tag base address is defined to be BUSPID[0–4], which represents a unique processor ID value. BUSPID is software readable by the PIR SPR. See Section 2.1.2.2, "Processor Identification Register (PIR)." BUSPID is assigned by the hardware configuration mechanism; see Section 8.23, "Hardware Configuration Mechanism (HCM)."
- Transaction ID[0–2]—The transaction ID uniquely identifies a bus operation that belongs to the 620, as specified by the BUSPID. This 3-bit field allows a maximum of 8 simultaneous bus operations to be in progress from any one bus master. The 620 can support up to 5 simultaneous pending bus operations, two reads and three writes.

8.10.3 Bus Tag Allocation/Deallocation

Defining allocation and deallocation of tags in a precise manner prevents bus tag aliasing, where the same tag value is associated with two different bus operations simultaneously. The objective is for a single tag at any point in time to refer to no more than one bus operation.

8.10.3.1 Bus Tag Allocation

When a bus operation is allocated, it is associated with that bus operation and can not be used by a bus device to refer to any other bus operation until the tag is deallocated.

8.10.3.2 Bus Tag Deallocation

When a bus operation is deallocated, it is deassociated with that bus operation and can no longer be used by a bus device to refer to that bus operation. The deallocated bus tag is then available for use by a bus device to refer to another bus operation. Note that for a given bus operation, a bus tag may be allocated and deallocated at different times. Refer to the following documentation.

Bus tag deallocation depends on ASTATIN and ARESPIN. Refer to Section 8.4.3, "Address Status and Address Response Signals."

8.10.4 Memory Read—Bus Tag Allocation/Deallocation

This section describes bus tag allocation and deallocation for memory read bus operations. The memory read bus operations are Read and RWITM. See Figure 8-30. For this discussion, the master is the bus device that initiated the memory read operation. The memory is the device that sources the data in response to the memory read operation, which could be the memory subsystem or I/O. Cases associated with each of the ASTATIN or ARESPIN codes are discussed below.

8.10.4.1 ASTATIN NoAck/PosAck, or ARESPIN of Null or Shared

The data bus tag is deallocated by the master on the later of the two deallocation cases in Figure 8-30. The data bus tag is deallocated by the memory on the earlier of the two deallocation cases. The data bus tag is deallocated by the snooper/L3 when the response is known.

This definition prevents any aliasing of the bus tag; the memory system will always deallocate the tag before the master can allocate the same tag for another bus operation.

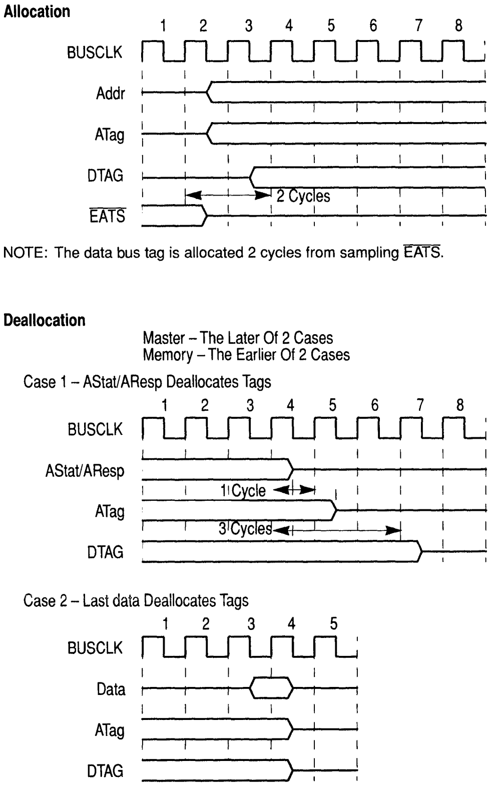
8.10.4.2 ARESPIN of Modified

This is the intervention case. Tag allocation/deallocation is the same as above. The memory may return part of the data block before recognizing the Modified ARI (Address Response In) code and deallocating the tag. The intervening cache will then source the intervention data to the requesting master.

8.10.4.3 Retry or ARESPIN Rerun

Master tag deallocation for ASTATIN/ARESPIN Retry code is described in case #1 in Figure 8-30. Depending on the type of operation that receives ARespin Rerun code, master tag deallocation is described by case #1, or by snooping the Rerun Reply bus operation which tag matches, refer to Section 8.13, “The ReRun Mechanism.” The data bus tag is deallocated by the memory on the earlier of the two deallocation cases. The data bus tag is deallocated by the snooper/L3 when the response is known.

Note: The master can begin rearbitration for the address bus as early as 1 cycle after sampling the ASTATIN/ARESPIN Retry (or Rerun, under certain conditions). If the master is parked onto the address bus, the same address tag can be re-allocated as early as cycle 5 in case #1. Systems which deallocate the data tag as late as three cycles from the ASTATIN/ARESPIN sample point must handle the case of the master re-allocating the same address tag before the system has deallocated the data bus tag.



NOTE: The data bus tag is allocated 2 cycles from sampling $\overline{\text{EATS}}$.

The address bus tag is allocated when the address is sourced by the master at the beginning of cycle 2.

The data bus tag is allocated 2 cycles from sampling EATS. See Section 8.5.7, "Minimum Latency to Sample Read Data."

Case #1 applies to AStat if AStat completes the bus operation or to AResp if AResp completes the bus operation.

The address bus tag is deallocated 1 cycle from sampling ASTATIN or ARESPIN.

The data bus tag is deallocated 3 cycles from sampling ASTATIN or ARESPIN, 2 cycles from the address bus tag deallocation. This latency allows a cross-bar configuration to pass the response from the address controller to each data slice.

The address and data bus tags are deallocated when the last data is sampled by the data consumer.

Note that the last read data for intervention may be the last data sourced by the intervening cache.

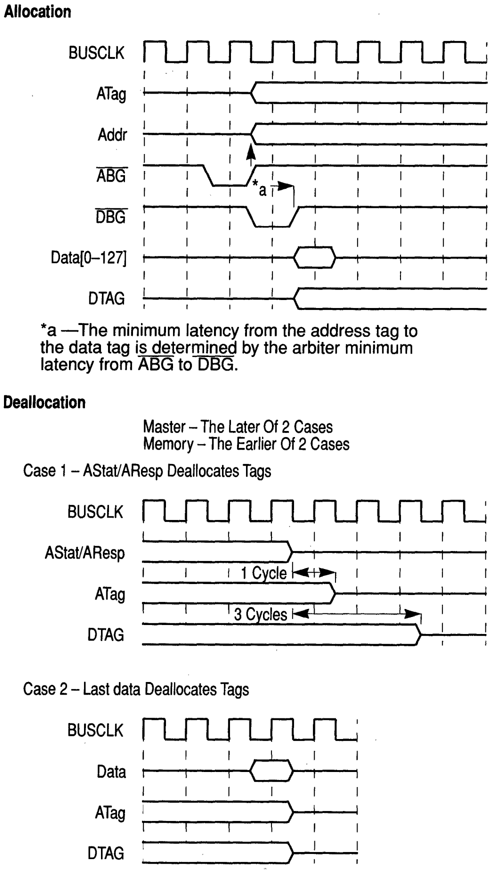
Figure 8-30. Memory Read Bus Tag Allocation/Deallocation

8.10.5 Memory Write—Bus Tag Allocation/Deallocation

This section describes bus tag allocation and deallocation for memory write bus operations. The memory write bus operations are Write with Flush, Write with Kill and Write with Clean. See Figure 8-31. The address tag is not snooped for memory writes.

The data bus tag is deallocated by the Master or Memory-Atomic (Write with Flush Atomic— see Section 8.12.6, "Write-With-Flush") on the later of the two deallocation cases.

The data bus tag is deallocated by the Memory (Not-Atomic) on the earlier of the two deallocation cases. This definition prevents any aliasing of the bus tag; the memory system will always deallocate the tag before the master can allocate the same tag for another bus operation.


Allocation
BUSCLK
ATag
Addr
ABG
DBG
*a
Data[0–127]
DTAG
Deallocation
Master – The Later Of 2 Cases
Memory – The Earlier Of 2 Cases
Case 1 – AStat/AResp Deallocates Tags
BUSCLK
AStat/AResp
1 Cycle
ATag
3 Cycles
DTAG
Case 2 – Last data Deallocates Tags
BUSCLK
Data
ATag
DTAG

*a —The minimum latency from the address tag to the data tag is determined by the arbiter minimum latency from ABG to DBG.

The address bus tag is allocated when the address bus grant is sampled, which is also when the address is sourced.

The data bus tag is allocated when the data bus grant is sampled, which **may** be when the first data is sourced.

Case #1 applies to AStat if AStat completes the bus operation or to AResp if AResp completes the bus operation.

The address bus tag is deallocated 1 cycle from sampling ASTATIN or ARESPIN.

The data bus tag is deallocated 3 cycles from sampling the response, 2 cycles from the address bus tag deallocation.

If the write operation is not retried then the address and data bus tags are deallocated when the last data is sampled by the data consumer.

Figure 8-31. Memory Write Bus Tag Allocation/Deallocation

8.10.6 PIO—Bus Tag Deallocation

The same bus tag is used for all PIO bus operations caused by a single load/store to the direct store segment. The bus tag is deallocated for the last PIO bus operation like a T=0 bus operation. The bus tag is deallocated for a PIO Reply in the cycle after the address is sampled on the bus. Note that a PIO Reply with the error bit set does not deallocate the data bus tag. The bus master still requires the PIO device to satisfy the data bus transfer requirements. The bus tag is deallocated for a PIO Load Last like a read bus operation. The bus tag is deallocated for a PIO Store Last like a write bus operation.

8.10.7 Address-Only Bus Operations—Bus Tag Deallocation

This section describes bus tag deallocation for address only bus operations. The address-only bus operations are LARX-Reserve, IKill, DKill, DClaim, Flush, Clean, TLBIE, TLBSYNC, SYNC and EIEIO.

The address bus tag will be allocated when the address is sourced, in the same manner that the address bus tag is allocated for memory reads and writes. The address bus tag will be deallocated for address-only bus operations one cycle from sampling the response, in the same manner that the address bus tag is deallocated for memory reads and writes. The data bus tag is not allocated for address-only bus operations.

8.10.8 Snoopers - Bus Tag Deallocation

Unless otherwise stated, a snooper may deallocate the bus tag for a bus operation when the snooper has snooped the cache and determined that a ReRun for TLBSYNC/SYNC or a data-only bus operation that uses the bus tag will not be needed.

8.11 Parity Protection

This section discusses parity protection.

- Odd Parity—Bus parity is defined as odd parity. All bits over which parity is computed, including the parity bit itself, that are a binary 1 add up to an odd number.
- Parity Generation—Parity is always generated for all parity protected signals during a transfer, independent of transfer content beign valid (that is, even if only 1 byte of data is valid parity is still computed over a quad-word).
- Memory and PIO Parity Errors—Parity error detection and handling is the same for PIO and memory operations.
- Machine-Check exceptions are Asynchronous—A parity error that causes a machine-check exception will occur asynchronously with respect to the operation that caused the exception. If a PIO Load Last causes a DSI exception and a machine check exception is pending from a PIO load immediate, there is no guarantee between the order of the machine-check and the DSI exceptions.
- Address Bus Parity Checking and Handling (ASTATOUT)
  - — Every address bus device parity checks the address bus for a bus operation, except for the master for that address bus operation. If a parity error is detected and HID0(EBA) is asserted then the bus device will assert the ASTATOUT AParErr code, otherwise ASTATOUT will be the appropriate code for a non-parity error condition. The definition of HID0(EBA) can be found in Section 2.1.2.3, “Hardware Implementation-Dependent Register 0 (HID0).”

  - ASTATIN and Master—If a master bus device receives the ASTATIN AParErr code then BUSCSR(BUSPARERR[0]) will be set and a machine-check exception will be issued. Note that EBA does not affect the actions taken when ASTATIN is AParErr. The master operation will be aborted and not reissued to the bus. The data for a store is discarded. The data for a read is indeterminate.
  - ASTATIN and Non-Master—A non-master device will never issue a machine-check exception or set its own BUSCSR(BUSPARERR[0]) when an address bus parity error is detected. A non-master bus device that receives the ASTATIN AParErr code will not issue a machine-check exception, will not set BUSCSR(BUSPARERR[0]), and will treat ASTATIN AParErr just like ASTATIN Retry.

- Data Bus Parity Checking and Handling—Every data bus device, including the master, parity checks the data bus for every data bus transfer. The detection of a data parity error will not prevent the bad data from being forwarded to the caches or the registers.
  - Data tag errors—Every data bus consumer, that expects to receive data, parity checks the data tag (DPCntl) for every BUSCLK that $\overline{\text{DVAL}}$ is asserted. If a parity error is detected then BUSCSR(BUSPARERR[1]) is set. If HID0[EBD] is asserted the a machine-check exception will be generated. The definition of HID0[EBD] can be found in Section 2.1.2.3, "Hardware Implementation-Dependent Register 0 (HID0)."
  - Data errors—If a parity error is detected with the data bus (DPHigh[0–7], DPLow[0–7] and DPDPCntl) by the consumer of the data, then BUSCSR(BUSPARERR[2]) is set. If HID0[EBD] is asserted then a machine-check exception will be generated. If the device is not the data consumer then BUSCSR(BUSPARERR[2]) is not modified and a machine-check is not generated.

Table 8-34. Address Bus Parity Protection

BUSPARERR[0–2]	Definition
0	Address bus parity error.
1	Data bus tag parity error.
2	Data bus data parity error.

Refer to Section 2.1.2.4, “Bus Status and Control Register (BUSCSR),” and Section 4.6.2, “Machine Check Exception (0x00200).”

- Signals Excluded from Parity Protection—There is no parity protection for the following classes of signals:
  - — Clocks (BUSCLK)
  - — Arbitration ($\overline{\text{ABR}}$, $\overline{\text{HPR}}$, $\overline{\text{ABG}}$, $\overline{\text{DBR}}$, $\overline{\text{DBG}}$ and $\overline{\text{EATS}}$), ASTATOUT[0–1] and ARESPOUT[0–2]
  - — ASTATIN[0–1] and ARESPIN[0–2]
  - — $\overline{\text{DVAL}}$[0–1]
  - — $\overline{\text{DERR}}$
  - — Miscellaneous Control (Reserve)
- Address Bus Parity—The address bus is parity protected by 3 bits, called AP[0–2]. Table 8-35 defines the coverage for each parity bit.

Table 8-35. Address Bus Parity Protection

AP[N]	Coverage
0	A[0–31] (includes the address tag)
1	A[32–63]
2	ATYPE[0–4], ASIZEDATA[0–3], $\overline{\text{ASIZEBURST}}$

- Data Bus Parity—The data bus is parity protected by17 bits, called DPHigh[0–7], DPLow[0–7] and DPCntl. Table 8-36 defines the coverage for each parity bit.

Table 8-36. Data Bus Parity Protection

Data Parity	Coverage
DPHigh[0–7]	DH[0–63] bytes 0 to 7
DPLow[0–7]	DL[0–63] bytes 0 to 7
DPCntl	DTAG[0–7], $\overline{\text{DBB}}$ and $\overline{\text{DCACHE}}$

8.12 Bus Operations

The purpose of this section is to describe the function of each bus operation. Refer to Figure 8-32 for the bus topology. Note that local bus and remote bus are also referred to as the ‘upper level’ and ‘lower level’, respectively.

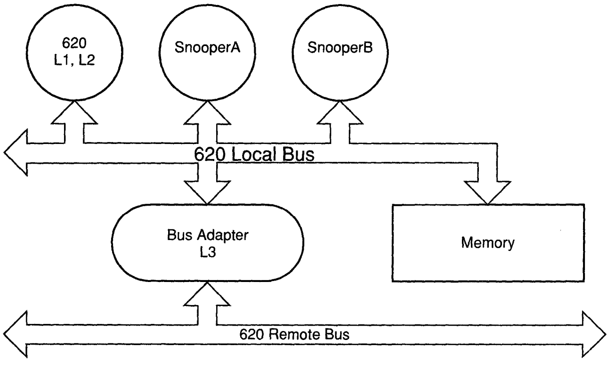


Figure 8-32. The Bus Topology

Related topics:

- Section 8.8, “ASIZEDATA[0–3] and ASIZEBURST Definition”
- Section 8.17.16, “Cache State Transition Definition”
- Section 8.6.1.1, “Write-Through Address Attribute (W-Bit)”
- Section 8.6.1.2, “Cache-Inhibited Address Attribute (I-Bit)”
- Section 8.6.1.3, “Memory Coherent Address Attribute (M-Bit)”
- Section 8.6.2, “Atomic Address Attribute (A-Bit)”
- Section 8.6.3, “Intervention Address Attribute (N-Bit)”
- Section 8.7, “Intervention and Push Definition.”

8.12.1 RWITM (Read-With-Intent-To-Modify)

The Read-With-Intent-To-Modify (RWITM) bus operation is intended to bring a block into a cache marked modified for the purpose of writing. $\overline{\text{ASIZEBURST}}$ is always asserted. In general, the RWITM is caused by a store, **stwcx.** or **dcbtst** instruction; see Section 8.5.7, “Minimum Latency to Sample Read Data.”

The RWITM bus operation is accepted from the upper level and ReRun until completed. If the cache level is marked I, then a RWITM bus operation is passed to the lower level. If the cache level is marked S, then a DClaim bus operation is passed to the lower level. If the cache level is marked E or M, then no bus operation is passed to the lower level. When the response for the bus operation to the lower level, if any, is Null, then the Null response can be passed to the upper level and the cache can be marked M. Data is sourced from the highest valid cache or memory if memory is at the same level as the highest valid cache.

- The N-Bit—The N-bit is sourced from the intervention enable mode bit.
- The A-Bit—If the A-bit is deasserted then the function of RWITM is normal as defined previously. If the A-bit is asserted and the reservation is clear, then the RWITM is retried to the upper level with the assumption that what cleared the reservation at this level will eventually clear the reservation at the upper level.

  If the A-bit is asserted and the reservation is set, then the RWITM operates as previously defined. When the response from the lower level is Null, then the cache is marked M, the reservation is cleared, and the null response is passed to the upper level.

8.12.2 LARX-Reserve

The data for a cacheable LARX will be supplied by the highest cache level that hits. However, all reservations below the highest cache level that hits are notified of the reservation via the LARX-Reserve bus operation. If the L2CLC[1] (L3 Enable) is deasserted, indicating that there is no reservation below the 620, then the LARX-Reserve bus operation is inhibited.

The LARX-Reserve bus operation propagates down all cache levels to the lowest level. Each level may Null response a requester when the LARX-Reserve bus operation has been accepted (for example, when the 620 passes the LARX-Reserve to the L3). If the LARX-Reserve bus operation conflicts with a bus operation coming from below that will invalidate the reservation, such as a write, then the LARX-Reserve may be aborted and the response may be Null. This assumes that the write is going to be propagated to the top level and abort all reservations that match above this level. The LARX-Reserve bus operation is complete when it sets the reservation in the lowest level.

8.12.3 Read

The Read bus operation is caused by a load, load multiple, DCBT, or LARX. Read burst bus operations will always be block sized. Read non-burst bus operations will use all supported non-burst sizes and alignments. See Section 8.8.5, "Supported Non-Burst Data Sizes and Alignments," and Section 8.5.7, "Minimum Latency to Sample Read Data."

- The N-Bit—Burst reads: The N-bit is sourced from the intervention enable mode bit. NonBurst reads: The N-Bit is 0.
- The S-Bit—The 620 as a Read bus operation master will source the S-bit from the PTE I-Bit. The 620 as a Read bus operation snooper will interpret the S-Bit according to the following table and to the table called Section 8.17.18, "Snooper Cache State Transitions Due to Bus Operations."

Table 8-37. The Bus S-Bit for the Snooped Read Operation

$\overline{\text{ASIZEBURST}}$	Bus S-Bit	Snooper Definition
No	1	Normal Read Non-Burst Operation (WI=01).
No	0	Normal Read Non-Burst Operation (WI=10).
Yes	1	RWNITC (Read With No Intent To Cache).
Yes	0	Normal Read Burst Operation (WI=00).

- AResp: The Address Response—If I=1 and M=0, then AResp is not valid and is ignored. AResp is valid for all other T=0 memory access modes. Refer to Section 8.4.3.2, “Address Status and Address Response Validation.”

- The A-Bit—The A-bit, when asserted, indicates that the read was caused by a cache-inhibited or cacheable write-back LARX. Refer to Section 8.20, “Atomic Memory Reference Support.” The read will set the reservation for each level that the read accesses. If the read is cacheable write-back and hits on a block marked S, E, or M, then the LARX-Reserve bus operation will be issued to the lower level if the L3 is enabled.

- The G-Bit—The G-bit, when asserted, indicates that the cache-inhibited read is guarded. The G-bit, when deasserted, indicates that the cache-inhibited read is non-guarded. This bit is undefined and assumed unguarded for cacheable reads. A guarded cache-inhibited read may only complete once.

8.12.4 Write-With-Kill

The “Write-With-Kill” bus operation is issued by the 620 as a master due to the following conditions:

- Copybacks: A processor load, store, or DCBZ allocates the addressed block, which in turn causes the deallocation of a modified block with a different block address.
- Flushes: A processor DCBF hits against the addressed block which is marked modified.
- Pushes: A bus snoop RWITM, Write-With-Flush or Flush hits against the addressed block which is marked modified.

Write-With-Kill is always a burst block sized write that tells all snoopers to mark a block that address matches to the I state.
Memory will be updated independent of the response.

The W and M bits are defined in Table 8-38 and are not sourced from the PTE.

Table 8-38. Supported Write-With-Kill Types

Cause/Comment	W-Bit	M-Bit	PowerPC 620 Master Generated	PowerPC 620 Snooper Generated
Copyback (CB): Caused by a processor load, store, or DCBZ.	0	0	Y	N
Reserved.	0	1	N	N
Flush (F): Caused by a processor DCBF.	1	0	Y	N
Push (P): Caused by a bus snoop RWITM, Write-With-Flush, or Flush.			N	Y
IO Write (IO)	1	1	N	N

8.12.5 Write-With-Clean

The "Write-with-Clean" bus operation is issued by the 620 as a master due to the following conditions:

- A processor DCBST hits against the addressed block which is marked modified.
- A bus snoop Read (Burst and NonBurst) or Clean hits against the addressed block which is marked modified.

The block is written through all levels to memory and the final state is S or E. Write-with-Clean is always burst block sized.

8.12.6 Write-With-Flush

The Write-with-Flush bus operation is described as follows:

- As a Master—The 620 will issue the Write-With-Flush bus operation due to T=0 cache-inhibited or write-through stores, T=0 cache-inhibited STCX and MMU RC bit updates. The Write-With-Flush bus operation will always be a non-burst write to memory and will use all supported non-burst sizes and alignments. See Section 8.8.5, "Supported Non-Burst Data Sizes and Alignments." Memory may commit the store independent of the response (AResp).
- As A Snooper—The 620 will snoop into the L1 and L2, if enabled, and will cause a flush of the addressed block if the addressed block is detected. If the block is detected to be modified then the snooper will retry the Write-with-Flush bus operation and then the addressed block is pushed back to memory with a Write-with-Kill bus operation. The Write-With-Flush bus operation address is only significant as a block address and the size, alignment and burst/non-burst attributes are insignificant.
- Address Response—Refer to Section 8.4.3.2, "Address Status and Address Response Validation," and Section 8.4.9, "Address Response In/Out."

- The A-Bit—The atomic bit indicates whether this bus operation was caused by a cache-inhibited STCX. The atomic bit modifies the functionality of Write-With-Flush as follows:

  If the bus operation is atomic and the reservation is set, then the device may give the Null response when the bus operation has been accepted, passed to the next lower level, and the response is Null. This process is followed by all levels down to the lowest level, each level rerunning or retrying the upper level until the lowest level gets the Null response. The Null response then propagates up each level, as long as the reservation is still set. The device may update the cache when the response is Null and the reservation is set. The reservation is not cleared when the Write-With-Flush bus operation is retried.

  If the bus operation is atomic and the reservation is clear, then the device must retry the upper level with the assumption that whatever cleared the reservation is going to clear the reservation at all above levels (for example, whatever cleared the L3 reservation is going to clear the 620 reservation). The device may not update the cache when the response is Null and the reservation is clear.

  Since the memory has no knowledge of whether the reservation is set, it will conditionally execute the Write-With-Flush (both Atomic and Non-Atomic) based on the response. The memory will complete the write when the response is null. The memory will wait if the response is ReRun. The memory will abort if the response is retry.

8.12.7 IKill

The intent for IKill is to invalidate a block in all instruction-only caches. Instruction-data and data-only caches will be invalidated by other cache control operations. The address is a physical address.

Refer to Section 2.3.4.6, "Processor Control Instructions," and Section 2.3.5.2, "Memory Synchronization Instructions."

8

8.12.8 DKill

The intent of DKill is to invalidate all copies of this block in all instruction-data and data-only caches, including the 620 that issues the DKill.

8.12.9 DClaim

The DClaim bus operation can be caused by any of the following conditions:

- A write-back store to a block marked shared (S).
- A write-back DCBZ to a block that is either marked shared (S) or invalid (I).

The intent of DClaim is to invalidate all copies of this block in other caches and to mark my copy of the addressed block M in all levels. This bus operation differs from DKill in that it does not kill the master's copy in lower levels.

DClaim is accepted by the upper level and ReRun until the DClaim is passed to the lower level and the response for the lower level is a Null, at which time the cache is marked M and the Null response is passed to the upper level. This has the effect of waiting until the DClaim has propagated to the lowest level and succeeded before marking the block M.

If the A-bit is deasserted then the function of DClaim is as defined previously. If the A-bit is asserted and the reservation is clear, then the DClaim is retried to the upper level with the assumption that what cleared the reservation at this level will eventually clear the reservation at the upper level.

If the A-bit is asserted and the reservation is set, then the DClaim is passed to the lower level. When the response from the lower level is Null, then the cache is marked M, the reservation is cleared and the null response is passed to the upper level.

8.12.10 Flush

The Flush bus operation is issued by the 620 due to the DCBF instruction. Flush causes any cache that has a copy of the block marked modified to write the block back to memory. All caches mark the block invalid.

8.12.11 Clean

Clean causes any cache that has a copy of the block marked modified to write the block back to memory; the block is marked exclusive in the lowest cache level (L3, for example), otherwise, the block is marked shared.

8.12.12 The tlbie, tlbsync, sync and eieio Instructions

The definitions of **tlbie** and **tlbsync** is located in Section 2.3.6.3.3, "Translation Lookaside Buffer Management Instructions." The 40-bit A[24–63] is defined to be the EA[24–63] of the **tlbie**.

The definition of **sync** is located in Section 2.3.4.7, "Memory Synchronization Instructions."

The definition of **eieio** is located in Section 2.3.5.2, "Memory Synchronization Instructions."

8.12.13 ReRun

Refer to Section 8.13.2, "The ReRun Bus Operation."

8.12.14 Null

The null bus operation is ignored by all bus devices and is intended to enable a bus device to take ownership of the address bus when a bus device does not have an address bus operation to run.

8.12.15 PIO Loads, Stores and Reply

Refer to Section 8.16, "PIO Load and Store Bus Operations," for a description of PIO loads and stores.

8.12.16 External Control In and Out

The architectural definition of the external control instructions is located in Section 2.3.5.4, "Optional External Control Instructions."

The address format for the External Control bus operations is located at Section 8.9.6, "The External Control Request Address Format (ExtCon)."

The External Control Out bus operation is a word-sized non-burst address-data bus operation that is treated as a write by the external device addressed by the RID[0–4].

The External Control In bus operation is a word-sized non-burst address-only bus operation that is treated as a read by the external device addressed by the RID[0–4].

Address positive acknowledge is provided by the external device addressed by the RID[0–4] for both bus operations.

Data on the data bus is aligned according to the word address just like a cache-inhibited load or store.

8.13 The ReRun Mechanism

The ReRun mechanism allows the snoop response for a bus operation to be extended beyond the fixed latency response time of ARESPOUT (BUSTLAR BUSCLKs). A bus device that desires to extend the snoop response latency asserts ARESPOUT ReRun.

A bus adapter or snooper will assert ARESPOUT ReRun in order to extend the response of a bus operation. If the master receives ARESPIN ReRun the master will reissue the same bus operation with the same tag and assert the R-Bit. The assertion of the R-Bit indicates to the snooper and bus adapter bus devices that this is not the first occurrence of this operation. The ReRun bus operation is used by a bus adapter for non-SYNC/TLBSYNC bus operations to tell the master to run the bus operation again.

For more information, refer to:

- Section 8.13.1, "AResp ReRun"
- Section 8.13.3, "The R-Bit"
- Section 8.13.2, "The ReRun Bus Operation"

The 620 supports ReRun as a master for all AResp-enabled bus operations. The 620 snooper will ARESPOUT ReRun the SYNC and TLBSYNC bus operations. The 620 will not ARESPOUT ReRun non-SYNC/TLBSYNC bus operations.

For more information, refer to:

- Section 8.13.4, “ReRun and the Master”
- Section 8.13.5, “ReRun and Memory”
- Section 8.13.6, “SYNC/TLBSYNC: ReRun and the Snooper/Bus-Adapter”
- Section 8.13.7, “Non-SYNC/TLBSYNC: ReRun and the Snooper/Bus-Adapter”

All bus operation types that can be rerun are divided into two groups that function differently with respect to the rerun bus operation. SYNC/TLBSYNC bus operations do not use the ReRun bus operation. Non-SYNC/TLBSYNC bus operations use the ReRun bus operation. The SYNC/TLBSYNC group consists of the SYNC and TLBSYNC bus operations. The Non-SYNC/TLBSYNC group consists of all AResp-enabled bus operations other than SYNC and TLBSYNC.

8.13.1 AResp ReRun

The AResp ReRun code indicates that this bus operation should be “re-run” by the master using the same ATYPE and bus tag. See Section 8.4.9, “Address Response In/Out.”

8.13.1.1 ARESPOUT ReRun

Bus adapters are allowed to ARESPOUT ReRun all AResp-enabled bus operations. Snoopers and memories are allowed to ARESPOUT ReRun only the SYNC and TLBSYNC bus operations. There is no limit to the number of times that a snooper or bus adapter can assert ARESPOUT ReRun. The master for a bus operation will never assert ARESPOUT ReRun.

8.13.1.2 ARESPIN ReRun

See the following sections for how ARESPIN ReRun is handled by the master, memory, snooper and bus adapter.

8.13.2 The ReRun Bus Operation

The following sections provide a detailed explanation of the ReRun bus operation.

8.13.2.1 The Function of the ReRun Bus Operation

The ReRun bus operation is an address-only bus operation issued from a bus adapter to the master, and possibly other bus adapters, to tell the master to reissue the non-SYNC/TLBSYNC bus operation that was AResp ReRun. A master that receives ARESPIN ReRun for a non-SYNC/TLBSYNC bus operation will wait to see the ReRun bus operation that tag matches the non-SYNC/TLBSYNC bus operation before rearbitrating and reissuing the bus operation that was ARESPIN ReRun. The ReRun bus operation is not used for the ReRun of SYNC/TLBSYNC bus operations. The tag for the ReRun bus operation is the tag for the operation that was AResp ReRun. The ReRun bus operation has no address and therefore is tag-only. See Section Table 8-16, “. ATYPE[0–4] Definition.” See Section Table 8-17, “. WIM-Bit Definition.” Other bus adapters, if there are any, should also take notice of the ReRun bus operation. See Section 8.13.7, "Non-SYNC/TLBSYNC: ReRun and the Snooper/Bus-Adapter."

8.13.2.2 When the Bus Adapter Issues the ReRun Bus Operation

The earliest time that a bus adapter may issue the ReRun bus operation is immediately following the address bus tenure of the operation that will be ReRun, which is prior to the master receiving ARESPIN ReRun.

The latest time that the bus adapter may issue the ReRun bus operation is BUSTLAR BUSCLKs from snooping another ReRun bus operation, from another bus adapter, that tag matches the operation being rerun. Note that this case does not exist in a single bus adapter system.

8.13.3 The R-Bit

The R-Bit is located in the most significant byte of the address. See Section Table 8-16, ". ATYPE[0–4] Definition." The R-Bit indicates whether this operation has already been issued and ARESPIN ReRun by at least 1 bus device. R=0 for the first time that a bus operation is issued. R=1 when the bus operation is reissued due to ARESPIN ReRun. If R=1 then the master must use the same bus operation type and tag as was used when the bus operation was run R=0. The R-Bit is defined for all AResp-enabled bus operations.

8.13.3.1 The R-Bit with Respect to AStat/AResp ReRun/Retry

The following table defines the value of the R-bit based on AStat Retry and AResp ReRun and Retry (for example, if an R=1 operation is AStat Retried, then the operation will be reissued R=1). Note that the master does not have to reissue bus operations that are AResp Retried (that is, as a result of the AResp Retry, the master may change, or even kill, the bus operation).

Table 8-39. The R-Bit with respect to AStat/AResp ReRun/Retry

R-Bit	ASTATIN/ARESPIN	Reissued R-Bit
0	ASTATIN Retry	0
	ARESPIN Retry	0
	ARESPIN ReRun	1
1	ASTATIN Retry	1
	ARESPIN Retry	0
	ARESPIN ReRun	1

8.13.3.2 The R-Bit and Address Tag Matching for Snoopers and Bus Adapters

Snoopers will ignore the R-Bit and will not tag match for the non-SYNC/TLBSYNC bus operations. The following table defines how bus adapters handle non-SYNC/TLBSYNC bus operations and how snoopers and bus adapters handle the SYNC and TLBSYNC bus operations.

Table 8-40. The R-Bit and Address Tag Matching Handling for Snoopers and Bus Adapters

R-Bit	Address Tag Match	Definition of: - Bus Adapters and Snoopers for SYNC/TLBSYNC - Bus Adapters for Non-SYNC/TLBSYNC
0	No	The snooper/adapter accepts the bus operation as a new bus operation to process.
0	Yes	ILLEGAL CASE: The snooper/adapter should never be processing a bus operation and snoop an R=0 bus operation that tag matches.
1	No	The snooper/adapter has already finished this bus operation and should ignore this and all future occurrences of this bus operation.
1	Yes	The snooper/adapter should use the response of this bus operation to indicate ReRun or $\overline{\text{ReRun}}$.

8.13.4 ReRun and the Master

The following sections provide information on ReRun and the master.

- ARESPOUT ReRun—The master will never assert ARESPOUT ReRun.
- ARESPIN ReRun—A master that receives ARESPIN ReRun will behave differently based on whether the bus operation is a SYNC/TLBSYNC, or a bus operation other than SYNC/TLBSYNC.
- SYNC/TLBSYNC: ARESPIN ReRun and the ReRun Bus Operation—The master will not wait to rearbitrate internally. The master will not snoop for the ReRun bus operation that tag matches the bus operation that received ARESPIN ReRun.
- Non-SYNC/TLBSYNC: ARESPIN ReRun and the ReRun Bus Operation—The master will wait to rearbitrate internally until a ReRun bus operation is snooped that tag matches the bus operation that received ARESPIN ReRun. Note that the earliest time that the ReRun bus operation may be snooped by the master is immediately following the address tenure for the bus operation that will receive ARESPIN ReRun. With respect to read and write data the master will treat ARESPIN ReRun like ARESPIN Retry (that is, any data received will be discarded and any data sent will be resent when the operation is run again).
- The R-Bit—Refer to Section 8.13.3, "The R-Bit," for when the R-Bit is asserted and deasserted. If R=1 then the master must use the same bus operation type and tag as was used when the bus operation was run R=0.

8.13.5 ReRun and Memory

See Section 8.14, "The Definition of Memory."

- ARESPOUT ReRun—The memory will never assert ARESPOUT ReRun.
- ARESPIN ReRun—The memory treats ARESPIN ReRun like ARESPIN Retry.
- The ReRun Bus Operation—The memory ignores the ReRun bus operation issued by the bus adapter(s).
- The R-Bit—The memory ignores the bus R-Bit and does not have to tag compare R=1 bus operations with any memory operations already in progress.

8.13.6 SYNC/TLBSYNC: ReRun and the Snooper/Bus-Adapter

This section describes the ReRun mechanism for the SYNC and TLBSYNC bus operations and the snooper and bus adapter bus devices.

- ARESPOUT ReRun—Snooper and bus adapter bus devices may use the ReRun mechanism for extending the response of SYNC and TLBSYNC bus operations.
- ARESPIN ReRun—If a snooper or bus adapter receives ARESPIN ReRun for a SYNC/TLBSYNC bus operation then the snooper or bus adapter will drop the operation if ARESPOUT was not ReRun. The snooper or bus adapter will hold onto the operation if ARESPOUT was ReRun.
- The ReRun Bus Operation—The snooper and bus adapter bus devices will not issue the ReRun bus operation. (The master will not wait to internally rearbitrate the SYNC or TLBSYNC bus operation.)
- The R-Bit—See Section 8.13.3, "The R-Bit," for how the snooper and bus adapter handle the R-Bit for SYNC/TLBSYNC bus operations.

8.13.7 Non-SYNC/TLBSYNC: ReRun and the Snooper/Bus-Adapter

This section describes the ReRun mechanism for the non-SYNC/TLBSYNC bus operations and the snooper and bus adapter bus devices.

- ARESPOUT ReRun—Only bus adapter bus devices may use the ReRun mechanism for extending the response of non-SYNC/TLBSYNC bus operations. Snooper type bus devices must ARESPOUT Retry if the snoop response is not available by ARESPOUT.
- ARESPIN ReRun—If a snooper receives ARESPIN ReRun for a non-SYNC/TLBSYNC bus operation the snooper will take no action except to mark all cache blocks that hit exclusive (E State) to the shared state (S State). See Section 8.17.18, "Snooper Cache State Transitions Due to Bus Operations." The snooper will drop the operation as soon as it is determined that either no cache state change to shared is needed or the state change to shared is complete.

  A bus adapter that receives ARESPIN ReRun and did not assert ARESPOUT ReRun will not participate further for this bus operation.

A bus adapter that receives ARESPIN ReRun and did assert ARESPOUT ReRun will issue the ReRun bus operation when the remote status is obtained.

- The ReRun Bus Operation (Snooper)—The snooper will never issue the ReRun bus operation and will not snoop the ReRun bus operation.
- The ReRun Bus Operation (Bus Adapter)—The bus adapter will issue the ReRun bus operation when the remote status for the bus operation has been received by the bus adapter. The bus adapter **may** issue the ReRun bus operation before receiving the remote status, for the purpose of reducing the latency to complete the R=1 bus operation. The earliest time that the bus adapter may issue the ReRun bus operation is immediately following the address tenure for the bus operation that received ARESPIN ReRun.

  If there is only one bus adapter, then the bus adapter does not have to snoop the ReRun bus operation. If there are multiple bus adapters, then the bus adapter must snoop for the other bus adapters ReRun starting immediately following the address tenure for the bus operation that received ARESPIN ReRun. If a bus adapter sees the ReRun for another bus adapter that tag matches an operation that was ReRun by that bus adapter, then the bus adapter may not issue the ReRun bus operation after BUSTLAR BUSCLKs from snooping the ReRun bus operation. If the remote status has not been received when ARESPOUT occurs for the R=1 bus operation, then ARESPOUT is ReRun.
- The R-Bit—See Section 8.13.3, "The R-Bit," for how the snooper and bus adapter handle the R-bit for non-SYNC/TLBSYNC bus operations.

8.14 The Definition of Memory

A definition of memory is follows:

- Supported Bus Operations—Only the following bus operations are received by the memory. All other bus operations are ignored by the memory.
  - RWITM and Read (all types)
  - Write-With-Kill, Write-With-Clean and Write-With-Flush
  - SYNC and EIEIO (Significant to the memory only if reordering can occur)
  - All positive acknowledge bus operations handled by the memory; see Table 8-13.
- Minimal Requirements
  - Positive Acknowledgment—The memory must provide positive acknowledgment. See Section 8.4.10, "Address Status Acknowledge."
  - $\overline{\text{ABR}}$ and $\overline{\text{ABG}}$—A memory does not have to implement $\overline{\text{ABR}}$ and $\overline{\text{ABG}}$.
  - Parity Error Reporting—AStat AParErr is the address bus parity error reporting mechanism. The bus protocol does not specify a data bus parity error reporting mechanism. It is up to the memory to define how data bus parity error reporting is handled.

  - Receiving ASTATIN and ARESPIN—A memory must receive ASTATIN and ARESPIN in order to detect when a bus operation is aborted. A memory is required to monitor ASTATIN retry and ARESPIN retry. Refer to Section 8.4.17, “Address Status Out and Address Response Out Retry.”
  - Driving ASTATOUT: Flow Control, Positive Acknowledge, Parity—A memory must use ASTATOUT Retry for flow control. If positive acknowledge is enabled, then the memory must generate ASTATOUT PosAck for the memory space of the memory. If the memory checks parity, then the memory must generate ASTATOUT AParErr if an address bus parity error is detected.
  - Driving ARESPOUT—The memory does not have to drive ARESPOUT.
  - Guarded Space—Any accesses to guarded memory cannot occur until ASTATIN and ARESPIN responses indicate that the operation can proceed.
  - Supported Non-Burst and Burst sizes—See Section 8.8.5, “Supported Non-Burst Data Sizes and Alignments.” The memory must support the data sizes and alignments that the 620 can generate.
  - Burst data ordering—Data must always be received and transmitted according to critical quad-word first with incremental wrap-around within an aligned block. Refer to Section 8.5.3, “Data Sequence Ordering for Burst Operations.”
  - Aborting read return data based on a Retry or Modified response—The minimal requirement for a memory is that the memory shall not use a tag for an address bus operation three cycles after sampling an ARESPIN Retry or Modified response for that bus operation. This means that the minimal implementation only needs to place a known unused tag on the data tag bus, or deassert $\overline{\text{DVAL}}$, so that the master will not see the return data. A more sophisticated approach that does not waste data bus bandwidth will abort the read return data and start the next memory operation.
  - The ReRun mechanism—The memory treats ARESPIN ReRun like ARESPIN Retry. The memory can ignore the bus R-Bit (Address[7]) and does not have to tag compare R=1 bus operations with any memory operations already in progress. See Section 8.13.5, “ReRun and Memory.”

- RWITM and Read—Read and RWITM, without intervention, are treated like a basic read, because “with intent to modify” only has meaning for a cache. See Section 8.5.7, “Minimum Latency to Sample Read Data.”

  A memory may implement intervention. Refer to Section 8.7, “Intervention and Push Definition,” and Section 8.6.3, “Intervention Address Attribute (N-Bit).”

- Write-With-Kill, Write-With-Clean—Write-with-Kill and Write-with-Clean are both treated like a basic write, because kill and clean only have meaning for a cache.

  If a write is aborted (for example, ASTATIN Retry, AParErr, ARESPIN Retry, ReRun) then the data tag will be deallocated according to Section 8.10.5, “Memory Write—Bus Tag Allocation/Deallocation.” Write data that has been received prior to the data tag deallocation for an aborted write may, but does not have to be committed to the memory array.

- Write-With-Flush (Atomic and Non-Atomic)—A memory must treat any Write-With-Flush as a conditional operation that waits for ReRun, aborts for Retry, and commits for Null. See Section 8.12.6, "Write-With-Flush."
- SYNC and EIEIO—These operations only act as barriers. If the memory does not have the capability to reorder memory accesses, then these bus operations are treated as no-ops. If the memory can reorder memory accesses, then the memory must obey the barrier function of these bus operations.

8.15 The Definition of a Bus Adapter "Bridge-Chip"

This section describes the issues related to adapting the 620 bus to another bus.

- Non-Pended Bus Adapter—A non-pended bus is a bus that does not support retry as a mechanism to allow a higher priority operation to hold off a lower priority operation until the higher priority operation can get onto the bus. The following guidelines must be followed when a bus adapter connects a non-pended bus to the 620 bus.
  - Non-Pended Bus Memory Space—Memory that is mapped to the non-pended bus may not be mapped write-back, thus allowing a cache block to be in the modified state and to be cast-out or pushed to the non-pended bus. The result of having a push or cast-out directed towards a non-pended bus and to have the non-pended bus access that block would be to deadlock. Violating this rule can cause the deadlock described by Section 8.22.2, "The Non-Pended Push Deadlock."
  - PowerPC 620 Bus Masters and Push Buffers—All 620 bus masters must set aside at least one push buffer that can not be used for processor stores or cast-outs. In addition each master must guarantee that if there are other stores or cast-outs and they are constantly retried, the push buffer will sooner or later win internal arbitration and get a chance to arbitrate and run on the bus.

    Therefore, if all store/copy-back buffers are filled with stores to a non-pended bus, and a non-pended bus device wants to read memory, and the non-pended read will cause a push to memory, then the push buffer will sooner or later be available for the push caused by the non-pended read, and will get to the bus even though the stores to the non-pended bus continually get retried.
- The ReRun Mechanism—Refer to Section 8.13, "The ReRun Mechanism."
- TLBIE/TLBSYNC—If a bus adapter passes TLBSYNC and TLBIE between two nodes, then software must guarantee that only one processor on each node is allowed to issue TLBIE and TLBSYNC at a time. The 620 bus only allows a single TLBIE/TLBSYNC to be active at a time.
- SYNC—A Sync bus operation, originating on node A, and passed from node A to node B must not block operations on node B that need to complete on node A before the Sync bus operation on both nodes can be complete.

- Guarded Space—Any accesses to guarded memory cannot occur until ASTATIN and ARESPIN responses indicate that the operation can proceed. Aborted operations cannot speculatively access guarded space.
- ARESPOUT Retry Generation—Bus adapters cannot ARESPOUT Retry operations that are not directed to their own space. Livelock scenarios arise if bus adapters start retrying operations to another adapter's address space.

8.16 PIO Load and Store Bus Operations

The PowerPC architecture defines separate address spaces defined as the normal memory segment and the direct-store segment. They are distinguished architecturally by the segment table entry T-bit. If T=0 then the reference is a normal memory segment access, implemented by memory bus operations. If T=1 then the reference is a direct-store segment access, implemented by Programmed IO (PIO) bus operations.

Refer to the *PowerPC Microprocessor Family: The Programming Environments* for the architectural definition of the direct-store segment.

The architectural ramifications of the direct-store segment are listed as follows:

- Direct-store segment accesses and the resultant PIO bus operations are strongly ordered.
- They must also provide synchronous error reporting capability.
- A PIO reply error causes a DSI exception.

PIO bus operations and memory bus operations use the same processor interface.

8.16.1 PIO Bus Operations

The following instruction types are supported to the direct-store segment (T=1), in the same manner as the normal memory segment that is cache inhibited (T=0, I=1), with the additional architectural constraints of strongly ordered, synchronous error reporting, uncachable, and not memory coherent.

- Loads and Stores (LB*, LH*, LW*, LD*, STB*, STH*, STW*, STD*)(Note: The * is a wild-card for all instruction mnemonics with this prefix.)
- Load and Store Multiple (LM*, STM*)
- Load and Store String (LS*, STS*)

8.16.1.1 The Data Size of PIO Bus Operations

Section 8.8, "ASIZEDATA[0–3] and ASIZEBURST Definition," defines the legal PIO bus operation transfer sizes for the direct store segment. (Note that 16 byte sized transfers are not supported by the PIO mechanism and that the 620 will split direct store transfers that cross a double-word boundary into at least two PIO transfers. This is because the 620 will not gather direct store segment (PIO) stores.)

8.16.1.2 PIO Instructions Must Wait for an Error Free Reply

Direct Load and Store instructions to the direct store segment cannot retire until an error-free reply is received from the addressed BUC.

8.16.2 PIO Loads and Stores

5 PIO bus operations are defined in the 620, as shown in Table 8-41. These PIO operations allow for communication between the 620 and the BUCs. A single direct-store segment store or load instruction will generate one or more PIO bus operations and, optionally (for loads), one PIO Reply bus operation from the addressed BUC.

Table 8-41. PIO Bus Operations

PIO Operation	Type	Direction
PIO Load Immediate	A-Only/D-Only	620 => IO
PIO Load Last	A-Only/D-Only	620 => IO
PIO Store Immediate	Address-Data	620 => IO
PIO Store Last	Address-Data	620 => IO
PIO Reply	Address Only	IO => 620

AStat for PIO loads and stores supports positive acknowledge. Section 8.4.10, "Address Status Acknowledge." AResp is not valid and is ignored for PIO bus operations. See Section 8.4.3.2, "Address Status and Address Response Validation."

8.16.3 PIO Store Operations

A direct-store access store is comprised of 0 or more PIO Store Immediates, 1 PIO Store Last and 1 PIO Reply, in that order.

Each PIO Store Immediate and PIO Store Last operation transfers data from the 620 to the BUC. The PIO Store Last tells the BUC that this is the last PIO data transfer and that the BUC should respond with a PIO Reply. The PIO reply from the BUC provides the 620 with the E-bit status for the direct-store access store.

8.16.4 PIO Load Operations

A direct-store access load is comprised of 0 or more PIO Load Immediates, 1 PIO Load Last, and optionally 1 PIO Reply, in that order.

Each PIO Load Immediate and PIO Load Last operation transfers data to the 620 from the BUC. The PIO Load Last tells the BUC that this is the last PIO data transfer and that the BUC should respond with a PIO Reply if an error was detected for the PIO load bus operation sequence. The BUC indicates whether an error was detected for the PIO load immediate/last bus operation sequence by asserting $\overline{\text{DERR}}$ for the PIO load last. This indicates to the 620 to expect a PIO reply associated with the PIO Load Last. If $\overline{\text{DERR}}$ for the PIO load last is not asserted, then 620 will not expect a PIO reply.

8.16.5 PIO Reply

The following subsections discuss PIO reply.

8.16.5.1 PIO Reply in Response to a PIO Load/Store Last Operation

The PIO Reply is an address-only transaction that is used by the PIO slave to indicate error status to the PIO master. PIO Store Last operations are always followed by a PIO Reply. PIO Load Last operations are followed by a PIO Reply only if $\overline{\text{DERR}}$ for the PIO Load Last operation is asserted. If $\overline{\text{DERR}}$ is asserted for the PIO Load Immediate, then the 620 will take a machine-check exception.

8.16.5.2 PIO Reply E-bit

If the E-bit of the PIO Reply is set, then the 620 will issue a DSI exception. See Section 4.6.3, "DSI Exception (0x00300)." If the E-bit is clear, then the PIO operation and the direct memory segment operation that caused the PIO operation will complete without an exception.

8.16.5.3 Flow Control for PIO Reply

Only the PIO master is allowed to flow control a PIO Reply. The 620, as a PIO master, will never flow control a PIO reply. that is, The 620 will always accept the PIO reply in response to the PIO operation in progress. Further, the 620 will not wait for the ASTATIN window, nor will the 620 recognize any attempt to flow control the operation.

8.16.5.4 PIO Reply Received Before or After the Load Last $\overline{\text{DERR}}$

The PIO Reply may be received before or after sampling $\overline{\text{DERR}}$ for the PIO Load Last data. The 620 will wait until the latter of receiving the $\overline{\text{DERR}}$ for the load data and the PIO Reply before internally completing the PIO Load Last and posting a DSI.

8.17 Memory and Cache Coherence

The following sections provide information on the memory and cache coherency of the 620.

8.17.1 Physical Memory Size

The PowerPC architecture supports a 2^{64} byte physical memory address space. The 620 bus architecture supports a 2^{48} byte physical memory address space. The 620 supports a 2^{40} byte physical memory address space. The upper 8 bits of the 48-bit bus address will be driven 0 as an output and ignored as an input. Refer to Section 8.9, "Address Bus."

8.17.2 Cache Block Size

The cache block is the smallest increment of memory over which coherency information is maintained. This is also called the coherency block. The cache block is an aligned 64-byte block (four quad-words).

8.17.3 WIMG Bit Definitions

This section contains the details that are 620 implementation specific.

8.17.3.1 Write Through

The W-bit is defined as follows.

- 1 - Write Through—Stores to write through blocks (W=1) will update memory immediately with non-burst writes. If the block is in the cache, the store will also update the cache's copy, and leave the state of the cache unchanged (S,E, or M). If the block is not in the cache (I), the block is not allocated into the cache. The 620 will write the data to memory with the same size and alignment as the store.
- 0 - Write Back (not write through)—Stores to write back blocks (W=0) will only update the cache and the block in the cache will be marked modified. Store misses will cause the blocks to be read into the cache, where they are marked modified. Modified blocks can be written back to memory, at a later time, using the burst write bus operation.

When the cache-inhibited bit is set, the W bit is a don't care.

8.17.3.2 Cache Inhibit

The I-bit is defined as follows:

- 1 - Cache Inhibited
- 0 - Cache Enabled

Cache-enabled memory accesses index into the cache. Except for DCBST, DCBF and DCBI, cache-inhibited memory accesses do not index into the cache and will not detect cached blocks if they exist in the cache.

8.17.3.3 Memory Coherent

The M-bit is defined as follows:

- 1 - Memory Coherent—The 620 will snoop all bus operations marked memory coherent, independent of the I-bit.
- 0 - Not Memory Coherent—The 620 snooper will ignore all bus operations marked not memory coherent (with the exception of TS, TI, SY and IK).

8.17.3.4 Guarded

The G-bit is defined as follows.

- 1 - Guarded—The 620 will not speculatively access guarded pages in main memory.
- 0 - Not Guarded—The 620 will speculatively access non-guarded pages in main memory.

8.17.4 Supported WIMG Combinations (Memory Access Modes)

The following abbreviations are used to the clarify the WIMG codes.

- "Ca" and "Noca" indicate cacheable and non-cacheable, respectively.
- "Wb" and "Wt" indicate write-back and write-through, respectively.
- "Coho" and "Noco" indicate "coherent" and "non-coherent," respectively.

There are 6 memory access modes supported by the 620. They are listed below and followed by the abbreviation that will be used in this document.

- WIM = 001: Cache Enabled, Coherent, and Write-Back (Ca Coho Wb)
- WIM = 101: Cache Enabled, Coherent, and Write-Through (Ca Coho Wt)
- WIM = X11: Cache Inhibited, and Coherent (Noca Coho)
- WIM = 000: Cache Enabled, Non-coherent, and Write-Back (Ca Noco Wb)
- WIM = 100: Cache Enabled, Non-coherent, and Write-Through (Ca Noco Wt)
- WIM = X10: Cache Inhibited, and Non-coherent (Noca Noco)

8.17.5 WIMG-Bit Overrides

There are several ways to control the WIMG bits on 620. The MMU provides a base value for WIMG (generated by the translation mechanism) which can be overridden by several configuration bits as detailed below:

The following instructions and operations have hardwired PTE WIMG bits. WIMG values are sourced from the PTE WIMG bits and 0 and 1 indicated hardwired values.

Table 8-42. Hardwired PTE WIMG-Bit Instructions/Operations

WIMG	Instruction/ Operation	Definition/Reason
XX0X	All	The M-Bit is forced to a 0, or deasserted, when the BUSCSR SPR bit BUSSNPEN is deasserted. Refer to Section 2.1.2.4, "Bus Status and Control Register (BUSCSR)."
XX1X	LARX and STCX	The M-Bit is forced to 1 if the operation is atomic (except LARX Reserve - see Table 8-18), caused by either the LARX or STCX instructions. Note that LARX/STCX is not supported until BUSSNPEN is asserted.
X10X	All	All Instruction fetch accesses are marked I=1 and M=0 if BUSSNPEN is deasserted. All Data Loads and Stores are marked I=1 and M=0 if the DL1 is disabled. See BUSCSR bit BUSSNPEN in Section 2.1.2.4, "Bus Status and Control Register (BUSCSR)," and HID0 bit DCE in Section 2.1.2.3, "Hardware Implementation-Dependent Register 0 (HID0)."
1011	MMU Load and store	The MMU RC bit stores for the 620 are write through real mode, which is simpler than copy-back mode. Since the W-bit is a don't care for loads, MMU loads also use 1011.
0101	ExtConIn ExtConOut	External Control In and Out are treated as guarded and cache-inhibited.

8.17.6 Inconsistent PTE WIM-Bit Memory Modes

Inconsistent WIM memory modes are when two virtual pages, marked with different WIM settings, alias to the same physical page. All memory instructions, including cache management instructions, are subject to the restrictions stated in this section.

The 620 does not guarantee hardware memory-cache coherence for any WIM inconsistency except for WIM = (001,101) Write-Through and Write-Back Memory Coherent.

8.17.7 Coherency Paradoxes

The following sections list the identified types of coherency paradoxes and how they are handled by the 620.

8.17.7.1 L1/L2 Not Coherent

The following cases are L1/L2 not coherent: E/M, M/E, E/E, M/S, E/S, M/I, E/I, S/I. The 620 does not detect these cases and exhibits unspecified behavior if they occur.

8.17.7.2 ARESPIN Paradoxes

ARESPIN paradoxes occur when ARESPIN is not what it should be for a coherent cache memory system. ARESPIN paradoxes are described below. See Section 8.17.17, "Master Cache State Transitions Due to Instructions," and Section 8.17.18, "Snooper Cache State Transitions Due to Bus Operations."

- Master or Snooper and ARESPIN is reserved—A machine-check exception will be made if a reserved AResp code is detected. This rule dominates over the next rule.
- Master or Snooper and ARESPIN is invalid due to ASTATIN—ARESPIN is ignored. Any paradox is not detected.

8.17.7.3 Cache-Inhibited Paradoxes

Cache-inhibited memory accesses do not index into either the L1 or L2 caches. Paradoxes between cache-inhibited operations and data in the L1 or L2 will not be detected by the 620.

8.17.8 Multi-Level Cache Definition

A multi-level cache is a hierarchy of levels that work together to function as one cache. The nomenclature used in this specification is the L1 is the highest cache level and the level closest to the processor. The L3 is the lowest level cache; the L3 is the level furthest from the processor.

8.17.8.1 The Definition of L1, L2, and L3

There are three levels of caches defined for the 620: L1, L2, and L3. The L1 is the first level cache completely internal to the 620. The L2 is the second level cache whose controller and interface are built in to the 620 but the tag and data memory is external. The L3 is the third level of cache and is external to the 620. Refer to Section 9.3.1.8, "L2CLC[0–1] Bits."

8.17.8.2 Inclusivity

A requirement for multi-level cache coherence is that the level above must be inclusive in the level below. Maintaining this inclusivity will be referred to within this specification as 'vertical cache coherence'. Note that inclusivity is not required for instructio- only caches.

8.17.9 Time and Hierarchical Priority

The following sections provide information on the time and hierachical priorities on the 620.

8.17.9.1 Time Priority

"Time priority" prioritizes bus operations based on the order that they complete on the bus. There is no priority associated with the order that operations start on the bus. Number 1 is the highest priority.

1. A bus operation that has completed on the bus.
2. All bus operations that have not yet completed on the bus.

See Section 8.18, "Address Collision Detection and Handling (CD)."

8.17.9.2 Hierarchical Priority

"Hierarchical Priority" prioritizes bus operations based on their relation or direction with respect to the root of the hierarchy, the lowest level, which is the system coherency point. Number 1 is the highest priority.

1. A bus operation directed away from the root of the hierarchy.
2. A bus operation directed towards the root of the hierarchy.

8.17.9.3 Time and Hierarchical Priority

Hierarchical priority has priority over time priority.

1. Hierarchical Priority
2. Time Priority

8.17.10 Cache Coherency Protocol

Cache coherence is based on a distributed (snooping) versus a centralized (directory) coherence mechanism.

The 620 distributed mechanism places a cache block in all caches into 1 of 4 possible states, Modified, Exclusive, Shared, and Invalid, abbreviated as M, E, S, and I (MESI protocol).

8.17.10.1 Vertical Cache Coherence

Multiple cache levels together functionally form a single cache. Inclusivity of a cache level above in a level below is called vertical cache coherence.

8.17.10.2 Horizontal Cache Coherence

Coherency between different caches is called horizontal cache coherence. Horizontal cache coherence is maintained by all cache devices by snooping the bus below that level, and optionally the bus above that level.

The following tables illustrate the rules concerning cache state coherence in both the horizontal and vertical dimensions. Vertical lines represent the separation between multilevel caches. Horizontal lines represent the separation between cache levels, usually implemented by a bus interface. The box formed by these vertical and horizontal lines contains the allowable cache states for that cache level.

Note that the term exclusive means that this block is contained in this cache and no other (horizontal) cache. The term modified means that this block is *modified* with respect to memory.

8.17.11 Invalid (I)

Invalid (I) specifies that there is not a valid copy of this block in this cache. Nothing is implied about whether memory is the owner of this block.

Cache levels above a block marked I must be marked I. Cache levels below a block marked I may be marked in any of the four states. Other caches may have this block marked in any of the four states.

The following table describes the rules about I with respect to MESI. Level N on processor 1 is the frame of reference.

Table 8-43. I with Respect to MESI

Cache Level	P1	P2
L N-1	I	
L N	I	MESI
L N+1	MESI	

8.17.12 Shared (S)

A block marked Shared (S) indicates that this block is valid, not modified, and possibly shared in another cache.The memory is the owner of this block.

Levels above a block marked S may be marked I or S, but not E or M. Levels below a block marked S may be marked S, E, or M, but not I. Other caches may have this block marked S or I, but not E or M. (Note: Shared above Modified occurs when Invalid is above modified and the processor makes a load causing a read. Shared above Modified enables multiple caches to share an unmodified block that is Modified at a lower level.)

The following table describes the rules about S with respect to MESI. Level N on processor 1 is the frame of reference.

Table 8-44. S with Respect to MESI

Cache Level	P1	P2
L N-1	IS	
L N	S	IS
L N+1	SEM	

8.17.13 Exclusive (E)

A block marked Exclusive (E) indicates that this block is valid, not modified, and exclusive. No other cache may have a copy of this block. The memory is the owner of this block.

Levels above a block marked E may be marked S or I, but not E or M. A block marked E must be at the lowest cache level. Other caches at this level must have this block marked I.

The following table describes the rules about E with respect to MESI. Level N on processor 1 is the frame of reference.

Table 8-45. E with Respect to MESI

Cache Level	P1	P2
L N-1	IS	
L N	E	I

8.17.14 Modified (M)

Modified (M) specifies that this block is valid, modified, and exclusive. The cache in state M is the owner of this block.

Levels above a block marked M may be marked I, M, or S (a block may be marked E only at the lowest level). Levels below a block marked M may only be marked M. Other caches must have this block marked I.

Table 8-46 describes the rules about M with respect to MESI. Level N on processor 1 is the frame of reference.

Table 8-46. M with Respect to MESI

Cache Level	P1	P2
L N-1	MSI	
L N	M	I
L N+1	M	

8.17.15 Ownership

The concept of ownership is applied to the memory location that will service a read.

- The M State—Since a block marked M is the only copy of a block in the system, then it must be the owner because the data can only be sourced from the modified copy.
- The E State—The E state, for the 620, defines memory or a lower level as the owner, not a cache block marked E. This precludes other bus devices from defining the E state as the owner.
- The S State—The S state defines memory as the owner.

8.17.15.1 Transfer of Block Ownership Between Caches

The 620 implements intervention, the transfer of ownership between caches, for the RWITM bus operation, but not the Read bus operation. Refer to Section 8.7, "Intervention and Push Definition."

8.17.16 Cache State Transition Definition

The following sections define the cache state transitions for the 620 caches with respect to processor instructions and bus operations, one table for each. The third table defines the cache state transitions for a cache level below the 620 bus, which for this document will be called the "L3 cache".

Each row of the following tables is a case or set of cases that can be defined as a single statement. Each of the columns in the following tables are defined as follows:

- Instr./Op.—The "Instruction or Operation" column lists the instruction or operation that causes this state transition. All rows are instructions, except for deallocate which means "cache block deallocate".
- Memory Mode—The "Memory Mode" column refers to the "WIM" memory access mode bits found in the PTE, as described in the *PowerPC Microprocessor Family: The Programming Environments* user's manual. See Section 8.17.4, "Supported WIMG Combinations (Memory Access Modes)."

Note that many combinations of operations and access modes are not listed. Those that are not listed are considered to be paradoxical. Refer to Section 8.17.7, "Coherency Paradoxes."

- Coherency State—The "Coherency State" column defines the coherency for the addressed cache block. The possible states are limited to I, S, E, and M. See Section 8.17.10, "Cache Coherency Protocol ."

  The notation "->" means that any of the previous states on the left before this operation will go to the final state on the right after this operation. A list of states that do not have the "->" symbol will not change state.

  A blank box means that the cache does not get accessed.
- Bus Operation—The "Bus Operation" column defines the bus operation, if any, that this instruction/operation causes. Section 8.12, "Bus Operations."
- ARESPOUT/ARESPIN
  - — ARESPOUT is the output response caused by this instruction/operation.
  - — A blank ARESPOUT defines the Null response.
  - — ARESPIN is the input response combined from all output responses.
  - — A blank ARESPIN means that the snoop response for this case is "don't care".
  - — "S,Null": The final state is S if "Shared", E if "Null".
  - — "M,$\overline{M}$" and "Retry,$\overline{\text{Retry}}$" indicate whether the response is modified or retry.
- Comments—The definition of "C -> CM" is "cache-to-memory-and-cache" and "C -> C" is "cache to cache". Section 8.7, "Intervention and Push Definition."

  "E if the lowest cache level" means that the final state is E if the level of concern is the lowest cache level in the system, else the final cache state is S.

8.17.17 Master Cache State Transitions Due to Instructions

The following table describes cache state transitions that occur due to processor instructions given the following constraints:

- LD and ST are all loads and stores except for LARX and STCX.
- M=X: The PTE M-bit is ignored by the cache state table except for DCBTST. Refer to Section 8.12, "Bus Operations," for the computation of the bus M-bit.

Table 8-47. Master Cache State Transitions Due to Instructions

Num	Instruction/ Operation	Memory Mode	Coherency State	Bus Operation	AResp In	Comments
1	LD, DCBT	Ca	MES			
2	LARX		MES	LARX-Reserve		LARX-Reserve issued if L3 is enabled (page 69)
3						
4	LD, DCBT,LARX, DCBTST M=0		I -> S	Read-Burst	M	M indicates data intervention
5			I -> SE	Read-Burst	S,Null	
6	LD, LARX	Noca		Read-Non-Burst		
7	DCBT					No Op on Noca blocks
8	ST, DCBTST,STCX	Ca Wb	M			
9	ST, STCX		E -> M			
10	DCBTST		E			
11	DCBTST M=0		S			
12	ST, STCX, and DCBTST M=1		S -> M	DClaim		
13			I -> M	RWITM	M,$\overline{M}$	M indicates data intervention
14	ST	Ca Wt	MESI	Write with Flush		STCX not supported for Ca Wt
15	ST,STCX	Noca		Write with Flush		
16	DCBTST	Ca Wt Noca				No Op on Wt or Noca blocks
17	Deallocate	Ca	M -> I	Write with Kill (CB)		
18			ESI -> I			
19	DCBF		M -> I	Write with Kill (F)		
20			ESI -> I	Flush		
21	DCBI		MESI -> I	DKill		
22	DCBST		M -> SE	Write with Clean		E if no L3
23			SE -> SE	Clean	S,Null	Clean optional for ES if no L3
24			I	Clean		
25	DCBZ	Ca Wb	EM -> M			
26		Ca Wb	IS -> M	DClaim		
27		Ca Wt Noca				System Alignment Error Int

8.17.18 Snooper Cache State Transitions Due to Bus Operations

The following table describes the snooper cache state transitions that occur due to snooped bus operations given the following constraints:

- A,W,I=X: These bits are ignored by the 620 as a bus snooper.
- M=0: Bus operations that are marked not memory coherent (M=0) may be snooped by the 620 but do not affect cache states.
- M=1: Bus operations that are marked memory coherent (M=1) are snooped by the 620 as a bus snooper, independent of the cache inhibited I-bit.
- Write with Clean bus operation will always be marked M=0, and thus will be ignored by the 620 snooper.
- See Section 8.7, "Intervention and Push Definition"
- Rsrv state: R = 1 indicates a valid reservation with a matching address. R = 0 indicates an invalid reservation or a non-matching address.

Table 8-48. Snooper Cache State Transitions Due to Bus Operations

Num	Bus Operation		Snooper State	Rsrv State	AResp Out	AResp In	Comments
1	Read-Burst	N=1, S=0 N=1, S=1, $\overline{\text{l2en}}$ N=1, S=1, l2en, l3en	M -> S		M		Causes C -> MC data-only op. (Intervention)
2		N=1, S=1, l2en, $\overline{\text{l3en}}$	M -> E		M		
3		N=0, S=0 N=0, S=1, $\overline{\text{l2en}}$ N=0, S=1, l2en, l3en	M -> S		Retry		Causes Write with Clean (Push)
4		N=0, S=1, l2en, $\overline{\text{l3en}}$	M -> E		Retry		
5			S		S	Note3	
6		S=0	E -> S		S	Note3	
7		S=1			S	$\overline{\text{ReRun}}$	
8		S=1	E		S	$\overline{\text{ReRun}}$	
9			I	R=0			
10				R=1	S		

Table 8-48. Snooper Cache State Transitions Due to Bus Operations (Continued)

Num	Bus Operation		Snooper State	Rsrv State	AResp Out	AResp In	Comments
11	RWITM	N=1	M -> I		M		Causes C->C data-only op. (Intervention)
12		N=0	M -> I		Retry		Causes Write with Kill (Push)
13			ISE -> I			$\overline{\text{ReRun}}$	
14			E -> S			ReRun	
15			IS				
16	Write-With-Kill, DKill, DClaim		IESM -> I			$\overline{\text{ReRun}}$	
17			E -> S			ReRun	
18			ISM				
19	Write-With-Flush		M -> I		Retry		Causes Write with Kill (Push)
20			ISE -> I			$\overline{\text{ReRun}}$	
21			E -> S			ReRun	
22			IS				
23	Read-Non-Burst		M -> S		Retry		Causes Write with Clean (Push)
24			SE -> S		S	Note3	
25			I	R=0			
26				R=1	S		
27	Clean		M -> SE		M[1]		Causes Write with Clean (Push) E if lowest cache level, S otherwise
28			S		S		
29			E		S[2]	$\overline{\text{ReRun}}$	
30			E -> S		S[2]	ReRun	
31			I	R=0			
32				R=1	S		

Table 8-48. Snooper Cache State Transitions Due to Bus Operations (Continued)

Num	Bus Operation	Snooper State	Rsrv State	AResp Out	AResp In	Comments
33	Flush	M -> I		M[1]		Causes Write with Kill (Push)
34		ISE -> I			$\overline{\text{ReRun}}$	
35		E -> S			ReRun	
36		IS				
37	SYNC, TLBSYNC			ReRun, Null		Will ReRun until done. Will Null when done.

Notes:

1. M overrides ReRun. This is a performance optimization.
2. ARESPOUT = Shared is not significant to any other snooper which must have this block marked invalid. Note that the L2 E state implies that there is no L3 cache.
3. Read and Read-Non-Burst will mark the block S for the ReRun response, as well as the Null and Shared responses.

8.17.19 L3 Cache State Transitions Due to Bus-Above Operations

The L3 is not implemented by the 620. This section is intended to show how coherence would be maintained between the 620 and a cache external to the 620. Refer to Figure 8-32.

The following table describes the L3 state transitions that occur due to bus operations given the following constraints:

- A,W,I,M,N=X: These bits are ignored by the L3 as a bus-above snooper.
- All bus operations are snooped, independent of the M-bit and I-bit.
- M in L2
  - — M in L2 is undefined when the state of a block is not modified.
  - — M in L2, when asserted, indicates that the block is modified at a higher level.
  - — M in L2, when deasserted, indicates that this block is the highest level that is marked modified and that the valid data is in this cache level.
- Comments
  - — "Bus Above" means that the actions are taken with respect to the bus above the L3.
  - — "Bus Below" means that the actions are taken with respect to the bus below the L3.

- — The L3 must know what memory space(s) are mapped above and below the L3.
  - – "Address Above" means that the L3 decodes the address to be above the L3
  - – "Address Below" means that the L3 decodes the address to be below the L3
- — "1a or 1b" indicates that either 1a or 1b, but not both.

- ARESPIN Retry—Unless otherwise specified the ARESPIN retry response restores the L3 to the initial state.

Table 8-49. L3 Cache State Transitions Due to Bus-Above Operations

Num	Bus-Above Operation	M In L2	Snooper State	AResp Out	AResp In	Comments
1	Read Burst, RWITM, Write-With-Flush, Read-Non-Burst, Clean, Flush	Y	M			No Action
2	Read-Burst	N	M	M		Bus Above: Source Data-Only Operation
3			SE	S		Address Above: No Action Address Below: Bus Above: Data-Only Operation
4			I -> SE	S		Address Above: 1. L3 allocates cache block 2. Bus Above: Sink Data-Only Operation Address Below: 1. Bus Below: Read-Burst, E-state if lowest cache level and S response, else S-state. 2. Bus Above: Source Data-Only Operation

Table 8-49. L3 Cache State Transitions Due to Bus-Above Operations (Continued)

Num	Bus-Above Operation	M In L2	Snooper State	AResp Out	AResp In	Comments
5	RWITM	N ->Y	M	M Null		Address Above: Data is sourced by the L3 with the M response. Address Below: Bus Above: Source Data-Only Operation with Null response.
6		N ->Y	E -> M	Null		Address Above: No Action and Null response because data is sourced from memory. Address Below: Bus Above: Source Data-Only Operation with Null response.
7		N ->Y	S -> M	Null ,M		Address Above: 1a. No Action and Null response if data is sourced from memory. 1b. Modified response if data is sourced by L3. Address Below: 1. Bus Below: DClaim 2. Bus Above: Source Data-Only Operation with Null response.
8		N ->Y	I -> M			Address Above: 1. L3 allocates cache block 2. Bus Above: Sink Data-Only Operation Address Below: 1. Bus Below: RWITM 2. Bus Above: Source Data-Only Operation with Null response.
9	Write-With-Kill, DKill	-> N	IESM -> I			Address Above: No Action Address Below: Bus Below: Write-With-Kill or DKill
10	Write-With-Clean	Y -> N	M -> SE			Address Above: No Action, E if lowest cache level. Address Below: Bus Below: Write-With-Clean, E if lowest cache level.
11	DClaim	N ->Y	IESM -> M			MInL2 must start as N Bus Below: DClaim if initial state is M

Table 8-49. L3 Cache State Transitions Due to Bus-Above Operations (Continued)

Num	Bus-Above Operation	M In L2	Snooper State	AResp Out	AResp In	Comments
12	Write-With-Flush, Read-Non-Burst	N	M -> I	Retry		Address Above: 1. Bus Above: Write with Kill (Push) 2. Bus Below: DKill if lower cache level exists. Address Below: Bus Below: Write with Kill (Push)
13			ISE -> I			Address Above: No Action Address Below: Bus Below: Write-With-Flush or Read-Non-Burst
14	Clean	N	M -> ES	S		Address Above: 1. Bus Above: Write with Clean, E-state if lowest cache level, else S-state. 2. Bus Below: Clean Address Below: 1. Bus Below: Write with Clean
15			ES	S		Bus Below: Clean
16			I			Bus Below: Clean
17	Flush	N	M -> I			Address Above: 1. Bus Above: Write with Kill (Push) 2. Bus Below: DKill if lower cache level exists. Address Below: Bus Below: Write with Kill (Push)
18			E -> I			No Action
19			IS -> I			Bus Below: Flush

8.18 Address Collision Detection and Handling (CD)

The purpose of this section is to describe the mechanism used to avoid the interaction between multiple bus operations. Operations that may interact are referred to "collide" with each other. The mechanism that detects collisions is called the "collision detection" (CD) mechanism. The address granularity of CD is the cache block address. (See Section 8.18.3, "CD-Rules.")

8.18.1 Bus Operation Serialization

Bus execution serialization with respect to the address bus means that two or more operations are serialized from beginning to end. Bus operations that collide are execution serialized. Bus operations that do not collide are not execution serialized.

There are two bus device types presently defined with respect to collision detection.

- S-Only: A device can only cache in the S state. (Not ME states)
- MES: A device can cache in the MES cache states and supports connectivity with one or more S-Only device types.

The 620 is an MES-type device. Note that if there are no S-Only type devices, then MES devices would not have to obey Section 8.18.3.6, "Rule 6: CD Between Snoop Buffers." The following table defines which CD rules must be followed by each device type in order to achieve correct operation. Each of the rules are defined in subsequent sections.

Table 8-50. CD Rules: Rules a Bus Device must Follow

Device Type	CD Rules
MES	Section 8.18.3, "CD-Rules" Section 8.18.3.2, "Rule 2: CD Previous Adjacent ASTATOUT Retry" Section 8.18.3.3, "Rule 3: CD Disabled Bus Operations" Section 8.18.3.4, "Rule 4: Operations that Take CD Priority" Section 8.18.3.5, "Rule 5: CD Based on Completion" Section 8.18.3.6, "Rule 6: CD Between Snoop Buffers"
S-Only	Section 8.18.3, "CD-Rules" Section 8.18.3.2, "Rule 2: CD Previous Adjacent ASTATOUT Retry" Section 8.18.3.3, "Rule 3: CD Disabled Bus Operations" Section 8.18.3.7, "Rule 7: CD Requirements for an IO Device"

8.18.2 CD States and State Transitions

All 620 operations that use collision detection (CD) must be in one of three CD states—OUT, IN or INBUSY. It is not necessary for other bus devices to implement these same states as long as the CD rules are followed. These states are as follows:

- The OUT State—The OUT state indicates that the operation has not passed collision detection. An operation that is in the OUT state may only snoop the L2 cache to determine coherency state and change CD states. An operation that is in the OUT state may not modify any state that may be shared with another operation. No operation will collide against another operation that is in the OUT state. All operations start and end in the OUT state.
- The INBUSY State—The INBUSY state enables the operation to be uninterrupted when an operation needs to perform multiple tasks that must be indivisible. An operation may go from the OUT state to the INBUSY state when the operation has passed CD, and may stay in the INBUSY state until the operation is complete or it is determined that the operation needs to go to the bus. An operation may go from the IN state to the INBUSY state when the address bus operation response indicates completion and may stay in the INBUSY state until the operation is complete.

- The IN State—The IN state indicates that the operation has passed CD and needs to go to the bus. Bus operations may share CD for the same block. The first bus operation that completes on the bus will go into the INBUSY state. All other bus operations that share CD to that block will be forced from the IN state to the OUT state. This CD sharing function is handled by the CD collection logic.

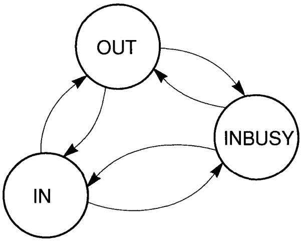


Figure 8-33. Collision Detection State Diagram

Table 8-51 provides CD state transition information.

Table 8-51. Collision Detection State Transitions

State Transition	Description
OUT -> IN	A snooper operation that passes CD will transition from the OUT state to the IN state.
OUT -> INBUSY	The following conditions will cause this state transition: A master operation that passes CD will transition to the INBUSY state until the L2 cache state is determined. If the master operation can be completed without going to the bus, then the operation stays in the INBUSY state until completion.
INBUSY -> OUT	Operations that complete in the INBUSY state can at any time give up the CD privilege and go from the INBUSY state to the OUT state.
INBUSY -> IN	If the 620 master operation determines, upon receiving the L2 cache state, that a bus operation is needed, then a state transition is made from INBUSY to IN.
IN -> INBUSY	When an operation completes on the bus the master or snooper may transition into the INBUSY state and not be interrupted until the completed bus operation is completed internally.
IN -> OUT	The following conditions will cause this state transition: 1. An operation can be forced from the IN state to the OUT state if another bus operation to the same block completes first. (Complete is ARESPIN Null, Shared, and Modified. Not complete is ARESPIN Retry and ReRun.) The operation forced into the OUT state then loses the ability to continue until CD is passed again and transitions to the IN or INBUSY states. 2. An operation, for the master or snooper, determines that no action is needed when completion occurs on the bus.

8.18.3 CD-Rules

The following sections discuss the rules for collision detection on the 620.

8.18.3.1 Rule 1: CD Address Precision

The address comparison precision for collision detection for the 620 is the 34-bit block address, given a 40-bit byte address and a 64-byte block (A[24:57]). All devices that require compatibility with the 620 will implement the same degree of precision.

- Non-A40 Bus Devices—Bus devices that implement more than 40 bits of physical address (for example, A44 or A48) must not include the upper address bits above A40 into collision detection if compatibility with the 620 is desired and addresses above A40 will be accessed. Bus devices that implement less than 40 bits of physical address (for example, A32) require the system not to address outside of the smallest address space device. (that is, An A32 device will limit the system to A32 addressing, even if there are A40 processors in the system.)
- 32-Byte Sized Block Bus Devices—Bus devices that implement smaller block sizes than 64 bytes, such as 32 bytes, must not include the lower order block address bits into collision detection that are not included in the 64 byte block address. Bus devices that implement larger block sizes than 64 bytes (for example, 128 bytes) must not exclude high order byte-in-block address bits into collision detection that are included in the 64-byte block address.

8.18.3.2 Rule 2: CD Previous Adjacent ASTATOUT Retry

CD that is based on completion needs to be able to retry all other operations that have not yet completed. See Section 8.18.3.5, "Rule 5: CD Based on Completion." Due to the overlap of ARESPOUT with ARESPIN it is necessary to keep bus operations to the same block from being too close as to make retrying the later operation impossible.

- Previous Adjacent ASTATOUT Retry Definition—When a master puts an address bus operation onto the bus the master will ASTATOUT Retry its own bus operation if the following condition is true. See Section 8.4.17, "Address Status Out and Address Response Out Retry." The condition exists when the previous address bus operation does not belong to this master, the present address bus operation does belong to this master, the previous address bus operation is adjacent, both addresses block address match (See Section 8.18.3, "CD-Rules."), and previous adjacent is enabled (Table 8-52).

  The first time that the 620 ASTATOUT retries its own bus operation due to the CD previous adjacent condition the 620 will reissue the bus operation without first issuing a Null bus operation. The 2nd to Nth time that the 620 ASTATOUT retries a bus operation due to the CD previous adjacent condition the 620 will first issue the Null bus operation and then reissue the bus operation. This will break the deadlock between devices that all want to issue bus operation that previous adjacent address match. See Section 8.22.3, "The Previous Adjacent Address Match Deadlock."

- The Definition of Previous Adjacent—The previous address bus operation is considered to be adjacent if it was sampled three or less BUSCLKs before the present address bus operation is sampled. Figure 8-34 illustrates an address A(0) and the sample points for three previous addresses. A(-3) and A(-2) are previous adjacent and A(-4) is not previous adjacent.

  Note that the definition of previous adjacent guarantees that for the BUSRESPTEN=3 configuration that there is a worst case of 1BUSCLK from ARESPIN(-4) to ARESPOUT(0) (see bottom of Figure 8-34). This allows 1BUSCLK for the completion of A(-4) to retry A(0). For simplicity of testing the definition of previous adjacent is the same when BUSRESPTEN is less than 3. See Section 8.4.6, "Address Status and Address Response Tenure (BUSRESPTEN[0–1])."

  Figure 8-34 defines when a previous address bus operation is adjacent.

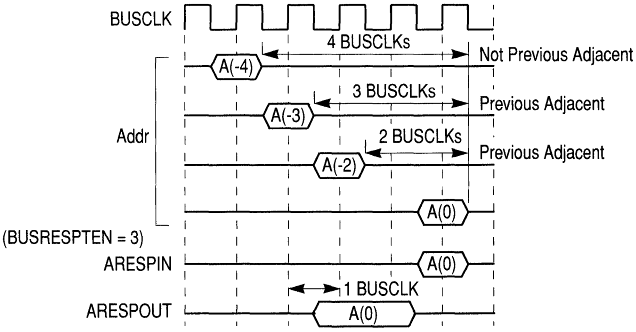


Figure 8-34. The Definition of Previous Adjacent

The following table defines previous adjacent is enabled.

Table 8-52. The Definition of Previous Adjacent is Enabled

Definition of Previous Adjacent is Enabled			Present Operation that is mine			
			RWITM, LR, WNB-A, DK, DC, FL, CL	Read		WBK[3], WBC[3], WNB-$\overline{\text{A}}$, Null, PIO Reply[1], TI, IK, EI, SY[2], TS[2], PLI, PLL, PSI, PSL, XCI, XCO
				I=0 or M=1	I=1 and M=0	
Previous adjacent operation that is not mine	RWITM, LR, DK, DC, FL, CL, WBK, WBC		Yes		No	
	Read, WNB	I=0 or M=1				
		I=1 and M=0	No			
	Null, PIO Reply, TI, IK, SY, EI, TS, PLI, PLL, PSI, PSL, XCI, XCO					

Notes:

1. The 620 will never be the master for PIO Reply and is included here for completeness.
2. Only SY and TS can be ReRun by the 620 as a master.
3. Store buffer operation will retry previous adjacent operation. See Section 8.18.3.4, "Rule 4: Operations that Take CD Priority."

8.18.3.3 Rule 3: CD Disabled Bus Operations

This section defines the bus operation types that do not collide even when both operations are to the same block address.

- Non-Physical Address Operations—Non-physical address operations do not collide. Non-physical address operations do not participate in the cache coherency protocol. The non-physical address operations are:
  — XCI, XCO—The graphics commands use the address bus to pass an address to the DMA controller.
  — PLI, PLL, PSI, PSL, PIO Reply—PIO operations are to PIO address space.
  — SY, TS, EI—SYNC, TLBSYNC, and EIEIO are address-less operations.
  — TI—The TLBIE address is virtual.
- The IKill Bus Operation—Although the IKill bus operation has a physical address, hardware does not maintain coherence for instruction only caches. Software must maintain coherence when executing the ICBI instruction.

8.18.3.4 Rule 4: Operations that Take CD Priority

This section defines when a master operation may take CD priority over a snooped bus operation and ARESPOUT Retry the snooped bus operation. See Section 8.4.17, "Address Status Out and Address Response Out Retry."

Table 8-53. Collision that may immediately retry the snooped bus operation

Case Number	Snooped Bus Operation	620 Master Bus Operation	Description
1	All AResp-enabled M=1 bus operations. (See Table 8-1)	All Block writes or intervention data transfers in the store buffer.[1] (Copy-backs, Pushes, block writes due to flush and clean, intervention data transfers.)	Data in the store buffer can not be read or written and is considered to be an operation in progress that cannot be interrupted.
2		Any operation that is in a critical section and can't be interrupted. 1. Have passed CD but do not yet have cache state to determine if a bus operation is necessary. 2. Completed address bus operation and either haven't yet updated the cache state or haven't completed the data read or write transfer.	An operation that does not need the bus to complete is allowed to retry snooped bus operations to the same block.

Note: A collision on any low priority store in the store buffer will cause that store to be marked as a high priority store. See Section 8.3.2, "High-Priority Bus Operations," and Section 8.3.4, "Internal Request Arbitration."

8.18.3.5 Rule 5: CD Based on Completion

In general, if an operation does not fall under either Section 8.18.3.3, "Rule 3: CD Disabled Bus Operations," or Section 8.18.3.4, "Rule 4: Operations that Take CD Priority," then priority is determined by the order of completion. When the priority between multiple operations that collide is determined by completion then one operation must succeed and all other must be retried. Completion is defined as ARESPIN is not Retry or ReRun.

Figure 8-35 and Figure 8-36 describe two aspects of CD when ordering is with respect to completion. Figure 8-35 defines CD between two bus operations that overlap in time. Figure 8-36 defines CD between two bus operations that complete out of order.

- CD Between Two Bus Operations that Overlap in Time—Operation A starts on the bus first and operation B starts on the bus second. It is given that operation A is not previous adjacent to B, so B does not ASTATOUT retry itself. See Section 8.18.3.2, "Rule 2: CD Previous Adjacent ASTATOUT Retry."

  The following diagram describes the CD responsibility of the master and snoopers between operation A which is completing and operation B which is starting. Sampling ARESPIN for bus operation A occurs at the end of BUSCLK 0.

  The master for bus operation A will retry bus operation B if address B occurs prior to or during cycle 0, addresses A and B collide, address B came after address A, and operation A completes successfully. If address B occurs in cycle +1 or later, then the snoopers and master that may be changing state based on the completion of operation A may retry operation B. Operation B will not be retried when the last of the devices to change state based on the completion of operation A is finished.

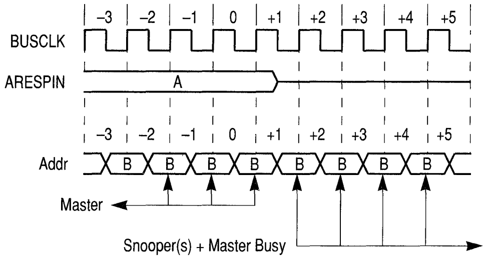


Figure 8-35. Address with Respect to Address Response

- CD between two bus operations that complete out of order—It is the responsibility of the master (A) to self ARESPOUT retry if another operation (B) starts after the master, collides, and completes before the master, Figure 8-36 illustrates this point. Operation A starts first and operation B starts second. Operation A gets ReRun at least once and operation B completes first. The master for operation A will see operation B complete first and will ARESPOUT Retry its own operation A.

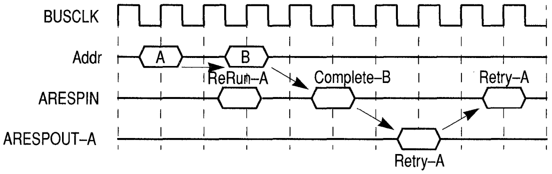


Figure 8-36. Out-of-Order Completion

- 620 as a Master—The 620 can issue two load operations as a master, one from the load port and one from the instruction fetch port. The 620 keeps track of which four bus snoop buffers collide with the two master ports using an 8-entry scoreboard. The scoreboard is described by the following table. A "Y" indicates a scoreboard bit that keeps track of a collision between a master port and a snoop buffer.

Table 8-54. Scoreboarding between the Master and Snoop Buffers

Master Operations		Snoop Buffers			
Port	**Port Operation**	**0**	**1**	**2**	**3**
Data Load	Read (I=0 or M=1), RWITM, LR, DK, DC, FL, CL	Y	Y	Y	Y
Instruction Fetch	Read (I=0 or M=1)	Y	Y	Y	Y

8.18.3.6 Rule 6: CD Between Snoop Buffers

Since the S-Only device does not obey the rule described by Section 8.18.3.5, “Rule 5: CD Based on Completion,” then the snooper needs to cover the period from sampling the address to sampling ARESPIN, as described by Figure 8-35.

The 620 can simultaneously snoop four bus operations. The 620 snooper keeps track of which snoop buffers collide with each other by using a 6-entry scoreboard, one scoreboard bit for each pair of snoop buffers that can collide. A “Y” indicates a scoreboard bit that keeps track of a collision between snoop buffers.

Table 8-55. Scoreboarding Between Snoop Buffers

Snoop Buffers	Snoop Buffers			
	0	**1**	**2**	**3**
0		Y	Y	Y
1			Y	Y
2				Y
3				

8.18.3.7 Rule 7: CD Requirements for an IO Device

Data may only be cached into the shared (S) coherence state.

1. Any of the following snooped bus operations will speculatively invalidate a pending read and require that the read data be discarded and the read operation reissued over again.
   - — Write-With-Flush
   - — Write with Kill M=1
   - — RWITM
   - — DClaim
   - — Flush
   - — DKill

2. Any of the following snooped operations will receive ARESPOUT Shared, independent of the completion status for the operation that the snoop operation collided against.

   — Clean

   — Read

3. IO should take no action for the following operations.

   — Write with Clean M=0.

   — LARX-Reserve

All other operations are CD-disabled. See Section 8.18.3.3, "Rule 3: CD Disabled Bus Operations."

8.19 TLB Coherency Control

Translation lookaside buffer coherency control is described in the following sections.

8.19.1 Page TLB

The 620 provides hardware assistance for handling TLB consistency. The 620 supports a hardware broadcast of TLBI instructions onto the bus. In addition, the 620 supports a hardware atomic update of the page table change bit.

8.19.1.1 The Software Procedure for TLB Invalidating

TLB invalidating sequences are implemented with the **tlbie**, **sync**, and **tlbsync** instructions and are fully described in the *PowerPC Microprocessor Family: The Programming Environments* user's manual.

8.19.1.2 Update of the TLB change bit (C-bit update)

The 620 processor provides hardware support for page table walking in the event of a TLB miss. The 620 guarantees that the change bit will be updated successfully before the operation that caused the C-bit update is started.

8.19.2 Block Address Translation (BAT) Registers

A second method for address translation of large blocks of memory is provided for in the 620 processor. The details of the translation can be found in the *PowerPC Microprocessor Family: The Programming Environments* user's manual. In general, this block translation function is maintained entirely by software. Therefore, the 620 does not provide any type of hardware coherency support for the update of BATs across processors in a multiprocessor system. This function is left up to software.

8.20 Atomic Memory Reference Support

This section defines 620 implementation-specific details.

- The Reservation Granule Size—The reservation granule for the 620, defined in the PowerPC architecture as the reservation block size that is snooped, is 1 aligned cache block or 64 bytes.
- A Reservation will not be lost when a cache deallocation occurs—The 620 will continue to hold the reservation when the cache block in which the reservation lies is deallocated. The reservation participates in the coherence protocol independent of the contents (valid/invalid) or mode (inhibited/enabled) of the cache.
- Snooped bus operations that clear the reservation—The following 5 bus operations will clear the reservation of the 620 if ARESPIN is not ReRun or Retry. Snooped bus operations other than those listed will not clear the reservation (RWITM, DClaim, Write with Flush, DKill, Write with Kill).
- The Definition of a Reservation—A reservation, as treated by the 620, is a multi-level single-entry tag-only cache that can only have two states, set and clear. The 620 prescribes a LARX/STCX solution that requires a reservation to be set at all levels in order for the STCX to succeed.
- Reservation Inclusivity—All reservations are inclusive in the same meaning of the word customarily used with respect to caches. This means that if a reservation is set at a higher level, then it must be set at all lower levels. If a reservation is cleared because of a store, a bus operation must be propagated to all higher levels to clear out the reservation.

Related Sections:

- Section 8.6.2, "Atomic Address Attribute (A-Bit)"
- Section , "The address command is defined by ATYPE[0–4] and A[0–7] according to Table 8-16. The following attributes are defined:"
- Section 8.17.17, "Master Cache State Transitions Due to Instructions"

8

8.20.1 Cache-Inhibited

- LARX—The LARX causes the reservation to be set and the Read-NonBurst-Atomic bus operation to be issued. The Read-NonBurst-Atomic bus operation will set all reservations at all levels below the processor. See Section 8.12.3, "Read."
- STCX—The STCX instruction will cause the Write-With-Flush-Atomic bus operation to be issued if the reservation is set. Otherwise, no bus operation will be issued and the STCX will fail.

  If the reservation is cleared by a non-previous adjacent address bus operation during the address tenure of the Write-with-Flush-Atomic, then the 620 will retry itself and the STCX will fail. If the reservation is set when the response to the Write-With-Flush-Atomic bus operation is Null (ARESPIN = Null), then the STCX succeeds and the reservation is cleared. See Section 8.12.6, “Write-With-Flush,” and Section 8.18.3.2, “Rule 2: CD Previous Adjacent ASTATOUT Retry.”

8.20.2 Write-Back

- LARX—The LARX causes the reservation to be set at each cache level and causes either the Read-Burst-Atomic bus operation to be issued if the addressed block is marked I or the LARX-Reserve bus operation to be issued if the addressed block is marked S, E, or M.

  The 620 will issue the LARX-Reserve bus operation only if the L3 is enabled. If the L3 is disabled the operation will complete internally without a LARX-Reserve bus operation. The Read-Burst-Atomic bus operation will set all reservations at all levels down to the highest level that has the addressed block valid. The LARX-Reserve bus operation is then issued to set all reservations from that level to the lowest level. See Section 8.12.3, “Read,” and Section 8.12.2, “LARX-Reserve.”

- STCX—If the reservation is clear, then the STCX fails and the data is not written to memory or the cache.

  If the reservation is set and the addressed cache block is marked I, then the STCX causes the RWITM-Atomic bus operation to be issued.

  If the reservation is set and the addressed cache block is marked S, then the STCX causes the DClaim-Atomic bus operation to be issued.

  If the reservation is set and the addressed cache block is marked E or M, then the STCX succeeds and no bus operation needs to be issued.

  The following text describes behavior after the bus operation is issued. If the reservation is set and the response is null, then the STCX succeeds and the data is written to the cache. If the reservation is clear and the response is retry or null, then the STCX fails and the STCX data is not written to the cache. See Section 8.12.1, “RWITM (Read-With-Intent-To-Modify),” and Section 8.12.9, “DClaim.”

8.20.3 Write-Through

The LARX and STCX instructions to the Write-Through memory access mode are not supported by the PowerPC architecture and will cause a DSI exception.

8.20.4 External Support for the Reservation

The following sections provide information on the external support for the reservation provided for by the 620.

8.20.4.1 The A-bit Address Attribute

All bus operations that are a direct result of either a LARX or STCX instruction are encoded on the bus with the atomic address attribute (for example, castouts caused by a **lwarx** do not have the atomic address attribute). Bus operations behave differently when marked atomic. Each bus operation will define how the atomic address attribute changes the bus operation function. See Section 8.6.2, “Atomic Address Attribute (A-Bit),” and Section 8.12, “Bus Operations.”

Note that this does not force all LARX and STCX instructions to generate bus transactions, but rather provides a means for identifying when they do.

If an implementation requires that all LARX and STCX instructions generate bus transactions, then the associated pages should be marked cache-inhibited.

8.20.4.2 The $\overline{\text{RESERVE}}$ Signal

The state of the reservation is always presented onto the $\overline{\text{RESERVE}}$ signal output pin. The latency from the internal reservation bit to the $\overline{\text{RESERVE}}$ signal will be no more than one BUSCLK cycle. This can be used to determine when an internal condition has caused the reservation bit to be set or reset.

The 620 does not require for a system to use this signal in order for LARX/STCX instructions to work in a multi-level cache hierarchy.

8.21 Processor Interface SPRs

The address map for the 620 SPRs is located in Section 2.1.2.3, “Hardware Implementation-Dependent Register 0 (HID0).” This section discusses several BUSCSR bits.

8.21.1 Bus Snooper Enable (BUSSNPEN)

The SPR bit BUSSNPEN enables the bus snooper. When BUSSNPEN is deasserted the snooper will ignore all bus operations and all bus operations issued by the 620 will have the bus M bit deasserted or forced to 0. Note that coherency between any two processors is not supported until the last processor snooper has been enabled. When BUSSNPEN is asserted the snooper will snoop bus operations and all bus operations issued by the 620 will not force the M bit to 0. The hardware reset state for BUSSNPEN is deasserted.

8.21.2 Bus Intervention Enable (BUSINTVEN)

The SPR bit BUSINTVEN enables intervention for the bus operations issued by the processor. Section 8.7.2, “Intervention Enable Bit (BUSINTVEN).” The hardware reset state is deasserted.

8.21.3 Bus Parity Error (BUSPARERR[0–2])

The SPR bits BUSPARERR[0–2] indicate that a bus parity error has occurred and is always set independent of the parity enable bits (HID0(EBA,EBD)). Refer to Section 8.11, “Parity Protection,” for the definition of BUSPARERR[0–2]. BUSPARERR[0–2] is deasserted by any write to BUSCSR SPR. The hardware reset state is deasserted.

Refer to Table 4-2 for the function of HID0(EBA,EBD). EBA and EBD do not effect the setting of BUSPARERR[0–2]. The hardware reset state for EBA and EBD is deasserted.

8.21.4 Bus Data Error and Enable (BUSDERR, BUSDERREN)

The SPR bit BUSDERR indicates that the bus signal $\overline{\text{DERR}}$ was asserted for a read data other than for PIO Load Last and is always set by this condition independent of the BUSDERREN SPR bit. BUSDERR is deasserted by any write to BUSCSR SPR. For PIO Load Last, the BUSDERR bit is set only if DErr received on a read data and a PIO Reply is received with the Error bit set (see Section 8.16.5, "PIO Reply"). The hardware reset state is deasserted. Section 8.5.9, "Data Error (DERR)."

The bus SPR bit BUSDERREN enables this error condition to cause a machine-check exception. This SPR bit does not effect the setting of BUSDERR or the behavior of the bus $\overline{\text{DERR}}$ signal. The hardware reset state is deasserted.

8.21.5 Bus Response Code Error (BUSRESPERR)

The SPR bit BUSRESPERR indicates that a reserved response code was detected for an address bus operation and is always set by this condition. A machine-check exception is initiated. The bus operation is aborted.

BUSRESPERR is deasserted by any write to BUSCSR SPR. The hardware reset state is deasserted. See Section 8.4.9, "Address Response In/Out."

8.21.6 Bus Positive Acknowledge Error (BUSPOSACKERR)

The SPR bit BUSPOSACKERR indicates that an expected positive acknowledge was not received for an address bus operation and is always set by this condition independent of the BUSPOSACKEN SPR bit. A positive acknowledge will be expected independent of the BUSPOSACKEN SPR bit. BUSPOSACKERR is deasserted by any write to BUSCSR SPR. The hardware reset state is deasserted. Section 8.4.10, "Address Status Acknowledge."

8.21.7 Bus Positive Acknowledge Error Enable (BUSPOSACKEN)

The SPR bit BUSPOSACKEN enables the positive acknowledge error condition to cause a machine-check exception. This SPR bit does not effect the setting of BUSPOSACKERR. The hardware reset state for BUSPOSACKEN is deasserted. In addition, when BUSPOSACKEN is deasserted bus operations that support positive acknowledge (refer to Table 8-13) will treat NoAck like a PosAck.

8.22 Deadlock Scenarios and Solutions

The following sections provide information on deadlock scenarios and solutions.

8.22.1 The Queue Full/Ping-Pong Deadlock

A bus master issues a bus operation to two bus devices, 1 and 2. The bus operation takes longer to execute than it takes for a master to rearbitrate for the bus and issue the same operation over again. Each time the master issues the bus operation it is retried by 1, then

2, then 1, and so on. Each bus device has executed the bus operation at least once, but the master thinks, or assumes, that it has not been executed once by all bus devices.

Figure 8-37 illustrates the deadlock.
Bus device 1 is busy for some other reason and retries the first address.
Bus device 2 accepts the bus operation and starts to execute the operation.
Bus device 2 is busy with the first address and retries the second address.
Bus device 1 accepts the second address and starts to execute the operation.
Bus device 1 is busy with the second address and retries the third address.
Bus device 2 accepts the third address and starts to execute the operation.
Repeat this pattern.

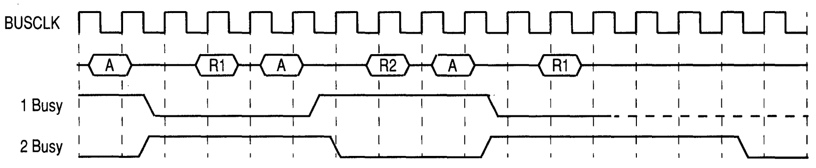


Figure 8-37. Queue Full/Ping-Pong Deadlock

A bus operation that gets AStat Retried will be aborted 1 cycle from sampling ASTATIN Retry. Aborted means that the ASTATIN Retried bus operation will not cause any subsequent bus operations to be ASTATOUT or ARESPOUT Retried due to a queue full condition.

Related Text:

- Section 8.4.17, “Address Status Out and Address Response Out Retry”
- Section 8.4.18, “ASTATIN and ARESPIN Retry”

8.22.2 The Non-Pended Push Deadlock

A bus device adapts a non-pended bus to the 620 bus. The non-pended bus starts an operation that locks up the non-pended bus until the operation is complete. The non-pended bus operation causes a snoop push by the 620 and there is no push buffer available for the push and all push buffers contain operations directed towards the non-pended bus.

- Deadlock prevention by the system—Non-Pended memory space can not be marked write-back and cached in the modified state. See Section •, “Non-Pended Bus Adapter—A non-pended bus is a bus that does not support retry as a mechanism to allow a higher priority operation to hold off a lower priority operation until the higher priority operation can get onto the bus. The following guidelines must be followed when a bus adapter connects a non-pended bus to the 620 bus..”

- Deadlock prevention by the 620:
  - — There is 1 store buffer set aside for bus cast-outs (pushes) that can not be used for stores and processor cast-outs.
  - — Internal operations will be reprioritized when a bus operation is retried.
  - — Internal arbitration will guarantee that a bus push will be issued to the bus eventually, even if other pushes continually get retried. See Section 8.3.4, "Internal Request Arbitration."

8.22.3 The Previous Adjacent Address Match Deadlock

N bus devices attempt to issue operations to the bus. Each of the devices collide. Each device responds to the detection of the previous adjacent address match by retrying itself and reissuing the same bus operation. Every time each bus device issues it's bus operation it collides with the operation of the previous bus device.

Refer to the solution described by Section 8.18.3.2, "Rule 2: CD Previous Adjacent ASTATOUT Retry."

8.23 Hardware Configuration Mechanism (HCM)

The Hardware Configuration Mechanism (HCM) provides external configuration information to the 620 during hardware reset in order for the 620 to be able to start fetching instructions.

There are two types of pins used by the HCM—LSSD and HCM-Only. LSSD pins function as HCM inputs when the TestInhibit signal is deasserted (LSSD disabled). LSSD pins are only used during chip manufacturing. The HCM-Only pins are always defined to be HCM inputs and should be weakly pulled or driven to the desired value.

All pins used by the HCM should have valid HCM values during the assertion of $\overline{\text{HRESET}}$ and should be held valid for three BUSCLKs after $\overline{\text{HRESET}}$ is deasserted.

Table 8-56. Hardware Configuration Variables

Pin Type	Signal name	Hardware Configuration Variable
HCM Only	BUSCLKGTL	BUSCLKGTL
LSSD/HCM	SHIFTGATE	BUSRATIO[0]
HCM Only	BUSRATIO1	BUSRATIO[1]
LSSD/HCM	L1_TESTCLK	PLLPUMPLOW
	L2_TESTCLK	PLLVCODIV[0]
	RCVRINHIBIT	PLLVCODIV[1]

Table 8-56. Hardware Configuration Variables (Continued)

HCM Only	BUSRESPTEN0	BUSRESPTEN[0]
	BUSRESPTEN1	BUSRESPTEN[1]
	BUSDX	BUSDX
	BUSPID0	BUSPID[0]
	BUSPID1	BUSPID[1]
	BUSPID2	BUSPID[2]
LSSD/HCM	DRVRINHIBIT1	BUSPID[3]
	DRVRINHIBIT2	BUSPID[4]

8.24 Debug Support

The 620 implements the following facility to support system debugging:

- Instruction Tracing—The 620 supports instruction tracing as described in *PowerPC Microprocessor Family: The Programming Environments*, and Section 4.6.11, “Trace Exception (0x00D00).”
- Data Address Breakpoint—The 620 supports data address breakpoint as described in Section 2.1.1, “Register Set.”
- Instruction Address Breakpoint—The 620 supports instruction address breakpoint as described in Section 2.1.1, “Register Set.” The IABR is also incorporated in the ESP debug support. An ESP breakpoint command can be used to initialize the IABR and set-up the 620 such that if an address compares, no exception will be taken, but 620 will stop the processor clock and allow the ESP to access the processor’s internal states. The clock will resume running when ESP issues a resume command.
- Single-Instruction Mode—When HID0[24]=0, 620 will run in the single-instruction mode. The 620 will dispatch, execute and complete one instruction at a time, see also Section 2.1.2.3, “Hardware Implementation-Dependent Register 0 (HID0).” HID0[24] is also incorporated in the ESP debug. The ESP instruction step command will cause 620 to stop the internal clock every time it completes a valid instruction, and allow the ESP to access processor’s internal states.
- Processor Internal Watchdog—The 620 implements a watchdog facility which will force the 620 into Checkstop state if 620 does not complete any valid instructions during the period of time that the Decrementer passes through zero twice. The Decrementer is enabled when the TBEnable_ signal is asserted and it decrements every BUSCLK. If the bus frequency is 66MHz, then it takes at least 65 seconds for the 32-bit Decrementer to pass through zero twice. The watchdog facility is disabled when HID0(14)=1.

8.25 Hard Reset

The hard reset sequence begins when the hard reset signal $\overline{\text{HRESET}}$ is asserted. While $\overline{\text{HRESET}}$ is asserted, the internal phase-lock-loop (PLL) will synchronize the internal processor clock with the system bus clock based upon the PLL bandwidth control signals described in Section 7.2.13.4.1, "Hard Reset (HRESET)—Input," and the bus ratio specified on the hardware configuration pins as described in Section 8.23, "Hardware Configuration Mechanism (HCM)." At the same time, the 620 will automatically reset the internal registers through the built-in scan chains. The whole process including PLL synchronization should take no more that 2ms. See Section 7.2.13.4.1, "Hard Reset (HRESET)—Input," for more detailed timing requirements. Once the $\overline{\text{HRESET}}$ is deasserted, the 620 will start its functional clock and fetch the first instruction from address 0x00000000FFF00100 in the system reset exception vector.

While $\overline{\text{HRESET}}$ is asserted, all processor and L2 interface I/O's are set in the tristated mode except for L2Clock and $\overline{\text{L2CLOCK}}$, and the system interface signals are ignored. All pins used by the HCM should have valid HCM values during the assertion of $\overline{\text{HRESET}}$ and should be held valid for at least three BUSCLKs after $\overline{\text{HRESET}}$ is deasserted. The JTAG Reset (JTAG_TRST) pin should be held asserted, and all LSSD Test input signals should be deasserted while $\overline{\text{HRESET}}$ is asserted.

Table 8-57 shows the state of the architectural registers and major arrays after the hard reset and before the first instruction is fetched. Most arrays are not initialized at the end of hard reset, but they can be initialized through the Array-Built-In-Self-Test (ABIST) mechanism which can be initiated with JTAG commands.

The following is also true after a hard reset operation:

- MSR is initialized with all 0's except IP bit is set which means 620 will operate with
  - — both instruction and data translation disabled, and in
  - — 32-bit, and big-endian mode,
- HID0 is initialized with all 0's which means 620 will operate with
  - — both instruction and data caches disabled, and parity checking disabled,
  - — internal watchdog enabled,
  - — superscalar mode disabled,
  - — static branch prediction without update of branch history table, and
  - — speculative instruction fetch from the memory disabled.
- External checkstop is enabled.
- Processor interface configuration is set through hardware configuration mechanism as described in Section 8.23, "Hardware Configuration Mechanism (HCM)."
- Processor and L2 interface I/O's operate in functional mode.

Table 8-57. Register/Array Settings after Hard Reset

Register	Width	Setting	Array	Setting
FPSCR	32	All 0s	GPR	undefined
CR	32	All 0s	FPR	undefined
LR	64	All 0s	I Cache	invalidated
CTR	64	FFFFFFFFFFFFFFFF	D Cache	invalidated
XER	32	All 0s	TLB	undefined
MSR	64	0000000000000040 (IP set)	SLB	undefined
SRR1	64	0000000000000040	I BAT's	undefined
SRR0	64	All 0s	D BAT's	undefined
DEC	32	All 0s	I ERAT	invalidated
SPRG0	64	All 0s	D ERAT	invalidated
SPRG1	64	All 0s	BHT	undefined
SPRG2	64	All 0s	BTAC	undefined
SPRG3	64	All 0s		
TimeBase	64	All 0s		
PVR	32	0014tMrm t: technology indicator M: major revision r: reserved m: minor revision		
DABR	64	All 0s		
IABR	64	All 0s		
ASR	64	All 0s		
SDR1	64	All 0s		
DSISR	32	All 0s		
HID0	32	00000102		
EAR	32	All 0s		
L2CR	64	0000000000000215		
L2SR	64	All 0s except bits (35–39) undefined		
BUSCSR	64	All 0s except bits (40–42, 44–47) sampled from pins		
PIR	32	All 0s except bits (59–63) sampled from pins		
PMC1	32	All 0s		
PMC2	32	All 0s		

Table 8-57. Register/Array Settings after Hard Reset (Continued)

Register	Width	Setting	Array	Setting
MMCR0	32	All 0s		
MMCR1	32	All 0s		
SIA	64	All 0s		
SDA	64	All 0s		
PMC1	32	5C		
PMC2	32	3		
PMC3	32	All 0s		
PMC4	32	2		
PMC5	32	All 0s		
PMC6	32	All 0s		
PMC7	32	2		
PMC8	32	All 0s		

Chapter 9
Secondary Cache Interface

The PowerPC 620 microprocessor implements an in-line secondary (L2) cache (that is, an integrated L2 cache controller and external tag and data SRAMs). This chapter provides information on the L2 cache interface operation, register set, and ECC errors, as well as providing timing diagrams.

9.1 Overview

The L2 cache interface includes the following features:

- A unified instruction and data cache
- Direct-mapped
- Data and tag memories are accessed with the same address
- The block size, which is the same as the sector size, is 64 bytes
- The L2 may be disabled
- Supports L2 cache configurations from 1 Mbyte to 512 Mbytes
- Supports single-, double-, triple-, and quad-register synchronous SRAMs
- Provides no support for asynchronous SRAMs
- Accommodates a wide range of access time SRAMs with a sub-synchronous interface configured at power-on
- Provides the following ECC protection:
  - A 9-bit ECC code is generated over 128 bits of data
  - A 6-bit ECC code is generated over the tag and coherency state
  - ECC codes are single-bit correcting and double-bit detecting
  - The L2 interface generates a machine check exception for uncorrectable errors
  - ECC operates in the following three modes:
    - Always corrected mode—ECC is generated for writes and always corrected for reads. This provides constant L2 read access latency.
    - Never corrected mode—ECC generation, checking, and correction are disabled. The 620 ECC inputs are disconnected; outputs are not defined.
    - Automatic switch mode—ECC is generated for writes and corrected for reads only when an error is detected. L2 read access latency is a function of when correctable errors are detected.

For more information on the ECC refer to Section 9.4, "ECC—L2 Error Detection and Correction."

9.2 L2 Cache Interface Operation

The following sections describe L2 cache operations.

9.2.1 L2 SRAM Connection Diagram

Figure 9-1 shows how SRAMs are connected to the L2 interface. Though only two SRAMs are shown, the blocks could be composed of multiple SRAMs.

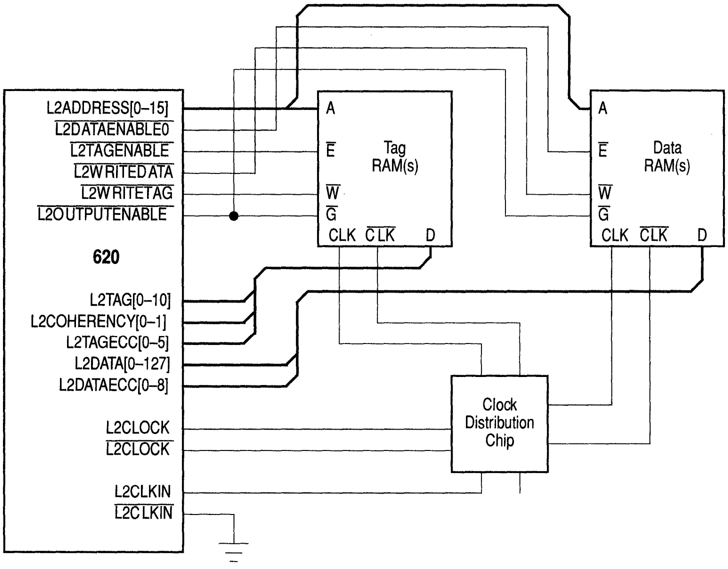

L2ADDRESS[0–15]
$\overline{\text{L2DATAENABLE0}}$
$\overline{\text{L2TAGENABLE}}$
$\overline{\text{L2WRITEDATA}}$
$\overline{\text{L2WRITETAG}}$
$\overline{\text{L2OUTPUTENABLE}}$
620
L2TAG[0–10]
L2COHERENCY[0–1]
L2TAGECC[0–5]
L2DATA[0–127]
L2DATAECC[0–8]
L2CLOCK
$\overline{\text{L2CLOCK}}$
L2CLKIN
$\overline{\text{L2CLKIN}}$
A
$\overline{\text{E}}$
Tag RAM(s)
$\overline{\text{W}}$
$\overline{\text{G}}$
CLK $\overline{\text{CLK}}$ D
A
$\overline{\text{E}}$
Data RAM(s)
$\overline{\text{W}}$
$\overline{\text{G}}$
CLK $\overline{\text{CLK}}$ D
Clock Distribution Chip

Figure 9-1. L2 Connection Diagram

9.2.2 Description of L2 Interface Digital PLL

To eliminate clock latency on the L2 data bus, a digital PLL is utilized to guarantee that the SRAM clocks are phase aligned to the internal processor clock. The output of the PLL is the differential CMOS pair of L2CLOCK and $\overline{\text{L2CLOCK}}$, as shown in Figure 9-1. The L2CLOCK pair (or single ended CMOS output when selected) must be fed into a fanout clock buffer, with one output (pair) fed back to the L2CLOCKIN and $\overline{\text{L2CLOCKIN}}$ inputs. This closes and completes the PLL feedback loop. As long as the length of the feedback output of the clock buffer is equal to the length of the buffer output traces to the SRAM clock inputs, the SRAM clocks will be phase aligned to the internal 620 processor clock, refer to Section 9.3.1.15, "L2PLLEN Bit."

9.2.3 L2 Direct Connectivity to SRAMs

The 620 interface supports direct connectivity (without external support) for up to two banks of SRAMs (Banks 0 and 1). The 620 will insert a high impedance L2CLOCK cycle to avoid bus contention for the following general conditions on the L2 data bus:

- When a write follows a read
- When a read follows a write
- For other conditions dead cycle injection depends on whether the SRAMs are enabled and the value of the L2 deadcycle bit in the L2CR register. For complete details when dead cycles are injected see Section 9.3.1.2, "L2NORWDEAD Bit."

Note that this mechanism can increase the maximum snoop latency through the cache by one L2CLOCK (L2RATIOSR). Section 8.4.5, "Address to ARESPOUT Latency (BUSTLAR[0–2])."

9.3 L2 Interface Register Set

The following sections provides information on the L2 interface register set. For more information refer to Chapter 2, "Programming Model."

9.3.1 L2 Configuration Register (L2CR)

The following section describes the usage of the L2CR bits. Refer to Section 2.1.2.6, "L2 Cache Status Register (L2SR)." The coherency block size of the L2 cache is fixed at 64 bytes.

9.3.1.1 Programming Restrictions

L2NORWDEAD=0; L2LATEWRITE=1; L2SINGSYNC=1

Incorrect operation may result if these four SPR bits are programmed as specified. All other possible combinations of these four bits are not restricted.

9.3.1.2 L2NORWDEAD Bit

When L2NORWDEAD (bit 37) is asserted dead cycles are not inserted between read and write cycles on the L2 interface. If you have determined that there would be no detrimental effect caused by having two buffers enabled on a net at a given time, and you can handle the added timing of having a signal take two transit times to resolve instead of one, you may choose to eliminate the injection of dead cycles between read and write cycles. This increases the throughput of the L2 interface and should positively affect performance.

Table 9-1 shows the definition of L2NORWDEAD.

Table 9-1. L2NORWDEAD Bit

L2NORWDEAD	Meaning
0	Dead cycles injected between read and writes.
1	Dead cycles not injected between read and writes.

9.3.1.3 L2SIZE[0–3] Bits

The L2SIZE[0–3] bits (L2CR[40–43]) are the cache capacity and organization bits; refer to Table 9-2. Note that values of L2SIZE not shown in Table 9-2 are reserved. Refer to Section 2.1.2.6, "L2 Cache Status Register (L2SR)," for more information.

Table 9-2. L2SIZE[0–3] Bits

L2SIZE[0–3]	L2 Cache Capacity
0000	1 MB
0001	2 MB
0010	4 MB
0011	8 MB
0100	16 MB
0101	32 MB
0110	64 MB
0111	128 MB
1000	256 MB
1001	512 MB

9.3.1.3.1 The L2TAGADD Signal

The width of the L2 cache tag decreases as the L2 cache capacity increases. The width of the L2 cache index increases as the L2 cache capacity increases. A minimal interface signal implementation is achieved by changing the definition of L2TAGADD[0–8] based on the L2 cache capacity, as shown by Table 9-3 and Table 9-4. Table 9-3 describes a single bank cache. Refer to Section 7.2.6.8, "L2 Enable Signals." for additional information about L2 signals.

Table 9-3. L2 Cache Capacity and Tag/Index Signal Definition—Single Bank

Total Capacity	Tag Size	Tag Index Size	Data Index Size	Tag: [L2TAG]|| [L2TAGADD]	Tag Index: [L2TAGADD]|| [L2Address]	Data Index: [L2TAGADD]|| [L2ADDRESS]
1 Mbytes	20 bits	14 bits	16 bits	[0–10]||[0–8]	[0–13]	[0–15]
2 Mbytes	19 bits	15 bits	17 bits	[0–10]||[0–7]	[8]||[0–13]	[8]||[0–15]
4 Mbytes	18 bits	16 bits	18 bits	[0–10]||[0–6]	[7–8]||[0–13]	[7–8]||[0–15]
8 Mbytes	17 bits	17 bits	19 bits	[0–10]||[0–5]	[6–8]||[0–13]	[6–8]||[0–15]
16 Mbytes	16 bits	18 bits	20 bits	[0–10]||[0–4]	[5–8]||[0–13]	[5–8]||[0–15]
32 Mbytes	15 bits	19 bits	21 bits	[0–10]||[0–3]	[4–8]||[0–13]	[4–8]||[0–15]
64 Mbytes	14 bits	20 bits	22 bits	[0–10]||[0–2]	[3–8]||[0–13]	[3–8]||[0–15]
128 Mbytes	13 bits	21 bits	23 bits	[0–10]||[0–1]	[2–8]||[0–13]	[2–8]||[0–15]

Table 9-4 describes a double bank cache.

Table 9-4. L2 Cache Capacity and Tag/Index Signal Definition—Double Bank

Total Capacity	Tag Size	Tag Index Size	Data Index Size	Tag: [L2TAG]|| [L2TAGADD]	Tag Index: [L2TAGADD]|| [L2ADDRESS]	Data Index: [L2TAGADD]|| [L2ADDRESS]
1 MBytes	20 bits	14 bits	15 bits	[0–10]||[0–8]	[0–13]	[1–15]
2 M Bytes	19 bits	15 bits	16 bits	[0–10]||[0–7]	[8]||[0–13]	[8]||[1–15]
4 MBytes	18 bits	16 bits	17 bits	[0–10]||[0–6]	[7–8]||[0–13]	[7–8]||[1–15]
8 MBytes	17 bits	17 bits	18 bits	[0–10]||[0–5]	[6–8]||[0–13]	[6–8]||[1–15]
16 MBytes	16 bits	18 bits	19 bits	[0–10]||[0–4]	[5–8]||[0–13]	[5–8]||[1–15]
32 MBytes	15 bits	19 bits	20 bits	[0–10]||[0–3]	[4–8]||[0–13]	[4–8]||[1–15]
64 MBytes	14 bits	20 bits	21 bits	[0–10]||[0–2]	[3–8]||[0–13]	[3–8]||[1–15]
128 MBytes	13 bits	21 bits	22 bits	[0–10]||[0–1]	[2–8]||[0–13]	[2–8]||[1–15]

9.3.1.3.2 L2 Cache Organization Examples

The 620 provides direct connect support for SRAM devices. The issue rate of some SRAM configurations may be less than other configurations because of address loading. Address repowering may be necessary depending on the configuration.

Some example configurations of the data and tag portions of the L2 are listed in Table 9-5 and Table 9-5. The L2 tag store only needs to be 1/4 of the depth of the L2 data store. A configuration is acceptable if it meets the functional requirements for the configured capacity and operates at the proper L2 cycle time.

The tag memory will vary based on the cache capacity and the availability of less dense and wide SRAMS. (For availability reasons the tag SRAMs may be the same as the data memory SRAMS, and waste all but 1/nth where n is the block size in quad words.)

Table 9-5. Cache Capacity and Organization for Data Portion of L2 Cache

Total Capacity	Total Devices	SRAM Density	SRAM Organization	Overall Organization
1 MByte	8	1 Mbit	64 Kbits x 18 bits	8 wide x 1 deep
2 MBytes	16	1 Mbit	64 Kbits x 18 bits	8 wide x 2 deep
2 MBytes	16	1 Mbit	128 Kbits x 9 bits	16 wide x 1 deep
4 MBytes	8	4 Mbits	256 Kbits x 18 bits	8 wide x 1 deep
8 MBytes	16	4 Mbits	256 Kbits x 18 bits	8 wide x 2 deep
16 MBytes	8	16 Mbits	1 Mbit x 18 bits	8 wide x 1 deep
32 MBytes	16	16 Mbits	1 Mbit x 18 bits	8 wide x 2 deep

Table 9-6. Cache Capacity and Organization for Tag Portion of L2 Cache

Total Capacity	Total Devices	SRAM Density	SRAM Organization	Overall Organization
1 MByte	2	1 Mbit	64 Kbits x 18 bits	2 wide x 1 deep
2 MBytes	2	1 Mbit	64 Kbits x 18 bits	2 wide x 1 deep
2 MBytes	4	1 Mbit	128 Kbits x 9 bits	4 wide x 1 deep
4 MBytes	2	4 Mbits	256 Kbits x 18 bits	2 wide x 1 deep
8 MBytes	2	4 Mbits	256 Kbits x 18 bits	2 wide x 1 deep
16 MBytes	2	16 Mbits	1 Mbit x 18 bits	2 wide x 1 deep
32 MBytes	2	16 Mbits	1 Mbit x 18 bits	2 wide x 1 deep

9.3.1.4 L2RATIOSR Bit

L2RATIOSR (L2CR[60–61]) indicates the frequency L2CLOCK in terms of processor clocks (PCLKs).

If the L2 is disabled L2RATIORP must have its value set to the default 01. Refer to Section 2.1.2.6, "L2 Cache Status Register (L2SR)."

L2RATIOSR is defined in Table 9-7:

Table 9-7. L2RATIOSR Code Definition

L2RATIONSR[0–1]	Definition	Unit
00	Reserved	PCLK
01	1	PCLK
10	2	PCLK
11	3	PCLK

Example of how to set L2RATIO:

If the processor clock is running in 2:1 mode with respect to the L2 interface, then the value is set to two PCLKs. Likewise, if the ratio is 1:1 for PCLKs to L2, then the value is set to one PCLK (default mode).

9.3.1.5 L2ECCMODE[0–1] Bits

The L2ECCMODE[0–1] (L2CR[52:53]) are the ECC mode select bits. Table 9-8 defines the SPR field that specifies the L2 cache ECC mode. Refer to Section 2.1.2.6, "L2 Cache Status Register (L2SR)."

Table 9-8. L2ECCMODE[0–1] Code Definition

L2ECCMODE[0–1]	L2 ECC Mode
00	Never correct (ECC disabled)
01	Always correct
10	Automatic switch correct
11	Reserved

9.3.1.6 L2CLKPECL Bit

The L2CLKPECL (L2CR[36]) is the PECL enable bit. Table 9-9 provides the definition of L2CLKPECL.

Table 9-9. L2PECLEN Code Definition

L2CLKPECL	Logic Type
0	CMOS
1	PECL

When L2CLKPECL is asserted L2CLOCKIN and $\overline{\text{L2CLOCKIN}}$ go into PECL mode regardless of the state of L2HSTLEN. L2CLKPECL does not affect the logic levels for L2CLOCK and $\overline{\text{L2CLOCK}}$. When the output buffers are in CMOS mode, a resistor divider network can be used to generate the proper PECL levels.

9.3.1.7 L2B2ENABLE Bit

L2B2ENABLE (L2CR[56]) is the double-bank enable bit. When asserted high, it enables the internal decode for a dual SRAM bank L2 cache. See Section 7.2.6.8, “L2 Enable Signals.”

9.3.1.8 L2CLC[0–1] Bits

The 620 supports 7 multi-level cache configurations, as specified by the HID0 and L2CR SPR configuration fields IL1 and DL1 and L2CLC[0–1] (L2CR[47–48]). HID0[16] is the IL1 cache enable. HID0[17] is the DL1 cache enable. L2CLC[0] is the L2 cache enable. L2CLC[1] is the L3 cache enable. See Table 4-2 for the vector offset values. Also, refer to Section 2.1.2.6, “L2 Cache Status Register (L2SR),” and Section 8.17.8, “Multi-Level Cache Definition.” L2CLC[0–1] is defined as follows. Any configurations not listed in Table 9-10 are not supported by the 620.

Table 9-10. Multi-Level Cache Configuration

Num	HID0[16–17]	L2CLC[0–1]	Enabled			
			IL1	DL1	L2	L3
1	00	00	N	N	N	N
2	01	00	N	Y	N	N
3	10	00	Y	N	N	N
4	11	00	Y	Y	N	N
5	11	01	Y	Y	N	Y
6	11	10	Y	Y	Y	N
7	11	11	Y	Y	Y	Y

9.3.1.8.1 L2CLC = 00 ($\overline{\text{L2}}$, $\overline{\text{L3}}$)

If either the IL1 or DL1 are disabled then snooping must be disabled (BUSSNPEN=0) and both the L2 and L3 must be disabled. If both the IL1 and DL1 are enabled then snooping can be enabled (BUSSNPEN=1) and the L2 and L3 may be left disabled. The DL1 will snoop all memory coherent bus operations directly. The DL1 and IL1 is the lowest cache level in the system.

9.3.1.8.2 L2CLC = 01 (L1, $\overline{\text{L2}}$, L3)

There is no L2, but there is an L3 external to the 620. The DL1 will snoop all memory coherent bus operations directly. One application for this configuration is when a bus adapter adapts the 620 to a 60x bus and needs to snoop the 60x bus and maintain coherence with the DL1.

9.3.1.8.3 L2CLC = 10 (L1, L2, $\overline{\text{L3}}$)

The L1 and the L2 are both between the processor and the system bus and there is no cache external to the 620. The L2 will snoop all memory coherent bus operations. The L2 is the lowest cache level in the system.

9.3.1.8.4 L2CLC = 11 (L1, L2, L3)

The L1 and the L2 are both between the processor and the system bus. There is an L3 external to the 620. The L2 will snoop all memory coherent bus operations.

9.3.1.9 L2LATEWRITE Bit

L2LATEWRITE (L2CR[51]) is the late/normal write select bit. Table 9-11 indicates the definition of L2LATEWRITE.

Table 9-11. L2LATEWRITE Code Definition

L2LATEWRITE	Mode
0	Disabled
1	Enabled

When L2LATEWRITE is enabled, the write data is put on the data lines a cycle after the control and address for that data has been applied to the SRAM inputs. This saves a dead cycle on the data lines when going from writes to reads.

When L2LATEWRITE is disabled, the write data is put on the data lines the same cycle as the control and address for that data has been applied to the SRAM inputs.

Refer to Section 2.1.2.6, “L2 Cache Status Register (L2SR).” For detailed diagrams, refer to Section 9.5, “L2 Cache Timing Diagrams.”

9.3.1.10 L2SINGSYNC Bit

L2SINGSYNC (L2CR[54]) is the L2 SRAM register depth select bit. Table 9-12 indicates the definition of L2SINGSYNC.

Table 9-12. L2SINGSYNC Code Definition

L2SINGSYNC	Mode
0	Two Registers
1	One Register

L2SINGSYNC is used to determine the number of pipeline registers that are in the SRAMs used in the L2.

Refer to Section 2.1.2.6, “L2 Cache Status Register (L2SR).” For detailed diagrams, refer to Section 9.5, “L2 Cache Timing Diagrams.”

9.3.1.11 L2WRCNTRDIS Bit

L2WRCNTRDIS (L2CR[45]) is the L2 write counter disable.

Table 9-13. L2WRCNTRDIS Code Definition

L2WRCNTRDIS	Counter Mode
0	Enabled
1	Disabled

The L2 interface transitions between read mode and write mode. During a read access sequence it is possible for an L2 write to be held waiting for a transition to write mode. The L2 write counter is used to track the number of processor clocks the pending L2 write is held waiting for an L2 cache write opportunity. If L2WRCNTRDIS is enabled (the default condition) and the L2 write counter reaches a count of 32, the L2 interface is forced into write mode to allow the pending L2 write operation to proceed. If the L2WRCNTRDIS bit is set the L2 write counter is disabled.

9.3.1.12 $\overline{\text{L2INIT}}$ Bit

$\overline{\text{L2INIT}}$ (L2CR[50]) is the L2 cache initialize enable bit. Table 9-14 indicates the definition of $\overline{\text{L2INIT}}$.

Table 9-14. $\overline{\text{L2INIT}}$ Code Definition

$\overline{\text{L2INIT}}$	Mode
0	Enabled
1	Disabled

When the $\overline{\text{L2INIT}}$ is enabled it indicates that the L2 SRAMs are being initialized. When in this mode, if the L2 is read it always returns a MESI state of invalid, and ECC is forced to pass. Refer to Section 2.1.2.6, “L2 Cache Status Register (L2SR).”

9.3.1.13 L2DPWR[0–1] Bit

L2DPWR[0–1] (L2CR[44–45]) is the point-to-point driver strength select bit. Table 9-15 indicates the definition of L2DPWR.

Table 9-15. L2DPWR Code Definition

L2DPWR[0–1]	Mode
0	L2 point-to-point drivers in 50 Ohm mode
1	L2 point-to-point drivers in 75 Ohm mode

L2DPWR is used to define the power of the point-to-point drivers used in the L2 interface. Refer to Section 2.1.2.6, “L2 Cache Status Register (L2SR).”

9.3.1.14 L2ECCERREN Bit

L2ECCERREN (L2CR[46]) is the ECC error enable bit. A multi-bit ECC error will cause a machine-check interrupt if L2ECCERREN is asserted. Refer to Section 2.1.2.6, "L2 Cache Status Register (L2SR)." The hardware reset state for L2ECCERREN is deasserted.

9.3.1.15 L2PLLEN Bit

L2PLLEN (L2CR[49]), the PLL enable bit, enables the L2 PLL to lock. The initial and hard reset state is deasserted. See Section 2.1.2.5, "L2 Cache Control Register (L2CR)," for how to program L2PLLEN.

9.3.2 L2 Cache Status Register (L2SR)

The following section describes the usage of the L2SR bits. Refer to Section 2.1.2.6, "L2 Cache Status Register (L2SR)."

9.3.2.1 L2ECC Bit

The L2ECC (L2SR[16]) is the error detected bit. This bit will be set when the first single or double bit error is detected and will remain set until the L2SR is written.

9.3.2.2 L2DATASYN[0–8] and L2TAGSYN[0–5] Bits

L2DATASYN[0–8] (L2SR [17–25]) and L2TAGSYN[0–5] (L2SR[26–31] are the L2 data and tag syndromes bits. A description of each of these follows:

- L2DATASYN is the syndrome for the L2 data.
- L2TAGSYN is the syndrome for the L2 tag.
- The syndromes are for the first detected ECC error.
- The reset state is undefined.
- The syndromes are cleared to all 0s for any write to the L2SR SPR.
- Data can not be written into the syndromes.

Bit 0 of the syndrome field reflects parity generated across the syndrome such that odd parity across the field indicates the single bit errors that were corrected, while even parity indicates a double bit error was detected but no specific information may be ascertained. If all bits are 0 no error is indicated. Some odd parity combinations are unused because more combinations were available than ECC or data to check; these patterns are invalid because they have no meaning. In addition, multiple bit errors (>2) may look like single or double bit errors in the syndrome or may appear as invalid combinations. ECC bits are interspersed with the rest of the bits being checked at powers of two and a modified Hamming code algorithm is used to generate the syndrome.

9.3.2.3 Tag Syndrome

Should a correctable ECC error occur on the L2COHERENCY[0–1], L2TAG[0–10], L2TAGADD[0–8], or the L2TAGECC[0–5] bits, the L2TAGSYN[0–5] bits of the L2SR will indicate the corrected single bit. Table 9-16 shows each syndrome and the associated bit that was in error.

Table 9-16. L2 Tag Syndrome Bits

L2TAGSYN[0–5]	Bit in Error
000000	No Error
100000	L2TAGECC[0]
000001	L2TAGECC[5]
000010	L2TAGECC[4]
100011	L2COHERENCY[0]
000100	L2TAGECC[3]
100101	L2COHERENCY[1]
100110	L2TAG[0]
000111	L2TAG[1]
001000	L2TAGECC[2]
101001	L2TAG[2]
101010	L2TAG[3]
001011	L2TAG[4]
101100	L2TAG[5]
001101	L2TAG[6]
001110	L2TAG[7]
101111	L2TAG[8]
010000	L2TAGECC[1]
110001	L2TAG[9]
110010	L2TAG[10]
010011	L2TAGADD[0]
110100	L2TAGADD[1]
010101	L2TAGADD[2]
010110	L2TAGADD[3]
110111	L2TAGADD[4]
111000	L2TAGADD[5]
011001	L2TAGADD[6]
011010	L2TAGADD[7]
111011	L2TAGADD[8]
All Other Combinations	Detectable Uncorrectable

C code used to generate the tag syndrome table:

```
/**********************************************************************/

/* Generates a table of the TagECC syndromes in binary for PPC 620 */


/***********************************************************************/

#include <stdio.h>
main () {
int g,i,j=0;

/* assign data to symbol positions */
printf("L2SR<17:25> -- L2TAGECC<0:8> Syndrome\n\n");
printf("Single Bit ECC Failure Indication Only\n\n");
printf(" TagECC Syndrome\tBit in Error\n",i);
printf("----------------\t------------\n",i);
printf("  000000\t\t  NONE\n");
printf("  100000\t\t  TagECC<0>\n",i,j);

/* The Tag symbol is composed of 22 info + 6 ECC bits = 28 bits */
/* 5 ECC bits are derived from the L2 info bits and the 6th is parity for */
/* the entire symbol.  In single bit errors, which is all this ECC code */
/* will detect, this most significant ECC bit also happens to yield an */
/* odd parity for the syndrome.  The L2 info consists of two Coherency */
/* bits, eleven Tag bits, and nine TAGADD bits. */

for(i=1;i<28;i++) {

/* Special cases: ECC bits are interspersed at powers of 2 */
  if (i == 0x01 || i == 0x02 || i == 0x04 || i == 0x08 ||
      i == 0x10 || i == 0x20 || i == 0x40 || i == 0x80) {
    switch (i)
      {
      case 0x01 :
        printf("  000001\t\t  TagECC<5>\n");
         break;
       case 0x02 :
        printf("  000010\t\t  TagECC<4>\n");
         break;
       case 0x04 :
        printf("  000100\t\t  TagECC<3>\n");
         break;
       case 0x08 :
        printf("  001000\t\t  TagECC<2>\n");
         break;
       case 0x10 :
        printf("  010000\t\t  TagECC<1>\n");
         break;
         break;
       default :
        printf("%x\t\tunknown) \n");
       }
  }
  else {
   /* Figure out what the ECC parity bit for this combination should be */
        g = giveparity(i);
    printf("  ");
    printbits(g);
    if (j-2 < 0) printf("\t\t  Coherency<%d>\n",j);
    else if (j-2 < 11) printf("\t\t  Tag<%d>\n",j-2);
         else printf("\t\t  TAGADD<%d>\n",j-13);
    j++;
  }
}
printf("   Other\t\t INVALID\n\n\n",i,j);
}

int printbits(int num) {
/* This function prints the binary sequence for a 6-bit number */
/* as ASCII characters to <STDOUT> */

  int lcv, mask=0x20;

  for (lcv=0;lcv<6;lcv++) {
    ((mask & num) == 0) ? printf("0") : printf("1");
    mask >>= 1;
  }
}

int giveparity(int num) {
/* Returns a 6 bit number with odd parity (MSB) given a 5 bit number */

  int lcv, par=0, mask=1;

  for (lcv=0;lcv<6;lcv++) {
    if ((mask & num) != 0) par++;
    mask <<= 1;
  }
  if ((par % 2) == 1)
       return num;
  else return num | 0x20;
}
```

9.3.2.4 Data Syndrome

Should a correctable ECC error occur on L2DATA[0–127] or L2DATAECC[0–8], the L2DATASYN[0–8] bits of the L2SR will indicate the single bit that was in error. Table 9-17 shows each syndrome and the associated corrected bit.

Table 9-17. L2 Data Syndrome Bits

L2DATASYN[0–8]	Bit in Error
000000000	No Error
100000000	DATAECC[0]
000000001	DATAECC[8]
000000010	DATAECC[7]
100000011	DATA[0]
000000100	DATAECC[6]
100000101	DATA[1]
100000110	DATA[2]
000000111	DATA[3]
000001000	DATAECC[5]
100001001	DATA[4]
100001010	DATA[5]
000001011	DATA[6]
100001100	DATA[7]
000001101	DATA[8]
000001110	DATA[9]
100001111	DATA[10]
000010000	DATAECC[4]
100010001	DATA[11]
100010010	DATA[12]
000010011	DATA[13]
100010100	DATA[14]
000010101	DATA[15]
000010110	DATA[16]
100010111	DATA[17]
100011000	DATA[18]
000011001	DATA[19]
000011010	DATA[20]
100011011	DATA[21]

Table 9-17. L2 Data Syndrome Bits (Continued)

L2DATASYN[0–8]	Bit in Error
000011100	DATA[22]
100011101	DATA[23]
100011110	DATA[24]
000011111	DATA[25]
000100000	DATAECC[3]
100100001	DATA[26]
100100010	DATA[27]
000100011	DATA[28]
100100100	DATA[29]
000100101	DATA[30]
000100110	DATA[31]
100100111	DATA[32]
100101000	DATA[33]
000101001	DATA[34]
000101010	DATA[35]
100101011	DATA[36]
000101100	DATA[37]
100101101	DATA[38]
100101110	DATA[39]
000101111	DATA[40]
100110000	DATA[41]
000110001	DATA[42]
000110010	DATA[43]
100110011	DATA[44]
000110100	DATA[45]
100110101	DATA[46]
100110110	DATA[47]
000110111	DATA[48]
000111000	DATA[49]
100111001	DATA[50]
100111010	DATA[51]
000111011	DATA[52]

Table 9-17. L2 Data Syndrome Bits (Continued)

L2DATASYN[0–8]	Bit in Error
100111100	DATA[53]
000111101	DATA[54]
000111110	DATA[55]
100111111	DATA[56]
001000000	DATAECC[2]
101000001	DATA[57]
101000010	DATA[58]
001000011	DATA[59]
101000100	DATA[60
001000101	DATA[61]
001000110	DATA[62]
101000111	DATA[63]
101001000	DATA[64]
001001001	DATA[65]
001001010	DATA[66]
101001011	DATA[67]
001001100	DATA[68]
101001101	DATA[69]
101001110	DATA[70]
001001111	DATA[71]
101010000	DATA[72]
001010001	DATA[73]
001010010	DATA[74]
101010011	DATA[75]
001010100	DATA[76]
101010101	DATA[77]
101010110	DATA[78]
001010111	DATA[79]
001011000	DATA[80]
101011001	DATA[81]
101011010	DATA[82]
001011011	DATA[83]

Table 9-17. L2 Data Syndrome Bits (Continued)

L2DATASYN[0–8]	Bit in Error
101011100	DATA[84]
001011101	DATA[85]
001011110	DATA[86]
101011111	DATA[87]
101100000	DATA[88]
001100001	DATA[89]
001100010	DATA[90]
101100011	DATA[91]
001100100	DATA[92]
101100101	DATA[93]
101100110	DATA[94]
001100111	DATA[95]
001101000	DATA[96]
101101001	DATA[97]
101101010	DATA[98]
001101011	DATA[99]
101101100	DATA[100]
001101101	DATA[101]
001101110	DATA[102]
101101111	DATA[103]
001110000	DATA[104]
101110001	DATA[105]
101110010	DATA[106]
001110011	DATA[107]
101110100	DATA[108]
001110101	DATA[109]
001110110	DATA[110]
101110111	DATA[111]
101111000	DATA[112]
001111001	DATA[113]
001111010	DATA[114]
101111011	DATA[115]

Table 9-17. L2 Data Syndrome Bits (Continued)

L2DATASYN[0–8]	Bit in Error
001111100	DATA[116]
101111101	DATA[117]
101111110	DATA[118]
001111111	DATA[119]
010000000	DATAECC[1]
110000001	DATA[120]
110000010	DATA[121]
010000011	DATA[122]
110000100	DATA[123]
010000101	DATA[124]
010000110	DATA[125]
110000111	DATA[126]
110001000	DATA[127]
Other	Detectable; uncorrectable or invalid

C code used to generate the data syndrome table:

```
/***********************************************************************/

/* Generates a table of the DataECC syndromes in binary for PPC 620 */

/***********************************************************************/


#include <stdio.h>
main () {
int g,i,j=0;

printf("L2SR<17:25> -- L2DATAECC<0:8> Syndrome\n\n");
printf("Single Bit ECC Failure Indication Only\n\n");
printf("DataECC Syndrome\tBit in Error\n");
printf("----------------\t------------\n");
printf("  000000000\t\t  NONE\n");
printf("  100000000\t\t  DataECC<0>\n");

/* The Data symbol is composed of 128 data bits + 9 ECC bits = 137 bits */
/* 8 ECC bits are derived from the data and the 9th is parity for */
/* the entire symbol.  In single bit errors, which is all this ECC code */
/* will detect, this most significant ECC bit also happens to yield an */
/* odd parity for the syndrome */\
for(i=1;i<137;i++) {

/* Special cases: ECC bits are interspersed at powers of 2 */
  if (i == 0x001 || i == 0x002 || i == 0x004 || i == 0x008 ||
      i == 0x010 || i == 0x020 || i == 0x040 || i == 0x080) {
    switch (i)
      {
      case 0x001 :
        printf("  000000001\t\t  DataECC<8>\n");
         break;
       case 0x002 :
        printf("  000000010\t\t  DataECC<7>\n");
         break;
       case 0x004 :
        printf("  000000100\t\t  DataECC<6>\n");
         break;
       case 0x008 :
        printf("  000001000\t\t  DataECC<5>\n");
         break;
       case 0x010 :
        printf("  000010000\t\t  DataECC<4>\n");
         break;
       case 0x020 :
        printf("  000100000\t\t  DataECC<3>\n");
         break;
        case 0x040 :
        printf("  001000000\t\t  DataECC<2>\n");
         break;
       case 0x080 :
        printf("  010000000\t\t  DataECC<1>\n");
         break;
       default :
        printf("%x\t\tunknown) \n");
       }
  }
  else {
    /* Figure out what the ECC parity bit for this combination should be */
    g = giveparity(i);
    printf("  ");
    printbits(g);
    printf("\t\t  Data<%d>\n",j);
    j++;
  }
}
printf("    Other\t\t INVALID\n\n\n",i,j);
}


int printbits(int num) {
/* This function prints the binary sequence for a 9-bit number */
/* as ASCII characters to <STDOUT> */
```

```
  int lcv, mask=0x100;

  for (lcv=0;lcv<9;lcv++) {
    ((mask & num) == 0) ? printf("0") : printf("1");
    mask >>= 1;
  }
}

int giveparity(int num) {
/* Returns a 9 bit number with odd parity (MSB) given an 8 bit number */

  int lcv, par=0, mask=1;

  for (lcv=0;lcv<9;lcv++) {
    if ((mask & num) != 0) par++;
    mask <<= 1;
  }
  if ((par % 2) == 1)
       return num;
  else return num | 0x100;
}
```

9.3.2.5 L2ECCADDR[0–24]

L2ECCADDR (L2SR[35–59) is the L2 quad word address of the first detected ECC error. These 25 bits map to bits 11 to 35 of the 40 bit address brought off chip (located on external address bus pins 35–59, which is also the same bit location as the position of this address in the L2SR register). The address will be held until the syndromes are cleared. The address written is qualified by either of the tag or data syndromes not having a 0 value. The index can be determined by masking bits according to Table 9-4.

9.4 ECC—L2 Error Detection and Correction

The data in the cache is protected by ECC. The data portion is protected with a 9-bit ECC code that covers all 128-bits of L2DATA. The tag portion is protected with a 6-bit ECC code, which covers L2TAG, the portion of L2TAGADD used for tag, and L2COHERENCY.

9.4.1 The ECC Algorithm

The syndromes generated are based on the modified Hamming code algorithm. The tag data portion used to generate the syndrome is L2COHERENCY[0–1] concatenated with L2TAG[0–10] concatenated with L2TAGADD[0–8]. If the full tag width is not required, based on L2SIZE, the L2TAGADD bits not used are forced to logic zero, see Table 9-3.

See Table 9-16 and Table 9-17 for the tag and data syndrome bit values.

9.4.2 ECC Correction Modes

The ECC associated with the L2 can operate in three modes which are software configurable:

- Always correct mode
- Automatic switch correct mode
- Never correct mode

9.4.2.1 Always Correct Mode

The data is encoded as it is written to the L2 cache. Data coming from the cache is always corrected.

When a correctable error is detected the L2 interface will log that an error occurred, store the cache index of the tag or data that caused the error, and the syndrome that resulted from the error. It is software's responsibility to poll this information which is in the L2SR register.

When an uncorrectable error is encountered a machine check interrupt is generated.

9.4.2.2 Automatic Switch Correct Mode

The data is encoded as it is written to the L2 cache. Data coming from the cache follows two paths: a corrected path and a non-corrected path. The corrected path takes a cycle longer than the non-corrected path. Under normal conditions data flowing from the L2 to the bus follows the corrected path. Instructions flowing from the L2 to L1 follows the uncorrected path, while data transferred from the L2 to the L1 data cache always follows the corrected path regardless of the ECC correction mode selected.

When a correctable error is detected for data that is going from the L2 to the L1, hardware automatically switches to the corrected path. All data from the L2 to L1 will follow the corrected path after the detection of a correctable error. The L2 interface will log that an error occurred, store the cache index of the tag or data that caused the error, and the syndrome that resulted from the error. It is software's responsibility to poll this information. Switching back to the uncorrected path is done automatically by hardware when the next write following the error is performed by the L2 interface.

When an uncorrectable error is encountered a machine check interrupt is generated.

9.4.2.3 Never Correct Mode

The data is not encoded as it is written to the L2 and the output buffers that drive the data lines for ECC are disabled. Data is not corrected as it flows from the L2 to the L1 data cache or the bus. There is no data load latency advantage for ECC disabled as compared to the other ECC modes. During instructions fetch transactions latency is reduced by one processor clock cycle when the never correct ECC mode is selected.

9.5 L2 Cache Timing Diagrams

Figure 9-2 through Figure 9-23 show the timing diagrams for the L2 interface.

Note: Assume that L2RATIOSR is at 1:1.

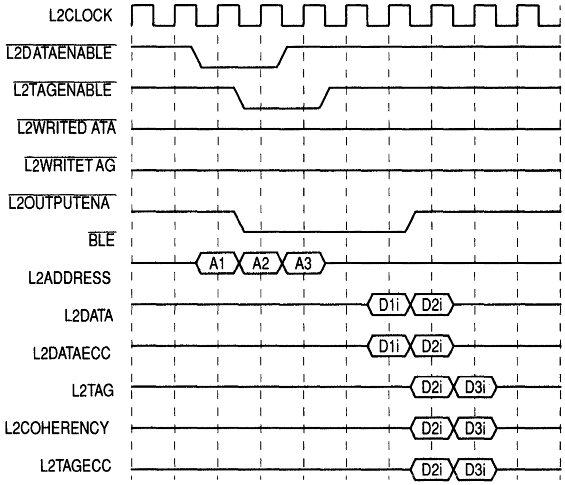


Figure 9-2. Timing of Quadruple Register L2 Reads

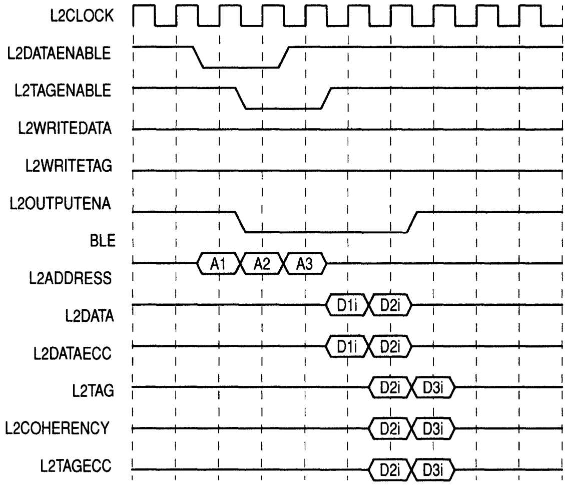


Figure 9-3. Timing of Triple Register L2 Reads

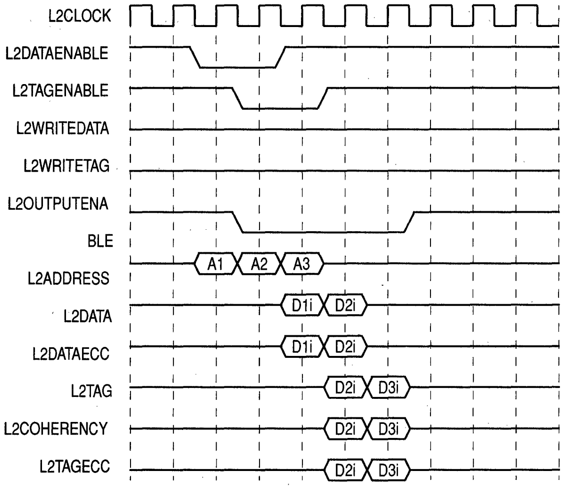


Figure 9-4. Timing of Double Register L2 Reads

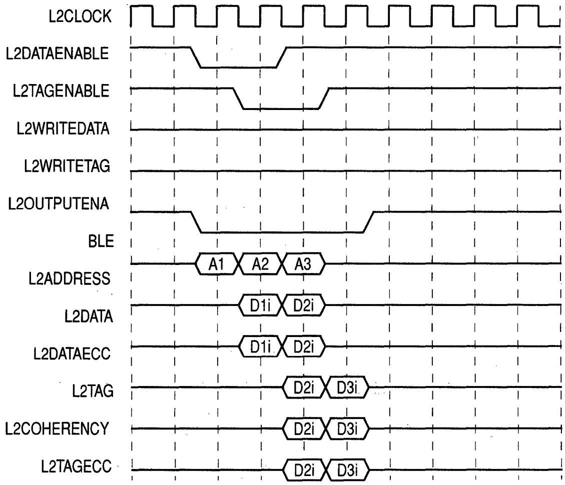


Figure 9-5. Timing of Single Register L2 Reads

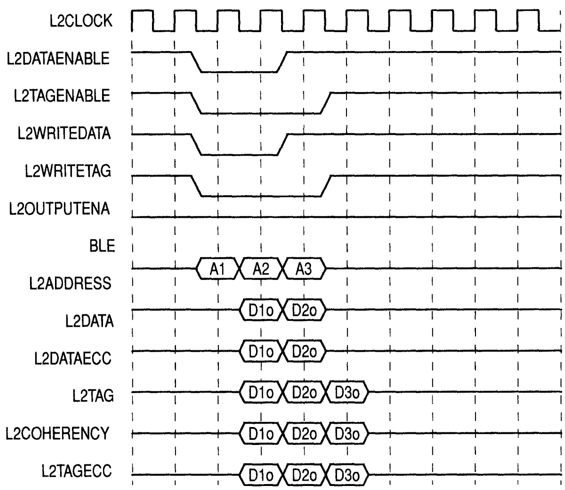


Figure 9-6. Timing of L2 LateWrites for all Register Depths

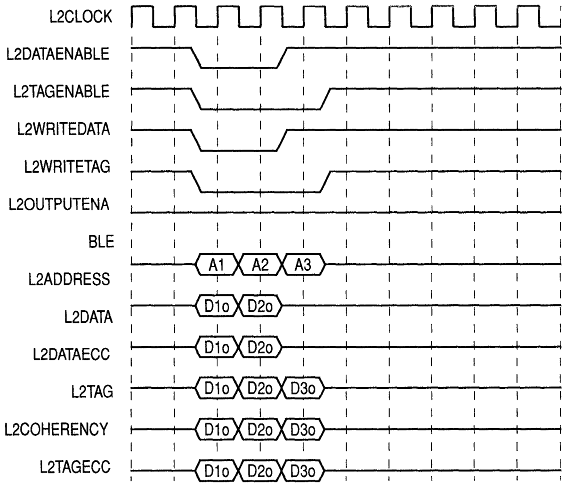


Figure 9-7. Timing of L2 Non-LateWrites for all Register Depths

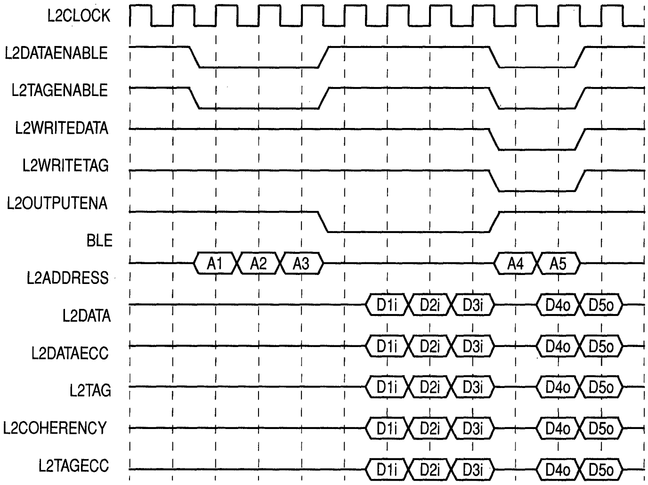


Figure 9-8. Timing of Quadruple Register L2 Reads followed by LateWrites

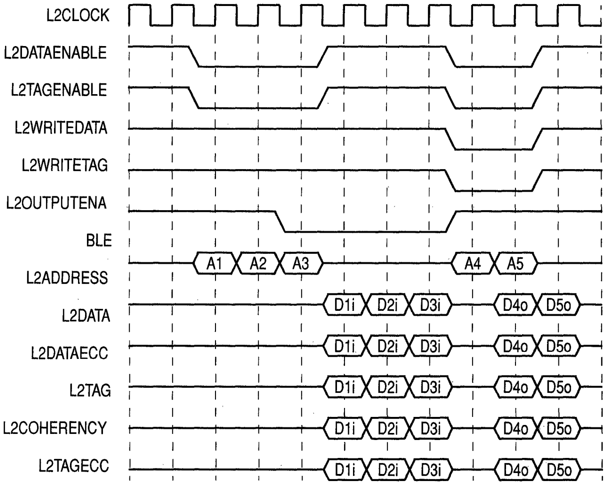


Figure 9-9. Timing of Triple Register L2 Reads followed by LateWrites

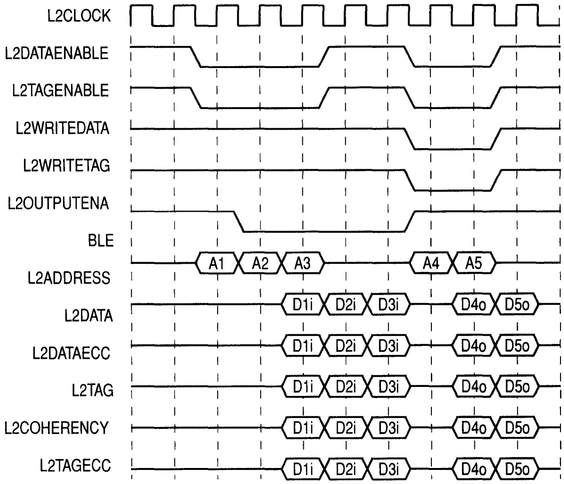


Figure 9-10. Timing of Double Register L2 Reads followed by LateWrites

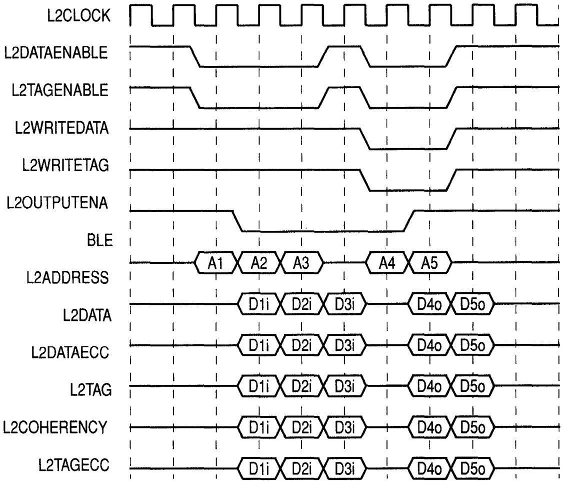


Figure 9-11. Timing of Single Register L2 Reads followed by LateWrites

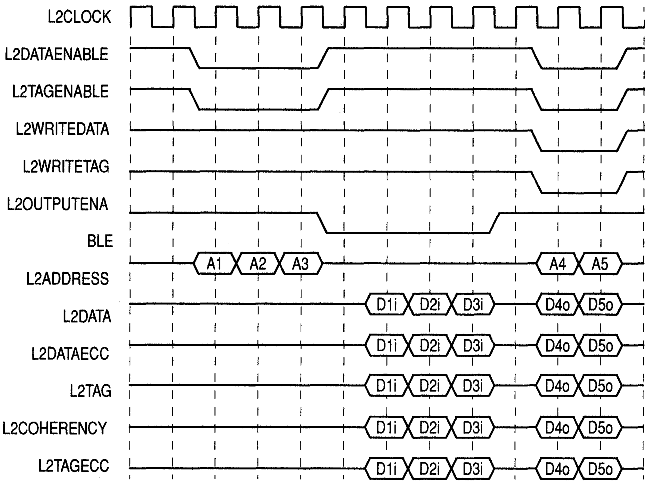


Figure 9-12. Timing for Quadruple Register L2 Reads followed by Non-LateWrites

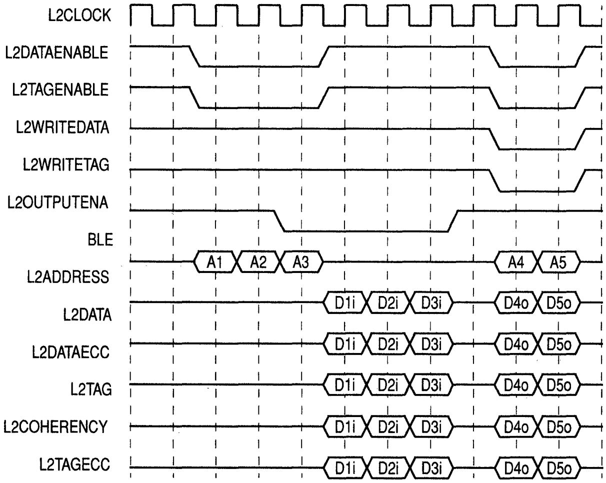


Figure 9-13. Timing for Triple Register L2 Reads followed by Non-LateWrites

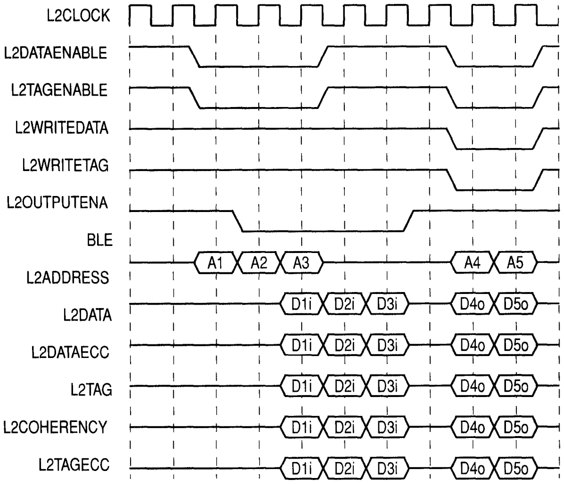


Figure 9-14. Timing for Double Register L2 Reads followed by Non-LateWrites

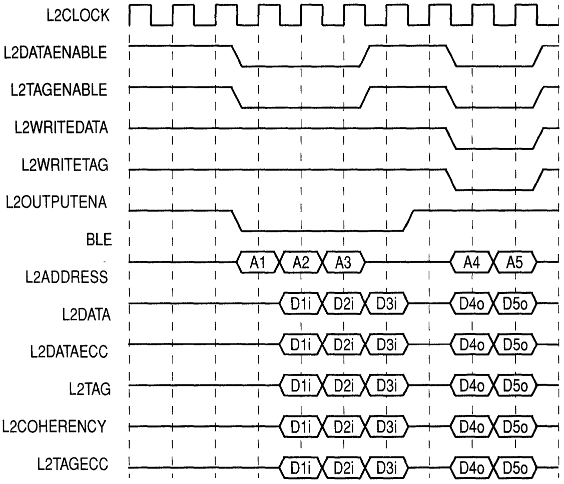


Figure 9-15. Timing for Single Register L2 Reads followed by Non-LateWrites

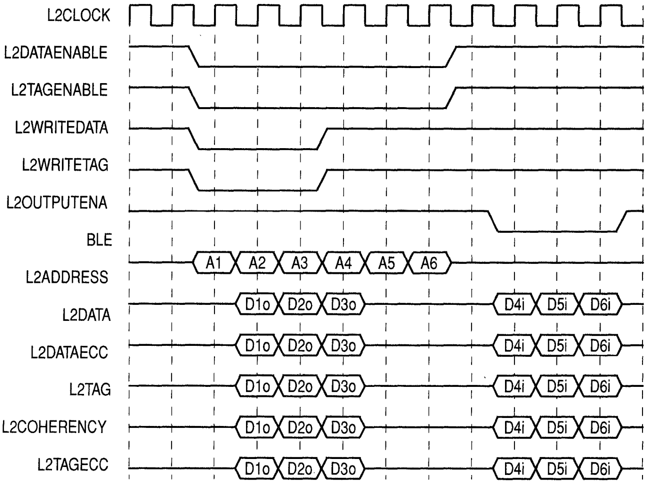


Figure 9-16. Timing for Quadruple Register L2 LateWrites followed by Reads

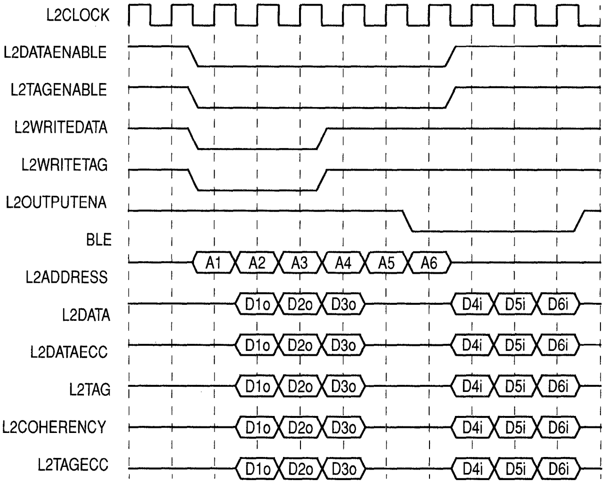


Figure 9-17. Timing for Triple Register L2 LateWrites followed by Reads

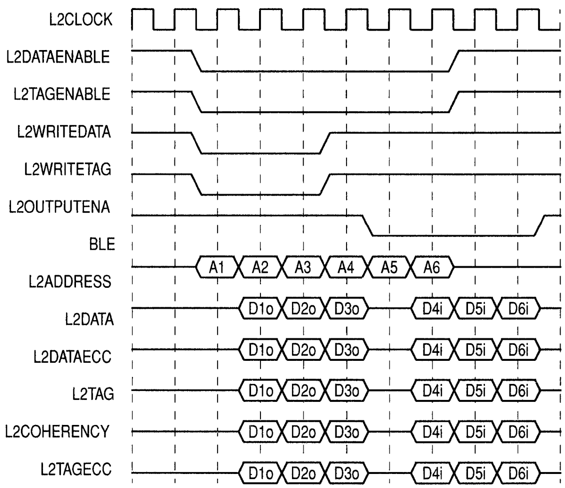


Figure 9-18. Timing of Double Register L2 LateWrites followed by Reads

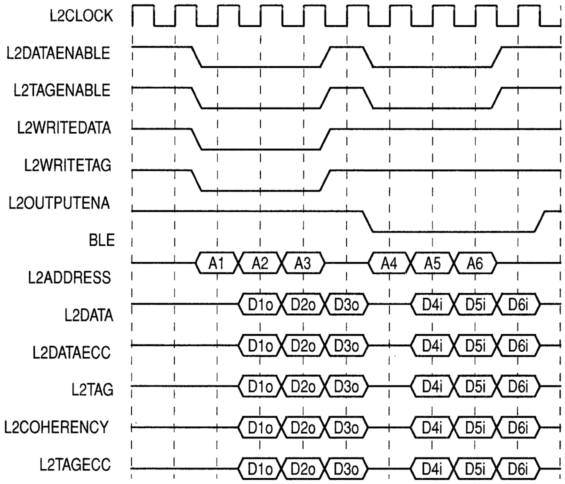


Figure 9-19. Timing of Single Register L2 LateWrites followed by Reads

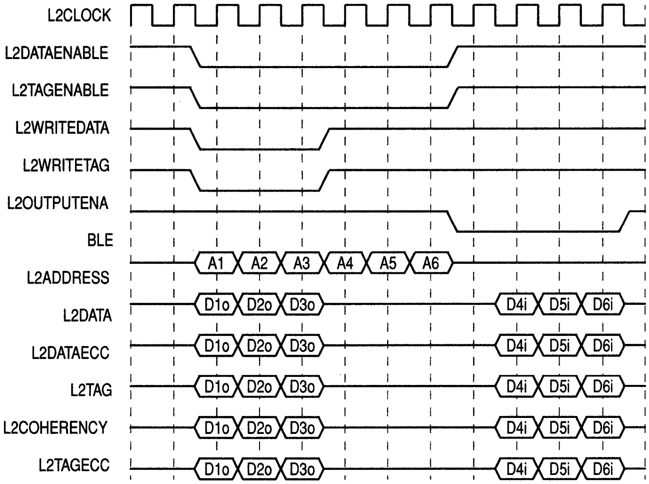


Figure 9-20. Timing for Quadruple Register L2 Non-LateWrites followed by Reads

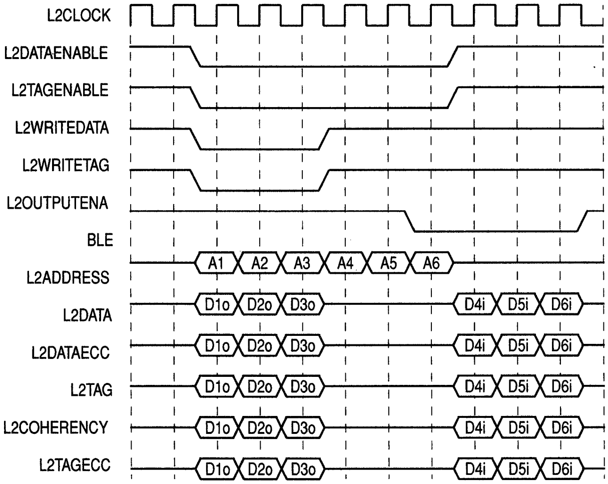


Figure 9-21. Timing for Triple Register L2 Non-LateWrites followed by Reads

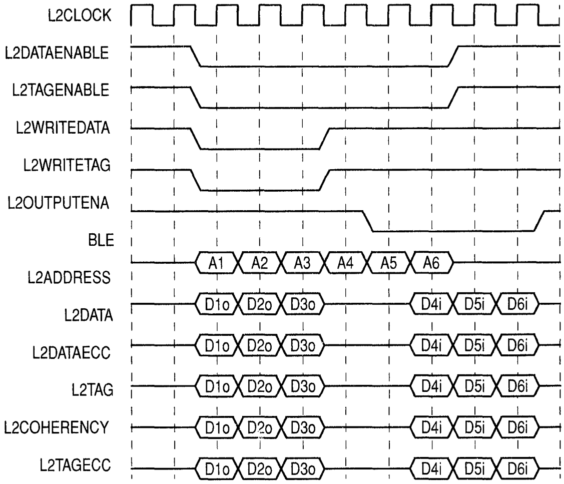


Figure 9-22. Timing for Double Register L2 Non-LateWrites followed by Reads

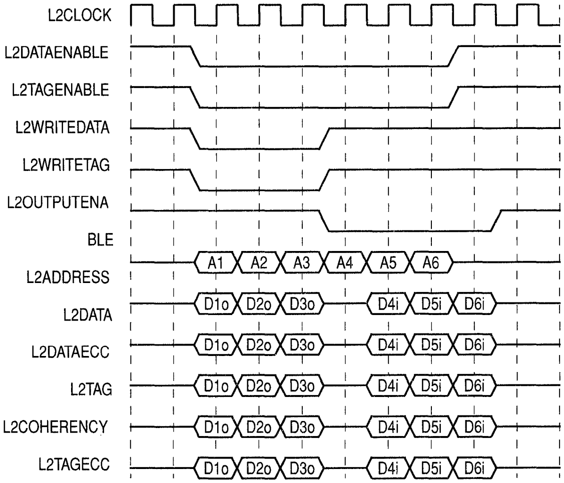


Figure 9-23. Timing for Single Register L2 Non-LateWrites followed by Reads

Chapter 10
Performance Monitor

The PowerPC 620 microprocessor provides a performance monitor facility to monitor selected 620 characteristics and to facilitate the generation of instruction traces. The performance monitor facility is not defined by the PowerPC architecture.

10.1 Overview

The performance monitor can be used for the following:

- To increase system performance with efficient software (especially in an MP system), memory hierarchy behavior must be understood in order to develop algorithms that schedule tasks (and perhaps partition them) and that structure and distribute data optimally.
- To improve processor and system architecture, 620's behavior must be known and understood in software environments of interest. Some environments may not be easily characterized by a benchmark or trace (for example, MP) and are therefore not easily investigated with simulators.
- The performance monitor helps system developers to debug and analyze their systems.

The capabilities of the 620 performance monitor include:

- Concurrently counting the number of occurrences of eight different, software-selectable events of interest on eight 32-bit counters
- To start the counting of software-selectable events (X_i) on the condition that n events of a second type (W) have occurred.
- The periodical designation of an instruction as a "sampled instruction" and recording information about it if it causes an event of interest.
- Counting the sampled storage instructions whose access time exceeds a software settable value (called thresholding).
- Storage of 32 bits of data per counter which indicates the status of the software-selectable events for a period of 32 cycles (called history mode).
- Mark a process to enable the counters while executing the instructions related to this process.

10.1.1 Functional Overview

The following subsections describe the functional characteristics of the 620.

10.1.1.1 Special Purpose Registers (SPR)

The contents of these registers can be read or modified by the **mtspr** and **mfspr** instructions. These MMCR0, MMCR1 and PMC1 thru PMC8 registers can be modified only in privilege state, but they can be read in privilege or problem state. The SIA and SDA registers can not be modified by software but they can be read in privilege or problem state.

- Monitor mode control register 0 and 1 (MMCR0 and MMCR1)—Controls the behavior of the Performance Monitor. Provides the ability to select the events to count and when they will be counted, set the Threshold value, select the Time Base input, enable History Mode and select the conditions that will enable a Performance Monitor interrupt.
- Performance monitor counters 1 thru 8 (PMC1 thru PMC8)—Store the number of times a software selectable event (maximum of one event per counter at a time) has occurred since the Performance Monitor was enabled for counting.
- Sample instruction address (SIA)—Stores the address of a sampled instruction.
- Sample data address (SDA)—Stores the address associated with the data used by the sampled instruction.

10.1.1.2 Thresholder

Provides the capability to obtain the distribution of memory access times that results from an application's accesses to all levels of a system's memory hierarchy, including main memory, during its execution. The threshold value is set by the software and can range from 0 to 63 cycles.

10.1.1.3 Time Base Interface

Allows the selection of four Time Base bits. When a transition from zero to one in the selected bit is detected the Performance Monitor can be configured to generate an interrupt or start monitoring the selected events.

10.1.1.4 Interrupt

An interrupt can be generated if the conditions specified by the software in the MMCR0 and MMCR1 are met. The conditions in which an interrupt can be generated are when a zero to one transition is detected on the Time Base selected bit and when the contents of one or more of the counters PMC1 thru PMC8 becomes negative (Most Significant Bit == one).

10.1.1.5 Event Selection

A maximum of one event per counter can be monitored at a given time. The event to be monitored is selected by software. For some of the events it is possible to count the cycles associated to the duration of an event as well as the number of times the event is detected.

10.1.1.6 Monitor Modes

The Performance Monitor can be enabled during a specific execution state (privilege/problem) and/or during the execution of a software marked process. A counter can also be enabled after a Performance Monitor interrupt is generated or after a software preset number of events is detected.

10.1.2 Event Counting Overview

Eight counters in the 620 count the occurrences of software-selectable events. The Monitor Mode Control Registers (MMCR0 and MMCR1) are used to control the performance monitor operation. The counters and the MMCRs are readable and writable Special Purpose Registers (SPRs). Control fields in the MMCRs select the events to be counted, indicate conditions that enable the performance monitor interrupt, set the Thresholder value, enable History Mode and specify the conditions under which counting is enabled, etc.

Counting is enabled if the condition in the processor state matches a software-specified condition. Since a software task scheduler may switch a processor's execution among multiple processes and because statistics on only a particular process may be of interest, a facility is provided to mark a process. A control bit (bit 61) in the Machine State Register (MSR), MSR(PMM), the "Performance Monitor Mark" bit, is used for this purpose. System software may set this bit when a "Performance Monitor marked process" is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR(PR) and MSR(PMM) together define a state that the processor (Privileged or Problem) and the process (marked or unmarked) may be in at any time. If this state matches a state specified by the MMCRs, the state for which monitoring is enabled, counting is enabled. The following are the possible states that can be monitored:

1. (Privileged) only
2. (Problem) only
3. (Marked and Problem) only
4. (Not Marked and Problem) only
5. (Marked and Privileged) only
6. (Not Marked and Privileged) only
7. (Marked) only
8. (Not Marked) only

In addition, one of two unconditional counting modes may be specified:

1. Counting is unconditionally enabled regardless of the states of MSR(PMM,PR)
2. Counting is unconditionally disabled regardless of the states of MSR(PMM,PR)

10.1.3 Triggering Modes Overview

The Performance Monitor provides several ways to trigger (enable for the first time) the event counting process.

- The execution of a **mtspr** instruction to a Performance Monitor Special Purpose Register (MMCRs or PMCs) that creates the scenario which causes the sampling process to be enabled. This is the default triggering mode. An example will be, a write to the MMCR0 in which counters are unconditionally enabled.
- An Instruction Address Break Point Register (IABR) match.
- A selected Time Base bit transition from zero to one.
- A Performance Monitor Counter 1 (PMC1) negative value.

The Monitor Mode Control Registers (MMCRs) provide the primary control for these operations but they are also affected by the contents of the Performance Monitor Counter 1 (PMC1), Time Base register and the Instruction Address Break Point Register. The interaction of these trigger mechanisms among themselves and with the different monitor modes described previously, allows the user to create many different scenarios.

10.1.4 Instruction Address Break Point Register (IABR) Match

There are some differences between the Performance Monitor IABR match and the normal IABR behavior. These differences are listed below.

1. An IABR match is not dependent on the IABR being turned on or the translation mode matching, but is only dependent on the address matching.
2. An IABR match will only occur on instructions that go into the branch queue (any branch instruction and condition register logical instructions).
3. The IABR indication occurs when the branch finishes, and not completes, therefor if the instruction is cancelled by an interrupt after finishing the match will still be reported.
4. The IABR match is only valid if the Performance Monitor is enabled, refer to Section 10.6, “The Counters,” for more information.

10.1.5 Event Sampling and Thresholder Overview

It is sometimes desirable to link an event to the instruction that caused it and to save certain information about the instruction. Hardware is simplified by providing this capability for sampled instructions only. Periodically, an instruction entering the processor is designated as a sampled instruction. Certain information about the instruction is recorded as it is processed and is made available if its execution causes an event of interest.

Software developers may want to know the effective address of an instruction that caused an L2 miss and the effective address of its operand. The performance monitor provides this information for “sampled” Load and Store instructions only. To simplify hardware, at most one sampled instruction is allowed to be in the processor at one time. Since no more than

16 instructions can be in the 620 at once (the completion buffer size), every 16th instruction that is dispatched to the execution units is given a special tag that identifies it as a "sampled instruction". The address of a sampled instruction is saved in an SPR called the Sampled Instruction Address register (SIA register). If the sampled instruction is a Load or Store instruction, the address of its operand is saved in a SPR called the Sampled Data Address register (SDA register). The number of L2 misses caused by sampled instructions may be counted. If a Performance Monitor exception occurs, subsequent writes to the SIA and the SDA are not allowed and a performance monitor interrupt may be initiated. Software may then record the information in these registers. Since the SIA and the SDA are written into during different machine cycles, it is necessary to indicate when their contents relate to the same instruction, i.e., that their contents are "coupled". A bit is set in SRR1, SRR1[1], when the SDA is written into and it is reset when the SIA is written into. The contents of the registers are coupled when SRR1[1] is a "1".

The performance monitor can also gather information about the access times of sampled Load and Store instructions. If the access time of a sampled instruction exceeds a software-specified number of cycles, a counter is incremented. This is called *thresholding* - a counter is incremented if the access time of a sampled instruction exceeds the threshold. A histogram of access times can be constructed by recording the number of accesses that exceed each threshold in a series of decreasing thresholds. The histogram generated reveals the average access time to each level of memory hierarchy and any access time variations to a given level that may be caused by resource conflicts (e.g., bus conflicts and interleaved memory bank conflicts). This information is especially useful in systems without an L2 cache.

The thresholding facility may be used with sampled Store instructions that store into cacheable lines. The Thresholder considers a sampled Store instruction to be complete when its data is written into a cache line in L1. A Store instruction that accesses a memory location that is not cacheable would not have the instruction's address, SIA contents, and the data address, SDA contents, held; the 620 can not determine when it completes. The instruction and data address will be stored in the registers but it can be easily be overwritten.

10.1.6 History Mode Overview

The Performance Monitor provides a way to relate the selected events in a cycle by cycle basis. In this mode the events are stored as they are received no addition is performed during this mode. A maximum of 32 cycles per counter can be stored. The counters can be linked to achieve several combinations. For example:

- 8 events with 32 cycles each
- 4 events with 64 cycles each
- 6 events, 4 with 32 cycles and 2 with 64 cycles

The intent of this feature is for small loops or small sections of the code to be analyzed in detail. The data is stored in the PMC registers. The most significant bit is the oldest cycle while the least significant bit is the most recent one.

10.1.7 Trace Mode Overview

The performance monitor facilitates the construction of instruction traces. The processor issues instructions in sequence and one at a time in trace mode. An instruction is not issued until the instruction preceding it in the instruction stream completes. In trace mode, the following information is recorded in SRR1 about the last instruction that completed:

1. It was a Load instruction
2. It was a Store instruction
3. It accessed the operand field length field (bits 25-31) of the XER register
4. It may have changed the effective-to-virtual address map

In addition, in trace mode, the EA of the operand of each Load and Store instruction (sampled or not) executed is saved in the SDA.

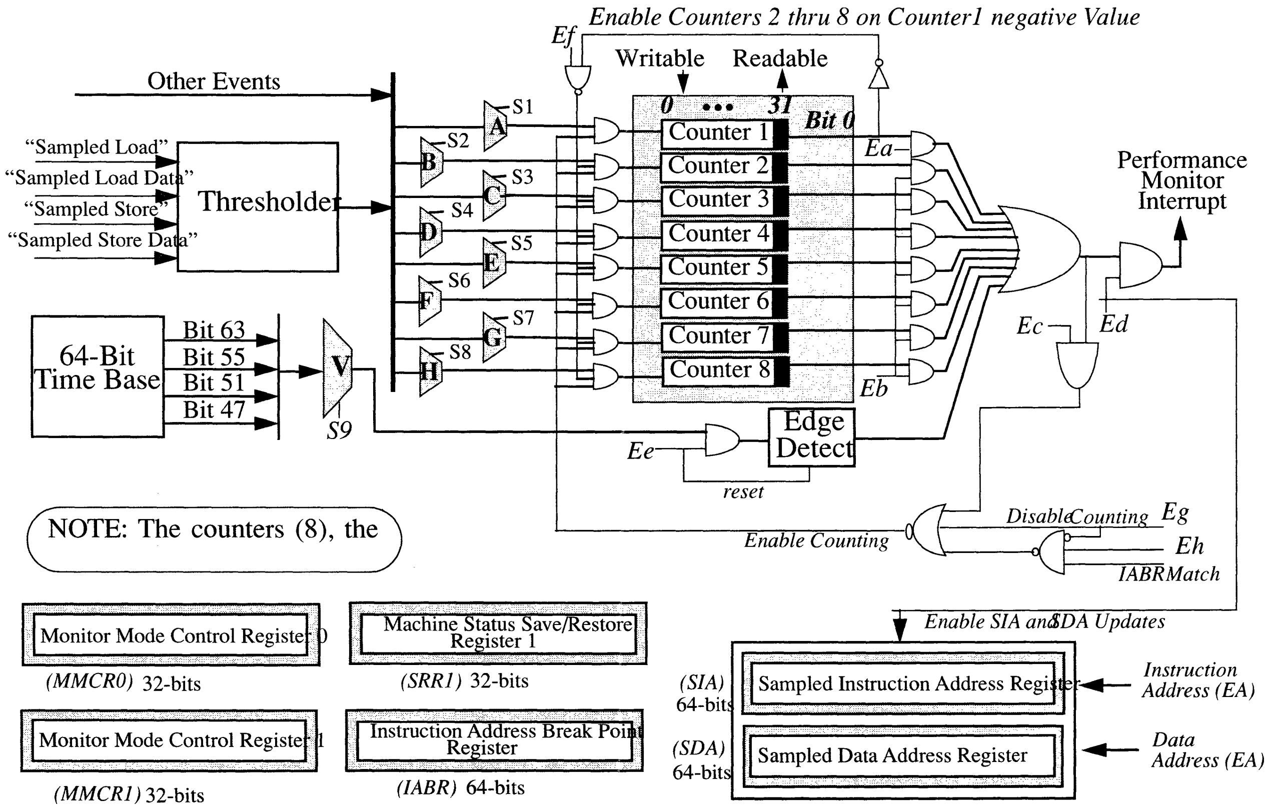
10.2 Performance Monitor Components

620 supports the following hardware structure for performance monitoring:

- Interaction with the following registers: (See "Related Registers" on page 9.)
  - — Time Base
  - — Machine Status Save/Restore register 1 (SRR1)
  - — Instruction Address Break Point Register (IABR)
  - — Sampled Instruction Address Register (SIA)
  - — Sampled Data Address Register (SDA).
- Two 32-bit mask registers to select events to count, refer to Section 10.4, "Monitor Mode Control Registers (MMCR0 and MMCR1)."
- A multiplexer structure to allow programmable event selection for eight counters.

  See Section 10.6, "The Counters," and Section 10.7, "Detailed Description of Events."
- Memory access times collection block. Refer to Section 10.5, "The Thresholder."
- Exception generation mechanism. Refer to Section 10.8, "Performance Monitor Interrupt."
- A cycle by cycle event status storage. Refer to Section 10.9, "History Mode."

Figure 10-1 shows the block diagram of the 620 performance monitor, and Table 10-1 lists the special purpose register addresses of the performance monitor SPR registers.



NOTE: The counters (8), the

Figure 10-1. Performance Monitor Block Diagram

- Legend for Figure 10-1:
  - Ea - MMCR0[16] Enable PMC1 exception signalling
  - Eb - MMCR0[17] Enable PMCn, n>1 exception signalling
  - Ec - MMCR0[6] Freeze counting on PMCn when a Performance Monitor Exception condition is detected
  - Ed - MMCR0[5] Performance monitoring signal exception enable.
  - Ee - MMCR0[9] Time base signal exception enable.
  - Ef - MMCR0[18] Disable PMCn (n>1) counting until PMC1 becomes negative.
  - Eg - MMCR0[0:4] Monitor mode and unconditionally disable bits.
  - Eh - MMCR1[29] Freeze Counters until IABR match.
  - S1 - MMCR0[19:25] PMC1 Event selection bits
  - S2 - MMCR0[26:31] PMC2 Event selection bits
  - S3 - MMCR1[0:4] PMC3 Event selection bits
  - S4 - MMCR1[5:9] PMC4 Event selection bits
  - S5 - MMCR1[10:14] PMC5 Event selection bits
  - S6 - MMCR1[15:19] PMC6 Event selection bits
  - S7 - MMCR1[20:24] PMC7 Event selection bits
  - S8 - MMCR1[25:28] PMC8 Event selection bits
  - S9 - MMCR0[7:8] Time Base selection bits
  - A - PMC1 Event selection MUX
  - B - PMC2 Event selection MUX
  - C - PMC3 Event selection MUX
  - D - PMC4 Event selection MUX
  - E - PMC5 Event selection MUX
  - F - PMC6 Event selection MUX
  - G - PMC7 Event selection MUX
  - H - PMC8 Event selection MUX
  - V - Time Base bit select MUX

Table 10-1. SPR Address of the 620 Performance Monitor Registers

SPR Address		r/w	Name	Function
5-9	0-4			
11000	XXXXX			Reserved for performance monitor
11000	00011	r	PMC1/RD	Performance Monitor Counter 1 (read only)
11000	00100	r	PMC2/RD	Performance Monitor Counter 2 (read only)
11000	00101	r	PMC3/RD	Performance Monitor Counter 3 (read only)
11000	00110	r	PMC4/RD	Performance Monitor Counter 4 (read only)
11000	00111	r	PMC5/RD	Performance Monitor Counter 5 (read only)
11000	01000	r	PMC6/RD	Performance Monitor Counter 6 (read only)
11000	01001	r	PMC7/RD	Performance Monitor Counter 7 (read only)
11000	01010	r	PMC8/RD	Performance Monitor Counter 8 (read only)
11000	01011	r	MMCR0/RD	Monitor Mode Control Register 0 (read only)
11000	01110	r	MMCR1/RD	Monitor Mode Control Register 1 (read only)
11000	01100	r	SIA	Sampled Instruction Address register
11000	01101	r	SDA	Sampled Data Address register —SDA(0:61) is physically the same as the, Data Address Breakpoint Register, DABR(0:61)
11000	10011	r/w	PMC1	Performance Monitor Counter 1
11000	10100	r/w	PMC2	Performance Monitor Counter 2
11000	10101	r/w	PMC3	Performance Monitor Counter 3
11000	10110	r/w	PMC4	Performance Monitor Counter 4
11000	10111	r/w	PMC5	Performance Monitor Counter 5
11000	11000	r/w	PMC6	Performance Monitor Counter 6
11000	11001	r/w	PMC7	Performance Monitor Counter 7
11000	11010	r/w	PMC8	Performance Monitor Counter 8
11000	11011	r/w	MMCR0	Monitor Mode Control Register 0
11000	11110	r/w	MMCR1	Monitor Mode Control Register 1

10.3 Related Registers

The following subsections provide information on registers related to the performance monitor.

10.3.1 Sampled Instruction Address Register (SIA)

The SIA contains the effective address of the last sampled instruction to be dispatched. This instruction may be a speculative instruction in a correct or an incorrect execution path.

Since the instruction stream may be restarted for various reasons (e.g., an incorrectly predicted branch), the sampled instruction may not be executed. When a Performance Monitor exception is requested, the content of the SIA is frozen. It will contain the effective address of an instruction that was dispatched in the 620 shortly before (no further in time than 16 instruction executions) the occurrence of the interrupt-causing condition. If the Performance Monitor is configured to cause an interrupt on an event that is caused by a sampled load or store instruction, the SIA will contain the effective address of the sampled instruction causing the event.

10.3.2 Sampled Data Address Register (SDA)

The SDA contains the effective address of the operand of the last sampled storage instruction to be executed by the Load/Store Unit. When a Performance Monitor exception is requested, the content of the SDA is frozen. It may contain the effective address of the operand of the instruction address stored in the SIA. When it does, bit 1 in the SRR1 is set as described below.

On the 620 the SDA and the DABR use the same physical register to store the data. This will create some limitations described below.

10.3.2.1 SIA and SDA Contents Freeze

The content of the SIA is modified in Performance Monitor mode. The SDA can be modified in Trace Mode, Performance Monitor mode and in DABR mode. To prevent the content of the SDA register to change (stay frozen) the processor can only be in one of these modes at a given time. Once the registers are frozen, if the processor enters one of the modes mentioned above the contents of the registers will no longer be frozen and their contents will be undefined. The following mode definitions are for illustration of the SIA and SDA behavior only:

- Performance Monitor Mode is active when the "Performance monitor Signal Exception Enable (PMXE)" bit, MMCR0[5]==1.
- Trace Mode is controlled by the MSR "Single-Step Trace Enable (SE)" and the "Branch Trace Enable(BE)" bits.
- DABR mode is controlled by the two least significant bits of the DABR register.

10.3.3 Machine Status Save/Restore Register 1 (SRR1)

SRR1 bits that provide information related to the SIA and SDA are shown in Table 10-2.

Table 10-2. SRR1 Performance Monitor Implemented Bit Fields

Bit	Description
1	Specifies whether the contents of the SIA and SDA are associated with the same instruction. 0 = SDA does not contain the effective operand address generated by the instruction that has its effective instruction address in the SIA. 1 = SDA contains the effective operand address generated by the instruction that has its effective instruction address in the SIA

10.4 Monitor Mode Control Registers (MMCR0 and MMCR1)

The monitor mode control registers (MMCR0, MMCR1) are partitioned into bit fields that allow the selection of events to be counted. Selection of allowable combinations of events causes the counters to operate concurrently.

Writes to the MMCRs are only allowed in privileged state. The MMCRs include controls such as counter enable control, counter negative interrupt control, counter event selection, and counter freeze control. Reading the MMCRs do not alter their content. The control functions of the fields in the MMCRs registers are defined in Table 10-3 and Table 10-4.

Table 10-3. MMCR0 Bit Fields

Bits	Mnemonic	Description
0	FC	Freeze Counters 0 = The PMCs are conditionally updated by processor events. 1= The PMCs are NOT updated by processor events.
1	FCS	Freeze Counting while in Supervisor state 0 = The PMCs are conditionally updated by processor events. 1 = The PMCs are not updated by processor events if MSR[PR] == 0.
2	FCP	Freeze Counting while in Problem state 0 = The PMCs are conditionally updated. 1 = The PMCs are not updated by processor events if MSR[PR] == 1.
3	FCM1	Freeze Counting while mark bit is set to one. 0 = The PMCs are conditionally updated. 1 = The PMCs are not updated by processor events if MSR(PMM) is set to 1.
4	FCM0	Freeze Counting while mark bit is set to zero. 0 = The PMCs are conditionally updated. 1 = The PMCn are not updated by processor events if MSR(PMM) is set to 0.
5	PMXE	Performance Monitor Exception Request Enable. This bit is reset to zero by hardware when a Performance Monitor exception is requested. 0 = Performance Monitor exception request is disabled. 1 = Performance Monitor exception request is enabled.
6	FCEX	Freeze counting of the PMCs when a Performance Monitor exception condition is detected (an enabled counter negative condition* or detection of an enabled time base transition). Could be overridden by MMCR0[18] 0 = Detection of an enabled counter negative condition or enabled time base transition has no effect on the behavior of the PMCs. 1= Detection of an enabled counter negative condition or enabled time base transition, freezes the contents of the PMCs until the condition is reset by software. Could be overridden by MMCR0[18].
7 - 8	TBSEL	Selects the Time Base bit whose transition from zero to one could cause a Performance Monitor interrupt (see bit 9 below). 00 = Time Base bit 63 01 = Time Base bit 55 10 = Time Base bit 51 11 = Time Base bit 47

Table 10-3. MMCR0 Bit Fields (Continued)

Bits	Mnemonic	Description
9	TBXE	Time Base exception request enable on bit transition (selected by above bits 7 and 8) from zero to one. Exception request needs also to be enable by MMCR0[5]. 0 = Do not request exception due to Time Base bit transition. 1 = Request exception if chosen Time Base bit transitions.
10-15	THRESHOLD	Threshold value. This number is multiplied by eight and the result is the number of processor cycles the threshold will be set to. Effective Threshold = MMCR0[10:15] X 8
16	PMC1XE	PMC1 Exception Request Enable. Exception request needs also to be enable by MMCR0[5]. 0 = PMC1 exception request disabled due to PMC1 negative value. 1 = PMC1 exception request enable due to PMC1 negative value.
17	PMCnXE	PMCn, n>1, Exception Request Enable. Exception request needs also to be enable by MMCR0[5]. 0 = PMCn, n>1, exception request disabled due to PMCn, n>1 negative value. 1 = PMCn, n>1, exception request enabled due to PMCn, n>1 negative value
18	TRIGGER	Disable PMCn, n>1, updates, until PMC1 has a negative value or a Performance Monitor exception is signalled (MMCR0[5] == 0). 0 = Enable PMCn, n>1, updates. 1 = Disable PMCn, n>1, updates until PMC1[0]==1 or until a Performance Monitor exception is requested (MMCR0[5]==0).
19-25	PMC1SEL	PMC1 event selector. See table: "Performance Monitor Counter 1 Monitored events and select bit pattern".
26-31	PMC2SEL	PMC2 event selector. See table: "Performance Monitor Counter 2 Monitored events and select bit pattern".

Notes:

1. Enabled Time Base transition is when a transition from zero to one is detected on the Time Base bit selected by MMCR0[7-8] and (MMCR0[9] == 1).
2. Enabled counter negative condition is when (PMCx[0] ==1) and (PMCyXE ==1).
3. "Conditionally updated" means that the counters will be updated if all the other MMCR's conditions are satisfied.

10

Table 10-4. MMCR1 Bit Fields

Bits	Mnemonic	Description
0 - 4	PMC3SEL	PMC3 event selector. See table: “Performance Monitor Counter 3 Monitored events and select bit pattern”.
5 - 9	PMC4SEL	PMC4 event selector. See table: “Performance Monitor Counter 4 Monitored events and select bit pattern”.
10 - 14	PMC5SEL	PMC5 event selector. See table: “Performance Monitor Counter 5 Monitored events and select bit pattern”.
15 - 19	PMC6SEL	PMC6 event selector. See table: “Performance Monitor Counter 6 Monitored events and select bit pattern”.
20 - 24	PMC7SEL	PMC7 event selector. See table: “Performance Monitor Counter 7 Monitored events and select bit pattern”.
25 - 28	PMC8SEL	PMC8 event selector. See table: “Performance Monitor Counter 8 Monitored events and select bit pattern”.
29	FCUIABR	Freeze Counters until IABR Match. After a “monitored” IABR match is detected this bit is reset to zero by the hardware. 0 = The PMCs are conditionally updated. 1 = The PMCs are not updated until a “monitored” IABR match occurs.
30	PMC1HIST	PMC1 History Mode 0 = PMC1 is conditionally incremented. 1 = PMC1 is in History mode. (See history mode section)
31	PMCnHIST	PMCn, n>1, History Mode 0 = PMCn, n>1, are conditionally incremented. 1 = PMCn, n>1, are in History Mode. (See History Mode section).

Notes:

1. An IABR match is said to be “monitored” if it occurs when PMC updates are permitted by MMCR0[0:4] and MSR[PR, PMM]
2. ”Conditionally incremented” means that the counters will be incremented if all the other MMCRs conditions are satisfied.

10.5 The Thresholder

It is often desirable to obtain the distribution of memory access times that results from an application’s accesses to all levels of a system’s memory hierarchy, including main memory, during its execution.

The 620 performance monitor therefore incorporates a facility called a thresholder that can obtain this information. It counts the number of sampled storage instructions whose access time exceeds a software-settable value or threshold. The counter value reflects the total number of instructions that exceeded the specified access time in machine cycles for the selected storage instruction. The access time of all sampled load instructions, with and without intervention, may be thresholded. Additionally, sampled store accesses, with and without intervention, to cacheable memory locations may be thresholded. The Thresholder circuit behaves in the following manner, when a sampled storage instruction is issued to the memory hierarchy, a decrementer is initialized with a value (threshold) that has been set by software in the MMCR0[10–15]. Then it is decremented by one every subsequent clock cycle. The threshold is said to be exceeded when the decrementer reaches zero before the storage instruction completes. It is not exceeded if the storage instruction is completed before the decrementer reaches zero. The term completed has different meaning depending of the type of storage instruction. For the load instructions it indicates that the data associated with the same request was received and for the store instructions it indicates that the data was written to the L1 cache. The Performance Monitor counter that is monitoring the Thresholder is incremented every time the threshold is exceeded.

By accumulating counts of accesses that exceed decreasing threshold values, a histogram can be generated that reveals the access time distribution of memory requests. The distribution reflects the proportionate numbers of "hits" at the various memory levels (e.g., L1, L2, and main memory) and will also reflect a smearing effect caused by bus and other resource conflicts that cause the access time to a given memory level (say to L2, for example) to vary over a range of values. This detailed distribution will reveal system effects that are not captured with previous approaches.

NOTE: The performance monitor can only threshold one type of sampled storage instruction at a given time. If a performance monitor counter is selected to monitor the threshold of two different types of storage instructions then the PMC value associated with the thresholder will reflect the value of the thresholded event with the highest priority as shown below (1 is the highest priority). Therefore the value of the thresholded lower priority event will be invalid.

1. Stores without intervention
2. Loads with intervention
3. Stores with intervention
4. Loads without intervention

If another sampled instruction is issued to the memory hierarchy while the decrementer in the Thresholder circuit is active, the Thresholder logic resets itself and begins decrementing the threshold value for the new sampled instruction. This scenario only happens when the instruction been monitored by the Thresholder was cancelled.

Figure 10-2 shows an illustrative distribution of access times that might be measured using the Thresholder on an imaginary system. Notice that main memory access times vary over a wide range in this example—perhaps due to interleaved memory bank conflicts or system bus conflicts. A distribution of L1 hit access times reveals how well the structure of the L1 supports the application running when this data was gathered. Figure 10-2 through Figure 10-6 provide timing information and block diagrams for various Thresholder states.

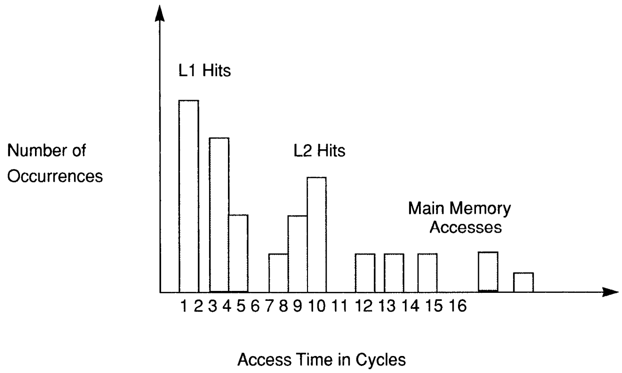


Figure 10-2. Access Times of an Imaginary System Using the Thresholder

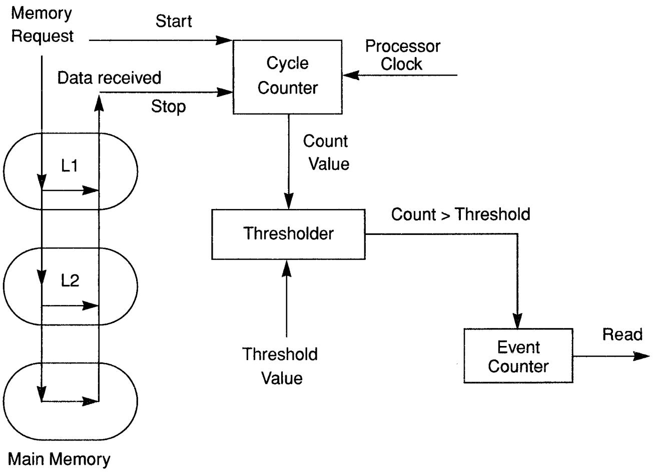


Figure 10-3. Block Diagram of the Delay Thresholding Mechanism for Load Instructions

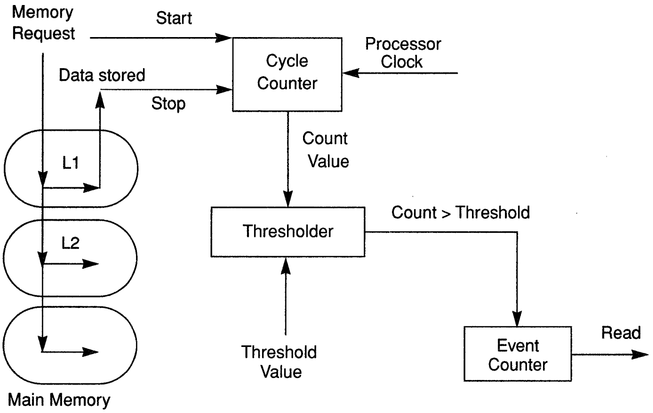
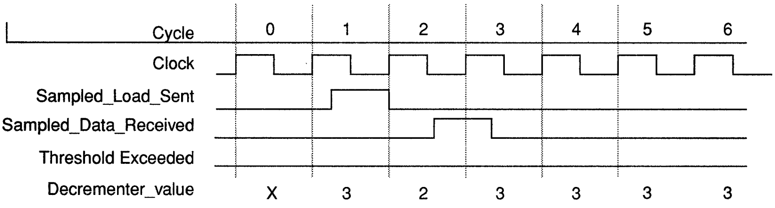
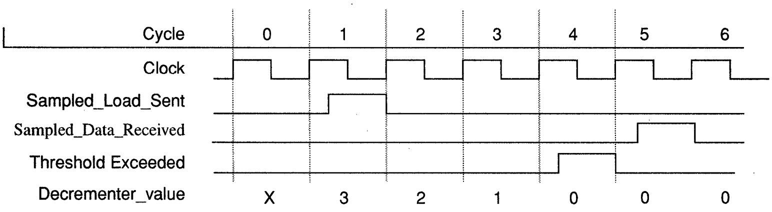


Figure 10-4. Block Diagram of the Delay Thresholding Mechanism for Store Instructions



NOTE: Threshold=3

Figure 10-5. SampLe Timing Diagram of the Thresholding Mechanism (Threshold Not Exceeded)



NOTE: Threshold=3

Figure 10-6. Sample Timing Diagram of the Thresholding Mechanism (Threshold Exceeded)

10.6 The Counters

The performance monitor counters contain the total number of times the software selectable monitored event has occurred since the performance monitor was enabled for counting. Each counter consists of a 32 bit register. These counters can be initialized to zero or any other value via a "mtspr" instruction. Each counter can monitor only one event at a time. The behavior of the counters is directly controlled by the Monitor Mode Control Registers (MMCR0 and MMCR1). To take advantage of the different counting and interrupt features controlled by the MMCRs the counters should not be initialized to a negative number. If a negative number is detected in a counter the performance monitor will take the action dictated by the bits in the MMCRs (i.e. Generate a Performance Monitor Exception request). These registers can only be modified in privilege state but they can be read in problem or privilege state.

NOTE: The performance monitor exception request could be generated or the counters could be disabled several cycles after the count became negative.

10.6.1 Enabling the Counters

The counters can be enabled to count based on different conditions that are controlled by the MMCRs. Once an event is selected to be monitored by a PMC the default is for that counter to be enabled and to count under all scenarios. A combination of MMCR0 bits can be used to control in which scenarios the counters will be enable to count. MMCR0[0] forces the counters to be disabled, MMCR0[1:4] selects the state in which the counters will be disabled and MMCR0[5:9 and 16:18] control when and the order in which the counters will be enabled. Some of the possible scenarios that can be achieved involve any combination of an enable and a disable scenario listed below.

Enable counting scenarios:

- Enable only PMC1.
- Enable only PMC2-8.
- Enable PMC1-8 at the same time.
- Enable PMC1 first and after it becomes negative enable PMC2-8.
- Wait for a Time Base transition; once it is detected enable only PMC1.
- Wait for a Time Base transition; once it is detected enable only PMC2-8.
- Wait for a Time Base transition; once it is detected enable PMC1 and PMC2-8.
- Wait for a Time Base transition; once it is detected enable PMC1 and once it becomes negative enable PMC2-8.

Disable counting scenarios:

- Disable unconditionally.
- Disable PMC1 only.
- Disable PMC2-8 only.
- Disable counting while in Privileged state.
- Disable counting while in Problem state.
- Disable counting if the process is marked.
- Disable counting if the process is not marked.
- Disable counting while in Privileged state and executing a marked process.
- Disable counting while in Privileged state and not executing a marked process.
- Disable counting while in Problem state and executing a marked process.
- Disable counting while in Problem state and not executing a marked process.

10.6.2 Performance Monitor Mark Bit

The performance monitor mark (PMM) bit is located in the MSR[61]. The purpose of this bit is to mark specific processes and use it in conjunction with the MMCR0[3–4], FCM0 and FCM1. With the combination of these bits it is possible to control the processes in which the performance monitor will be enabled or disabled.

10.6.3 Selecting the Events to be Monitored

The performance monitor events to be monitored can be selected by properly setting the MMCR0[19–31] and MMCR1[0–28]. There are 7 bits associated with the PMC1 and 6 bits associated with the PMC2. In Table 10-5 through Table 10-12 a correlation is established between each counter, the events to be monitored and the select pattern required in the MMCRs for the desired selection.

NOTE: If the select pattern specified in the corresponding MMCRs bits is not specified in the following tables the contents of the associated PMC will be undefined.

Table 10-5. PMC1 Monitored Events and Selected Bit Patterns

MMCR0[19–25]	Unit	Event Description
0x00	PERFMON	Processor cycles.
0x01	IFU	Number of instructions completed.
0x02	PERFMON	Time Base selected bit transition from zero to one.
0x03	PERFMON	Number of instructions dispatched.
0x04	IFU	Number of load instructions completed.
0x05	IPU	L1 instruction cache miss.

Table 10-5. PMC1 Monitored Events and Selected Bit Patterns (Continued)

MMCR0[19–25]	Unit	Event Description
0x06	DCACHE	A Load miss occurred in L1.
0x07	PERFMON	Threshold exceeded (loads with no L2 intervention)
0x08	DCACHE	Data cache EPAT miss.
0x09	PERFMON	Threshold Exceeded (Stores with No L2 Intervention)
0x0A	BIU	A Read-Burst missed the L2 and another bus device has modified data.
0x0B	IPU	L1 instruction cache IEPAT miss.
0x0C	IPU	Brought/wrote a line into the ICACHE and used it.
0x0D	DCACHE	Data cache detected an offset hit.
0x0E	PERFMON	Number of instructions deleted due to global cancel.
0x0F	PERFMON	Chaining the counters in history mode. (PMC1 to PMC8)
0x12	BIU	A master-generated store operation is retried.
0x14	IFU	MSR External Interrupt Enable bit, MSR(EE), is off
0x15	IPU	Branch unit idle.
0x16	IFU	A Single instruction serialization class instruction is in execution (Counts the total number of cycles this condition is detected.)
0x17	FPU	The FPU status and control register instructions.
0x18	BIU	One store buffer is in use.
0x19	BIU	A snooped operation cleaned data from the L2.
0x1A	PERFMON	Number of stores in the completion buffer.
0x1B	IPU	The Link Register stack is full.
0x1C	IPU	A conditional branch was resolved at dispatch.
0x1D	PERFMON	Number of loads in the completion buffer.
0x1E	PERFMON	Number of entries in the completion buffer.
0x1F	LDST	The Finished Store Queue (FSQ) is full.
0x51	DCACHE	Data cache and Instruction cache SLB miss occurred.
0x53	DCACHE	Data cache and instruction cache TLB miss.
0x56	IFU	A Single instruction serialization class instruction is in execution (Counts the number of times this condition is detected.)

Table 10-6. PMC2 Monitored Events and Selected Bit Patterns

MMCR0[26–31]	Unit	Event Description
0x00	IFU	Number of instructions completed.
0x01	PERFMON	Processor cycles.
0x02	PERFMON	Time Base selected bit transition from zero to one.
0x03	PERFMON	Number of instructions dispatched.
0x05	DCACHE	Data cache store address lookup.
0x06	BIU	A Sampled Read-Burst generated an L2 miss.
0x08	IPU	A conditional branch was predicted.
0x09	DCACHE	Store miss occurred in L1.
0x0A	PERFMON	Threshold Exceeded (Loads with L2 Intervention)
0x0B	BIU	A Read-with-Intent-to-Modify (RWITM) generated an L2 access.
0x0C	PERFMON	Threshold Exceeded (Stores with L2 Intervention)
0x0D	IFU	A Store conditional instruction failed to execute successfully
0x0E	BIU	A master-generated non-burst store operation is stalled waiting for a store buffer.
0x0F	PERFMON	Chaining the counters in history mode. (PMC2 to PMC1)
0x10	BIU	A Sampled Read-Burst missed the L2 and another bus device has modified data.
0x11	FXU	The Complex Integer Unit does not have a valid instruction to execute.
0x12	IFU	A system call interrupt was taken.
0x14	BIU	Two store buffers are in use.
0x15	BIU	A master-generated load operation is not retried.
0x16	LDST	A larx instruction has finished execution.
0x18	LDST	A sample store instruction was scheduled for execution.
0x19	IPU	The instruction buffer is empty this cycle.
0x1C	BIU	A snooped operation generated a push or an intervention.
0x1D	BIU	A master-generated store operation is loaded into the store buffer.
0x33	DCACHE	Data cache and Instruction cache SLB miss occurred.

Table 10-7. PMC3 Monitored Events and Selected Bit Patterns

MMCR1[0–4]	Unit	Event Description
0x00	IPU	Brought/wrote a line into the ICACHE and used it.
0x01	PERFMON	Processor cycles.
0x02	IFU	Number of instructions completed.
0x03	PERFMON	Time Base selected bit transition from zero to one.
0x04	IFU	Number of instructions dispatched.
0x05	DCACHE	A Load miss occurred in L1.
0x06	BIU	A sampled read-with-intent-to-modify (RWITM) generated an L2 miss.
0x07	IPU	The branch queue is full.
0x08	BIU	A sampled read-with-intent-to-modify (RWITM) missed the L2 and another bus device has modified data.
0x09	IFU	A store instruction was completed.
0x0A	DCACHE	A sampled store was completed.
0x0B	IFU	A load instruction is the next instruction to complete.
0x0C	BIU	A read-with-intent-to-modify (RWITM) generated an L2 miss.
0x0D	BIU	A sampled read-burst generated an L2 access.
0x0E	BIU	A master-generated non-burst store operation is stalled waiting for a store buffer.
0x0F	PERFMON	Chaining the counters in history mode. (PMC3 to PMC2)
0x10	LDST	A double word unaligned store was scheduled
0x11	BIU	A master-generated store operation is loaded into the store buffer.
0x13	BIU	Three store buffers are in use.
0x14	BIU	A master-generated Store Conditional (STCX) is cancelled.
0x15	BIU	A snooped operation generated a transition in the L2 from Exclusive or Shared to Invalid.
0x16	FPU	The FPU divide instructions.
0x18	IFU	I/O interrupts detected.

Table 10-8. PMC4 Monitored Events and Selected Bit Patterns

MMCR1[5– 9]	Unit	Event Description
0x00	IPU	L1 instruction cache IEPAT miss.
0x01	PERFMON	Processor cycles.
0x02	IFU	Number of instructions completed.
0x03	PERFMON	Time Base selected bit transition from zero to one.
0x04	IFU	Number of instructions dispatched.
0x05	IFU	Number of load instructions completed.
0x07	LDST	The load/store scheduled a sampled load instruction
0x08	BIU	A sampled read-with-intent-to-modify (RWITM) generated an L2 access.
0x09	LDST	The load/store received data from the data cache.
0x0A	BIU	A read-with-intent-to-modify (RWITM) missed the L2 and another bus device has modified data.
0x0B	DCACHE	Data cache sync request was made to the BIU.
0x0C	IFU	Global cancel due to a load or store instruction address conflict.
0x0D	FXU	The multi-cycle integer unit pipeline is busy with a valid instruction.
0x0E	BIU	A master-generated store operation is not retried.
0x0F	PERFMON	Chaining the counters in history mode. (PMC4 to PMC3)
0x10	DCACHE	Data cache detected an aliased hit.
0x11	FXU	The simple integer unit 1 does not have a valid instruction to execute.
0x12	LDST	A double word unaligned load was scheduled.
0x13	PERFMON	Completion stalled on a load operation.
0x14	BIU	A master-generated Bus operation received an ARESPIN Retry.
0x16	IPU	Branch completed.
0x17	IPU	The dispatch buffer is empty this cycle.
0x18	IPU	Link Register stack error.
0x19	IPU	The condition register logical unit produced a result.
0x1B	BIU	A snooped operation cleaned data from the L1.

Table 10-9. PMC5 Monitored Events and Selected Bit Patterns

MMCR1[10 –14]	Unit	Event Description
0x00	DCACHE	Data cache EPAT miss.
0x01	IPU	The instruction cache was accessed and a fetch block was fetched.
0x02	PERFMON	No instructions completed.
0x04	BIU	A read-burst (RB) generated an L2 access.
0x05	FPU	The FPU finished the execution of an instruction.
0x06	LDST	The load/store reservation stations are empty.
0x07	IPU	BTAC hit.
0x08	LDST	Completed Store Queue (CSQ) is full.
0x09	BIU	A master-generated store operation is stalled waiting for a store buffer.
0x0A	BIU	A snooped operation generated a transition in the L2 from Modified to Invalid.
0x0B	FPU	The FPU convert and round instructions.
0x0C	PERFMON	Processor cycles.
0x0D	BIU	A master-generated Bus operation received an ASTATIN Retry.
0x0F	PERFMON	Chaining the counters in history mode. (PMC5 to PMC4)

Table 10-10. PMC6 Monitored Events and Selected Bit Patterns

MMCR1[15–19]	Unit	Event Description
0x01	DCACHE	Store hit occurred in L1.
0x02	FXU	The Multi-cycle integer unit finished the execution of an instruction.
0x03	IPU	A BTAC miss was detected.
0x04	BIU	An instruction fetch generated an L2 miss.
0x05	IPU	A conditional branch was dispatched.
0x06	LDST	The Load queue is full.
0x08	BIU	A snooped operation generated a push or an intervention.
0x09	IFU	The MSR(EE) bit is off and an external interrupt is pending.
0x0A	BIU	A master-generated load operation is retried.
0x0B	FPU	The FPU move instructions and the select instruction.
0x0C	PERFMON	Processor cycles.
0x0D	BIU	A snooped operation accessed the L2.
0x0E	BIU	A snooped operation generated a transition in the L2 from Exclusive to Shared.
0x0F	PERFMON	Chaining the counters in history mode. (PMC6 to PMC5)

Table 10-11. PMC7 Monitored Events and Selected Bit Patterns

MMCR1[20–24]	Unit	Event Description
0x00	IPU	L1 instruction cache miss.
0x01	FXU	The simple integer unit 0 finished the execution of an instruction.
0x02	IPU	A branch was dispatched (any).
0x03	IFU	Global cancel due to a branch guessed wrong.
0x04	BIU	A bus operation was snooped.
0x06	PERFMON	No instructions dispatched.
0x07	FXU	The simple integer unit 0 does not have a valid instruction to execute.
0x0A	PERFMON	A store instruction was dispatched.
0x0B	PERFMON	Processor cycles.
0x0F	PERFMON	Chaining the counters in history mode. (PMC7 to PMC6)

Table 10-12. PMC8 Monitored Events and Selected Bit Patterns

MMCR1[25–28]	Unit	Event Description
0x1	BIU	A snooped operation hit the L2.
0x2	BIU	A Read-Burst generated an L2 miss.
0x3	PERFMON	A store conditional instruction executed successfully.
0x4	FXU	The Simple Integer unit 1 finished the execution of an instruction.
0x5	BIU	A bus operation was ASTATOUT Retried.
0x7	IPU	Prefetch bad.
0x8	PERFMON	Completion stalled on a store operation.
0xA	PERFMON	A load instruction was dispatched.
0xB	IFU	Misaligned data interrupt
0xC	PERFMON	Processor cycles.
0xF	PERFMON	Chaining the counters in history mode. (PMC8 to PMC7)

10.7 Detailed Description of Events

A detailed description of performance monitor events follows:

- Instruction Flow Unit Events

1. MSR External Interrupt Enable bit, MSR(EE), is off. (PMC1 Event 0x14)

   The MSR(EE) bit is disabled. For every processor cycle in which this bit is off, this event will be asserted.

2. A Single instruction serialization class instruction is in execution (PMC1 Event 0x16 and 0x56)

   Single instruction serialization class instructions (SIS class instructions) will only execute when they are the oldest in the machine and will also halt the dispatch of instructions which follow them. This event will be asserted for every processor cycle from the cycle after the instruction gets dispatched to the cycle after the instruction finishes. See Section 6.4.2, “Execution Serialization.”

3. A Load instruction is the next instruction to complete. (PMC3 Event 0x0B)

   A load instruction is the first instruction pending completion. The **dcbt** and **dcbtst** instructions are considered load instructions in the 620. For every processor cycle in which a load instruction is the first instruction pending completion, this event will be asserted for one processor cycle.

4. Number of instructions completed. (PMC1 Event 0x01, PMC2 Event 0x00, PMC3 and PMC4 Event 0x02)

   Number of instructions completed on a processor cycle. This number has a minimum value of 0 (no instructions completed) and a maximum value of 4 (4 instructions were completed). This event is asserted for one processor cycle.

5. Number of load instructions completed. (PMC1 Event 0x04 and PMC4 Event 0x05)

   Number of load instructions completed on a processor cycle. This number has a minimum value of 0 (no load instructions completed) and a maximum value of 4 (4 load instructions completed). The **dcbt** and **dcbtst** instructions are considered load instructions in the 620. This event is asserted for one processor cycle.

6. Number of instructions dispatched. (PMC3 and PMC4 Event 0x04)

   Number of instructions dispatched on a processor cycle. This number has a minimum value of 0 (no instructions dispatched) and a maximum value of 4 (4 instructions dispatched). This event is asserted for one processor cycle.

7. A store instruction was completed. (PMC3 Event 0x09)

   Only one store instruction is allowed to complete per processor cycle. This event indicates that a store instruction was completed. This event is asserted for one processor cycle.

8. Global cancel due to a load or store instruction address conflict.(PMC4 Event 0x0C)

   This event indicates that there was a global cancel due to a load instruction potentially bypassing a load or store instruction to the same effective address. For an instruction stream with a load instruction before a load or store instruction to the same effective address, upon completion of the load or store instruction a load bypass error will be generated. Due to this bypass error, a global cancel will be generated which cancels the execution of instructions after the load or store instruction, and causes them to be re-fetched and re-executed.

9. Global cancel due to a branch guessed wrong. (PMC7 Event 0x03)

   This event indicates that a branch has been guessed wrong and the incorrect path was taken. A global cancel is generated to cancel the instructions executed through the incorrect path and refetch the instructions on the correct path. This event is asserted for one processor cycle.

10. A store conditional instruction failed to execute successfully. (PMC2 Event 0x0D)

    A **stwcx.** or **stdcx.** instruction has completed and the resulting CR bit (bit 2 of CR field 0) was a 0. This indicates that the store conditional was executed unsuccessfully. This event is asserted for one processor cycle.

11. A store conditional instruction executed successfully. (PMC8 Event 0x03)

    A stwcx. or stdcx. instruction has completed and the resulting CR bit (bit 2 of CR field 0) was a 1. This indicates that the store conditional was executed successfully. This event is asserted for one processor cycle.

12. A load instruction was dispatched. (PMC8 Event 0x0A)

    This event indicates that a load instruction was dispatched. This includes speculatively dispatched instructions which may be later cancelled. This event is active for one processor cycle.

13. A store instruction was dispatched. (PMC7 Event 0x0A)

    This event indicates that a store instruction was dispatched. This includes speculatively dispatched instructions which may be later cancelled. This event is active for one processor cycle.

14. System call interrupt (PMC2 Event 0x12)

    This event indicates that a system call interrupt was taken. This event is asserted for one processor cycle.

15. EE bit is off and an external interrupt is pending. (PMC6 Event 0x09)

    Interrupts like decrementer, external, performance and system management interrupts are masked by the EE bit. If the EE bit is disabled these interrupts will be pending until EE is enabled. This event will be asserted for every processor cycle in which any of the interrupts above is pending while EE is off.

16. I/O interrupts detected. (PMC3 Event 0x18)

    This event indicates that an external or a system management interrupt was taken. This event is asserted for one processor cycle.

17. Misaligned data interrupt (PMC8 Event 0x0B)

    This event indicates that there was an exception due to misaligned data. This event is asserted for one processor cycle.

- Instruction Processing Unit Events

    1. The instruction cache was accessed and a fetch block was fetched. (PMC5 Event 0x01)

       A fetch block was accessed by the branch unit. The fetch block could have been fetched from the ICACHE (including the cache-reload-buffer), the L2 cache or system memory. This event is active for one clock cycle.

    2. L1 instruction cache miss. (PMC7 Event 0x00;PMC1 Event 0x05)

       The ICACHE was accessed, with valid translation if instruction side translation is enabled, and the instructions were not available in the instruction cache. A request was made by the ICACHE to the BIU for instructions. This event is active for one clock cycle.

3. L1 instruction cache IEPAT miss. (PMC4 Event 0x00;PMC1 Event 0x0B)

   The IEPAT did not contain an entry for the requested instruction address. A table walk request was sent to the ICACHE and the IEPAT was loaded with translation data from a table walk or from a segment register entry. This event is active for one clock cycle.

4. A BTAC miss was detected. (PMC6 Event 0x03)

   An entry for the fetch address was not found in the BTAC. This event is active for one clock cycle.

5. A branch was dispatched (any). (PMC7 Event 0x02)

   A branch instruction was dispatched. This includes speculatively dispatched instructions which may be later cancelled. This event is active for one clock cycle.

6. The branch queue is full. (PMC3 Event 0x07)

   The branch queue contained four guessed branches on the last processor clock. Every processor clock which occurs while the branch queue is full will cause another event signal. This event is active for one clock cycle.

7. A conditional branch was predicted. (PMC2 Event 0x08)

   A conditional branch was predicted at dispatch. The branch may be cancelled due to it or a previous branch being mispredicted. This event is active for one clock cycle.

8. A conditional branch was dispatched. (PMC6 Event 0x05)

   A conditional branch instruction was dispatched. This could include speculatively dispatched instructions which may be later cancelled. This event is active for one clock cycle.

9. Brought/wrote a line into the ICACHE and used it. (PMC3 Event 0x00, PMC1 Event 0x0C)

   A line was brought into the CRB and one or more instructions from the line were dispatched. This event is active for one clock cycle.

10. BTAC hit. (PMC5 Event 0x07)

    The branch target address cache had an entry that corresponded to the instruction fetch address. This signal is active for one clock cycle.

11. Prefetch bad. (PMC8 Event 0x07)

    The prefetch address was not used. A fetch correction had to be made at dispatch. This event is active for one clock cycle.

12. Branch unit idle. (PMC1 Event 0x15)

    The queue is empty, so the branch unit is idle.This event is active for one clock cycle.

13. Branch completed. (PMC4 Event 0x16)

A branch completed this cycle. This event is active for one clock cycle.

14. The dispatch buffer is empty this cycle. (PMC4 Event 0x17)

The dispatch buffer is empty this cycle. This event is active for one clock cycle.

15. Link Register stack error. (PMC4 Event 0x18)

The fetch address from the Link Register stack was not correct. A fetch correction had to be made. This event is active for one clock cycle.

16. The instruction buffer is empty this cycle. (PMC2 Event 0x19)

The instruction buffer is empty this cycle. This event is active for one clock cycle.

17. The condition register logical unit produced a result. (PMC4 Event 0x19)

The condition register logical unit finished an instruction. This event is active for one clock cycle.

18. The Link Register stack is full. (PMC1 Event 0x1B)

The Link Register stack holds 8 fetch addresses. This event is active for one clock cycle.

19. A conditional branch was resolved at dispatch. (PMC1 Event 0x1C)

The conditional branch was dispatched and was not guessed. This event is active for one clock cycle.

- Data Cache Events

1. Store hit occurred in L1. (PMC6 Event 0x01)

   Asserts for one cycle, once for each writeback store that is taken from the CSQ. It has nothing to do with the number of translation lookups done on behalf of stores.

2. Store miss occurred in L1. (PMC2 Event 0x09)

   Asserts from one cycle when a writeback store misses the cache. It may assert more than once for a given store if that store's cache allocate gets retried by the BIU due to collision detection.

3. Data cache and instruction cache TLB miss. (PMC1 Event 0x53)

   A TLB miss was detected in the data cache or the instruction cache. This event is asserted for two cycles for each TLB miss detected but this 2 cycle pulse will be counted as one occurrence.

4. A Load miss occurred in L1. (PMC1 Event 0x06, PMC3 Event 0x05)

   Asserts one cycle for each cache allocate on behalf of a load miss. This event will also assert on aliased hits since the DCACHE go to the BIU before performing second cycle lookups. (See "Data cache offset hit" below)

5. A sampled store was completed. (PMC3 Event 0x0A)

   Asserts for one cycle for each sampled store that completes.

6. Data cache EPAT miss. (PMC1 Event 0x08, PMC5 Event 0x00)

   Asserts for one cycle to indicate a DCACHE EPAT miss.

7. Data cache and Instruction cache SLB miss occurred. (PMC1 Event 0x51, PMC2 Event 0x33)

   This event will be asserted whenever a data cache or instruction cache SLB miss occurs. This event will be asserted for more than 1 cycle pulse but this pulse will be counted as one occurrence.

8. Data cache detected an aliased hit. (PMC4 Event 0x10)

   Asserts for one cycle for each aliased hit.

9. Data cache detected an offset hit. (PMC1 Event 0x0D)

   Asserts for one cycle for each offset hit. The total count of aliased hits and offset hits will indicate the number of times that a load request encounters a multi-cycle hit scenario. Substracting this total from the number of "Load misses" count will give a true count of cache load misses.

10. Data cache store address lookup. (PMC2 Event 0x05)

    Asserts for one cycle each time the DCACHE returns an address translation to the Load Store unit on behalf of a store instruction.

11. Data cache sync request was made to the BIU. (PMC4 Event 0x0B)

    This event is asserted when a SYNC request to the BIU is made, this could take several cycles. This event will count each of the cycles.

- Bus Interface Unit Events

  — L2-Related Data for Master-Generated Events

  1. A Read-Burst (RB) generated an L2 access. (PMC5 Event 0x04)

     Asserts for one cycle for each RB from the DCache that generated an L2 access. The RB is associated with either a normal data request or a table walk.

  2. A Read-Burst generated an L2 miss. (PMC8 Event 0x02)

     Asserts for one cycle for each RB from the DCache that generated an L2 miss.

  3. A Sampled Read-Burst generated an L2 access. (PMC3 Event 0X0D)

     Asserts for one cycle for each Sampled RB from the DCache that generated an L2 access. The Sampled RB is associated with a normal data request, not a table walk.

  4. A Sampled Read-Burst generated an L2 miss. (PMC2 Event 0x06)

     Asserts for one cycle for each Sampled RB from the DCache that generated an L2 miss.

5. A Read-with-Intent-to-Modify (RWITM) generated an L2 access. (PMC2 Event 0x0B)

   Asserts for one cycle for each RWITM from the DCache that generated an L2 access. A RWITM is the result of a cacheable store miss or touch-for-store miss in the DCache.

6. A Read-with-Intent-to-Modify (RWITM) generated an L2 miss. (PMC3 Event 0x0C)

   Asserts for one cycle for each RWITM from the DCache that generated an L2 miss.

7. A Sampled Read-with-Intent-to-Modify (RWITM) generated an L2 access. (PMC4 Event 0x08)

   Asserts for one cycle for each Sampled RWITM from the DCache that generated an L2 access.

8. A Sampled Read-with-Intent-to-Modify (RWITM) generated an L2 miss. (PMC3 Event 0x06)

   Asserts for one cycle for each Sampled RWITM from the DCache that generated an L2 miss.

9. An instruction fetch generated an L2 miss. (PMC6 Event 0x04)

   Asserts for one cycle for each instruction fetch that generated an L2 miss.

10. A Read-Burst missed the L2 and another bus device has modified data. (PMC1 Event 0x0A)

    Asserts for one cycle for each RB from the DCache that missed the L2 and found modified data in another bus device, as opposed to main memory.

11. A Sampled Read-Burst missed the L2 and another bus device has modified data. (PMC2 Event 0x10)

    Asserts for one cycle for each Sampled RB from the DCache that missed the L2 and found modified data in another bus device, as opposed to main memory.

12. A Read-with-Intent-to-Modify (RWITM) missed the L2 and another bus device has modified data. (PMC4 Event 0x0A)

    Asserts for one cycle for each RWITM from the DCache that missed the L2 and found modified data in another bus device, as opposed to main memory.

13. A Sampled Read-with-Intent-to-Modify (RWITM) missed the L2 and another bus device has modified data. (PMC3 Event 0x08)

    Asserts for one cycle for each Sampled RWITM from the DCache that missed the L2 and found modified data in another bus device, as opposed to main memory.

— Bus-Related Data for Master-Generated Events

1. A master-generated Bus operation received an ASTATIN Retry. (PMC5 Event 0x0D)

   Asserts for one cycle for any master-generated Bus operation that received an ASTATIN Retry response. This event includes all data loads, all instruction fetches and all stores.

2. A master-generated Bus operation received an ARESPIN Retry. (PMC4 Event 0x14)

   Asserts for one cycle for any master-generated Bus operation that received an ARESPIN Retry response. This event includes all data loads, all instruction fetches and all stores.

3. A master-generated load operation is retried. (PMC6 Event 0x0A)

   Asserts for one cycle for any master-generated load operation that received either an ASTATIN Retry or an ARESPIN Retry response. This event includes all data loads and all instruction fetches.

4. A master-generated load operation is not retried. (PMC2 Event 0x15)

   Asserts for one cycle for any master-generated load operation that received neither an ASTATIN Retry nor an ARESPIN Retry response. This event includes all data loads and all instruction fetches.

5. A master-generated store operation is retried. (PMC1 Event 0x12)

   Asserts for one cycle for any master-generated store operation that received either an ASTATIN Retry or an ARESPIN Retry response. This event includes all stores.

6. A master-generated store operation is not retried. (PMC4 Event 0x0E)

   Asserts for one cycle for any master-generated store operation that received neither an ASTATIN Retry nor an ARESPIN Retry response. This event includes all stores.

7. A master-generated store operation is loaded into the store buffer. (PMC2 Event 0x1D and PMC3 Event 0x11)

   Asserts for one cycle for any master-generated store operation that entered the store buffer.

8. A master-generated store operation is stalled waiting for a store buffer. (PMC5 Event 0x09)

   Asserts for every cycle that a master-generated store operation is stalled waiting for a store buffer. This event includes all stores.

9. A master-generated non-burst store operation is stalled waiting for a store buffer. (PMC2 Event 0x0E and PMC3 Event 0x0E)

   Asserts for every cycle that a master-generated non-burst store operation is stalled waiting for a store buffer. This event includes write-through stores, cache-inhibited stores, External Control Out operations and PIO Store operations.

10. Three store buffers are in use. (PMC3 Event 0x13)

   Asserts for every cycle that all three store buffers are valid.

11. Two store buffers are in use. (PMC2 Event 0x14)

   Asserts for every cycle that any two of three store buffers are valid.

12. One store buffer is in use. (PMC1 Event 0x18)

   Asserts for every cycle that one of three store buffers is valid.

13. A master-generated Store Conditional (STCX) is cancelled. (PMC3 Event 0x14)

   Asserts for one cycle for any master-generated STCX that received an ARESPIN Retry. Self-retry cases where the Master generates ARESP0UT Retry are included in this event.

— Snooper-Generated Events

1. A bus operation was snooped. (PMC7 Event 0x04)

   Asserts for one cycle for each snooped operation. This event includes all memory-coherent operations where the M-bit on the Bus is set to 1. In addition, the event includes non-memory-coherent operations TLBIE, TLBSYNC and SYNC.

2. A bus operation was ASTATOUT Retried. (PMC8 Event 0x05)

   Asserts for one cycle for each snooped operation that generated an ASTATOUT Retry. ASTATOUT Retry is generated when all four snoop buffers are full, when both state machines are full, when a snooped TLBIE encounters an active TLBIE or when a snooped TLBSYNC encounters an active TLBIE. Note that multiple conditions may occur in the same cycle.

3. A snooped operation accessed the L2. (PMC6 Event 0x0D)

   Asserts for one cycle for each snooped operation that accessed the L2. This includes only the memory-coherent operations (M=1) which check L2 state. This event asserts regardless of whether the L2 is enabled or disabled. This event is not signalled when the snoop buffers are full and the operation is ASTATOUT Retried.

4. A snooped operation hit the L2. (PMC8 Event 0x01)

   Asserts for one cycle for each snooped operation that accessed and hit the L2.

5. A snooped operation generated a push or an intervention. (PMC2 Event 0x1C and PMC6 Event 0x08)

   Asserts for one cycle for each snooped operation that generated either a push or an intervention.

6. A snooped operation cleaned data from the L1. (PMC4 Event 0x1B)

   A snooped operation such as Clean or Read-Burst generated a push or intervention from the L1 and also changed the L2 from Modified to Exclusive or Shared.

7. A snooped operation cleaned data from the L2. (PMC1 Event 0x19)

   A snooped operation such as Clean or Read-Burst generated a push or intervention from the L2 and changed the L2 from Modified to Exclusive or Shared.

8. A snooped operation generated a transition in the L2 from Modified to Invalid.
   (PMC5 Event 0x0A)

   A snooped operation generated a transition in the L2 from Modified to Invalid. This event includes operations such as DKill which simply invalidate the line and operations such as RWITM which generate a push or intervention and then invalidate the line.

9. A snooped operation generated a transition in the L2 from Exclusive to Shared.
   (PMC6 Event 0x0E)

   A snooped operation such as Read-Burst generated a transition in the L2 from Exclusive to Shared.

10. A snooped operation generated a transition in the L2 from Exclusive or Shared to Invalid. (PMC3 Event 0x15)

    A snooped operation such as Read-with-Intent-to-Modify generated a transition in the L2 from Exclusive or Shared to Invalid.

- Floating Point Unit Events

  1. The FPU finished the execution of an instruction. PMC5(0x05)

     Finish is generated for all valid instructions sent to the FPU regardless of whether the instruction updates an architected register or produces a valid result. Since the FPU is capable of finishing one instruction per cycle the finish signal may stay active for multiple cycles.

  2. The FPU move instructions and the select instruction. PMC6(0x0B)

     Execution of the FPU move instructions: fmr(.), fneg(.), fads(.), fnabs(.) and the FPU select instruction fsel(.) are counted.

3. The FPU convert and round instructions. PMC5(0x0B)

   Execution of the FPU convert instructions: fctid(.), fctidz(.), fctiw(.), fctiwz(.), fcfid(.) and the round to single precision are counted.

4. The FPU divide instructions. PMC3(0x16)

   Execution of the FPU divide instructions: fdiv(.), fdivs(.) are counted.

5. The FPU status and control register instructions. PMC1(0x17)

   Execution of the FPU status and control register instructions: mffs(.), mcrfs, mtfsfi(.), mtfsf(.), mtfsb0(.), mtfsb1(.) are counted.

- Integer Unit Events

1. The Simple Integer unit 0 finished the execution of an instruction. (PMC7 Event 0x01)

   This signal goes active for one cycle for each instruction that finishes in the Simple Integer unit 0, single cycle unit, (the 620 chip has two single cycle units).

2. The Simple Integer unit 1 finished the execution of an instruction. (PMC8 Event 0x04)

   This signal goes active for one cycle for each instruction that finishes in the Simple Integer unit 1, single cycle unit, (the 620 chip has two single cycle units).

3. The Multi-cycle integer unit finished the execution of an instruction. (PMC6 Event 0x02)

   This signal goes active for one cycle to indicate that an instruction has finished in the FXU multi-cycle unit.

4. The Multi-cycle integer unit pipeline is busy with a valid instruction. (PMC4 Event 0x0D)

   This event is active any time one or more instructions is executing in one or more of the multi-cycle units (multiply, divide, or logic). This event does not indicate which units are active, more than one unit can be operating at the same time.

5. The Simple Integer unit 0 does not have a valid instruction to execute. (PMC7 Event 0x07)

   This event is active any time that the Simple Integer Unit 0 is not currently executing an instruction and does not have any instructions waiting for execution

6. The Simple Integer unit 1 does not have a valid instruction to execute. (PMC4 Event 0x11)

   This event is active any time that the Simple Integer Unit 1 is not currently executing an instruction and does not have any instructions waiting for execution

7. The Complex Integer Unit does not have a valid instruction to execute. (PMC2 Event 0x11)

   This event is active any time that the Complex Integer Unit is not currently executing an instruction and does not have any instructions waiting for execution.

- Load/Store Unit Events

1. The Load queue is full. (PMC6 Event 0x06)

   The Load queue has a total of five entries. When all the five entries are valid, this event is asserted for a one cycle pulse.

2. The Load/Store reservation stations are empty. (PMC5 Event 0x06)

   The Load/Store unit has three reservation stations. When all three entries are valid, this event is asserted for one cycle.

3. The Load/Store scheduled a sampled load instruction (PMC4 Event 0x07)

   This event is asserted when any sampled load instruction is scheduled for execution. This event is asserted for one cycle.

4. Completed Store Queue (CSQ) is full. (PMC5 Event 0x08)

   The CSQ has a total of six entries. When all the six entries are valid this event is asserted for one cycle.

5. A double word unaligned store was scheduled (PMC3 Event 0x10):

   When a store crosses a double-word address, it is considered unaligned. Such stores are execution serialized. When an unaligned store becomes the oldest instruction on the machine, it is fired for execution, and this event is asserted for one cycle.

6. The Load/Store received data from the data cache. (PMC4 Event 0x09)

   When the Load/Store receives data from the data cache due to a previous load request, this event will be asserted for one cycle.

7. A double word unaligned load was scheduled. (PMC4 Event 0x12)

   When a load crosses a double-word address, it is considered unaligned. Such loads are execution serialized. When an unaligned load becomes the oldest instruction in the machine, it is fired for execution, and this event will be asserted for one cycle.

8. A sample store instruction was scheduled for execution. (PMC2 Event 0x18)

   This event is asserted when a sampled store instruction is scheduled for execution. This event is asserted for one cycle.

9. The Finished Store Queue (FSQ) is full. (PMC1 Event 0x1F)

   The FSQ has a total of five entries. When all five entries are valid this event is asserted for one cycle.

10. A larx instruction has finished execution. (PMC2 Event 0x16)

    A larx is serialized type instruction. This event will be asserted for one cycle.

- Performance Monitor Events

1. Processor cycles. (PMC1 Event 0x00, PMC2 Event 0x01, PMC3 Event 0x01, PMC4 Event 0x01, PMC5 Event 0x0C, PMC6 Event 0x0C, PMC7 Event 0x0B and PMC8 Event 0x0C)

   This event is asserted every cycle.

2. Time Base selected bit transition from zero to one. (PMC1 Event 0x02, PMC2 Event 0x02, PMC3 Event 0x03, PMC4 Event 0x03)

   The Time Base is a 64 bit counter that increments once per Processor Bus clock. The user can select which bit transition to monitor by using the MMCR0[7:8]. This event is asserted for one clock cycle when a transition form zero to a one is detected in the Time Base selected bit.

3. No instructions completed. (PMC5 Event 0x02)

   This event is asserted for one processor cycle when the number of instructions completed is zero.

4. No instructions dispatched. (PMC7 Event 0x06)

   This event is asserted for one processor cycle when the number of instructions dispatched is zero.

5. Completion stalled on a load operation. (PMC4 0x13)

   This event is asserted for one cycle when a load is the next instruction to be completed and no instructions are completed in that cycle.

6. Completion stalled on a store operation. (PMC8 0x08)

   This event is asserted for one cycle when a store is the next instruction to be completed and no instructions are completed in that cycle.

7. Number of instructions deleted due to global cancel. (PMC1 Event 0x0E)

   This event counts the total number of instructions deleted from the completion buffer due to a global cancel. There could be more than one instruction deleted from the completion buffer at a given time. In history mode independent of the number of instructions deleted the value of this signal in a cycle will be zero or one.

8. Number of entries in the completion buffer. (PMC1 Event 0x1E)

   This event counts the total number of entries in the completion buffer every cycle. In history mode independent of the number of entries in the completion buffer the value of this signal in a cycle will be zero or one.

9. Number of loads in the completion buffer. (PMC1 Event 0x1D)

   This event counts the total number of loads in the completion buffer every cycle. In history mode independent of the number of entries in the completion buffer the value of this signal in a cycle will be zero or one.

10. Number of stores in the completion buffer. (PMC1 Event 0x1A)

   This event counts the total number of stores in the completion buffer every cycle. In history mode independent of the number of entries in the completion buffer the value of this signal in a cycle will be zero or one.

11. Chaining the counters in history mode. (PMC1 Event 0x0F, PMC2 Event 0x0F,PMC3 Event 0x0F, PMC4 Event 0x0F, PMC5 Event 0x0F, PMC6 Event 0x0F, PMC7 Event 0x0F, PMC8 Event 0x0F)

   Refer to Section 10.9, "History Mode."

12. Threshold Exceeded (PMC1 Event 0x07, PMC1 Event 0x0A, PMC2 Event 0x0A, PMC2 Event 0x0C)

   The performance monitor can monitor four different types of thresholding items:

   - Loads with no L2 intervention
   - Stores with no L2 intervention
   - Loads with L2 intervention
   - Stores with L2 intervention

   Refer to Section 10.5, "The Thresholder."

10.8 Performance Monitor Interrupt

The conditions responsible for a Performance Monitor interrupt generation are:

- The value of PMC1 becomes negative (PMC1[0] == 1)
- The value of PMCn becomes negative (PMCn[0] == 1, n>1)
- A transition from zero to one is detected in the Time Base bit selected by the TBSEL bits (MMCR0[7:8]).

The interrupt can be disabled for each event independently (MMCR0[9 | 16 | 17]) or to all the events at the same time (MMCR0[6]). The Performance Monitor interrupt is classified as an external interrupt and for this reason it is masked by the MSR[16] or external interrupt enable bit. If a Performance Monitor interrupt is signalled while the external interrupts are disabled, it will be recorded and it will be reported when the external interrupts get enabled.

When a Performance Monitor interrupt is signalled the hardware resets the MMCR0[5], PMXE, bit to disable any other possible Performance Monitor interrupts.

To properly start any subsequent analysis the software is responsible to:

1. Reset the condition that caused the interrupt to be reported.
2. Set the MMCR0[5], PMXE, bit back to one if a Performance Monitor interrupt is desired
3. Because a Time Base transition could have occurred along with an enabled counter negative condition, software should always reset MMCR0[9], TBXE to zero, if its value was previously set to one.

The following is an example on how read Table 10-13. (First "Value" column.)

If {(MSR[16]==1) and (MMCR0[5]==1) and (MMCR0[16]==1) and (PMC1[0]==1)} then the Performance Monitor interrupt is scheduled and will be reported to software.

Table 10-13. Interrupt Generation

Mnemonic	Event	Value			
EE	MSR [16]	1	1	1	1
PMXE	MMCR0 [5]	1	1	1	0
TBXE	MMCR0 [9]	-	-	1	-
PMC1XE	MMCR0 [16]	1	-	-	-
PMCnXE	MMCR0 [17]	-	1	-	-
PMC1 Negative	PMC1 [0]	1	-	-	-
PMCn Negative (n>1)	PMCn [0]	-	1	-	-
Selected Time Base Bit	Time Base	-	-	↑	-

- = Don't Care

↑ = Signal Transition from zero to one.

10.9 History Mode

The PMCs can work in two different modes: as incrementers (counting mode) or as shift left registers (History mode). The default operation mode of the PMCs is the incrementer mode (counting the events). When the PMCs are working in History mode the contents of the registers are shifted by one bit to the left and the event detected is stored in the least significant bit (Event detected = 1, Event not detected = 0). Using this facilities it is possible to establish limited co-relation between the events. In history mode it is only possible to monitor a maximum of one event per PMC at a time. When an event can generate a value greater than one in a cycle (i.e. Number of instructions dispatched) this value will be reflected as a one (value > '1') or as a zero (value = '0'). An event diagram can be generated with the data stored in the PMCs as follows.

At the expense of monitoring less events, the PMCs are allowed to chain in history mode to provide more cycles per event. The counters can only be chained in the following order:

PMC1 chained to PMC8

PMC8 chained to PMC7

PMC7 chained to PMC6

PMC6 chained to PMC5

PMC5 chained to PMC4

PMC4 chained to PMC3

PMC3 chained to PMC2

PMC2 chained to PMC1

When a counter is chained, it becomes the high order 32 bits of a 64 bit register and it can't monitor any event. For example PMC2 chained to PMC1, PMC2 will be the high order 32 bits and PMC1 will be the low order 32 bits and at this time PMC2 will not be able to monitor any other event.

To chain two or more counters see Section 10.6.3, "Selecting the Events to be Monitored."

Figure 10-7 is an example of an event diagram in which PMC1 thru PMC4 are not chained, PMC6 is chained to PMC5 and PMC8 is chained to PMC7.

```
       PMC1                                          0100 1100 1001 1101 0011 0110 1001 1110
       PMC2                                          0011 1001 1010 1101 0011 0101 1010 1100
       PMC3                                          1010 1110 1010 1001 0001 0010 1001 0101
       PMC4                                          0101 0010 0000 1000 0000 1100 1111 0001
  PMC6-PMC5  0000 0000 0000 0000 1111 0000 0000 0000 0010 1010 0101 0011 0011 0110 0000 0001
  PMC8-PMC7  1000 0000 0000 0000 0000 1100 0000 0010 1000 1100 1011 0000 0000 0000 1100 0001
```

Figure 10-7. Example of a History Mode Event Diagram

10.10 Examples

This section provides various examples using the performance monitor.

10.10.1 Using the Thresholder

MMCR0 = 0x001E0441, MMCR1=0x00000000

Threshold value is 30, the counters are monitoring:

PMC1 event 0x11, PMC2 event 0x01, PMC3 thru PMC8 event 0x00.

10.10.2 Enable Interrupt

MMCR0 = 0x041E4441, MMCR1=0x00000000

Same as above but PMCn, n>1, are enable to generate an exception (MMCR0[17] == 1) when any of these PMC's value becomes a negative number. The exception will be propagated because the Performance Monitor is also enable to generate an exception (MMCR0[5]==1). If the exception is reported to the software or not depends on the state of the Enable External interrupt, EE bit, in the MSR.

10.10.3 Disable Counting when Interrupt is Generated

MMCR0 = 0x061E4441, MMCR1=0x00000000

Same as above but the PMCs will stop counting when the Performance Monitor interrupt is generated due to one or more PMCn, n>1, reaching a negative value.

10.10.4 Restriction of Events Counting Due to Processor State

MMCR0 = 0x2e1e4441, MMCR1=0x00000000

Same as above, but the Performance Monitor will only count the events when the instruction belongs to a marked process and the processor is on Privilege state.

10.10.5 PMCs in History and Counting Modes, Restricted Monitor Events and IABR Triggered.

MMCR0 = 0x2e1e4441, MMCR1=0x00000005

Same as above, but the Performance Monitor will trigger, start monitoring the events, only when an IABR match is detected while executing an instruction that belongs to a marked process and the processor is on Privilege state. After the IABR match is detected PMC1 will be counting events, PMCn, n>1, will be in history mode. Note that this run will finish (Exception reported by the Performance Monitor and PMCs disabled) 31 sampling cycles after a PMCn, n>1, detects an event. The reason is that it takes 31 cycles for this sample to propagate to bit zero making the value on that PMC negative.

Chapter 11
Power Management

The PowerPC 620 power saving mechanism is based on globally disabling the internal clock, except for clocks needed for the time base, decrementer, external interrupt detection, and the L2 interface clocks. (Globally disabling the internal clock is a function that is already needed for testability. References to globally disabling clocks will never apply to the time base, decrementer, external interrupt detection, or the L2 interface clocks.) Software initiates the Power Saving mechanism by executing a specific code sequence that includes a MTMSR instruction that sets the POW bit in the MSR. Hardware terminates the Power Saving mechanism by a hardware interrupt that clears that bit.

The 620 Power Saving mechanism supports the system to implement 2 modes, called Doze mode and Nap mode. Both of these modes save power by selectively disabling the internal clocks. No instructions will be dispatched from the core while in either mode.

- Doze mode—The 620 caches are not flushed before enabling the Power Savings Mechanism and the 620 is temporarily woken up by the external $\overline{\text{WAKEUP}}$ signal in order to snoop bus operations. The processor stays awake for the minimum number of cycles or until it completes all pending snoop operations (cache state change, intervention, or push) and then it goes back to sleep.
- Nap mode—All state that can be modified by bus snoop operations is flushed or invalidated prior to enabling the Power Saving mechanism. The $\overline{\text{WAKEUP}}$ pin does not need to be asserted because the 620 does not have any valid internal state that can be snooped. However, if the $\overline{\text{WAKEUP}}$ signal is asserted, the snoop operation will miss in the caches and the 620 will go back to sleep. The procedure for flushing all internal state in the 620 is described by Section 11.4, "Preparing to Enter Nap Mode."

11.1 Power Saving Management Enable—MSR[POW]

The following sections provide information on the power saving management enable bit of the MSR.

11.1.1 Entering Power Saving Mode

The MSR(POW) bit 45, called the power management enable, enables the Power Saving mechanism. The hardware reset state is disabled or 0. When MSR(POW) is disabled, all internal clocks are running and the $\overline{\text{WAKEUP}}$ signal is ignored. When MSR(POW) is asserted all internal clocks are disabled except for clocks needed for the time base, decrementer, external interrupt detection and the L2 interface, so that the 620 can detect the wake-up conditions while most of the logic is asleep. The following code sequence must be used to enable the 620 Power Saving mechanism. When MSR(POW) is set by a mtmsr instruction, the processor will go into the Power Saving mode. The processor clock will stop running after the MSR(POW) bit is set and one cycle after the ISYNC is completed and has generated the global cancel.

Note: MSR(EE) must be set to 1 prior to entering power savings mode. A code sequence to initiate power saving mode follows:

```
loop:
   1. sync          # Ensure that all outstanding bus ops have been completed.
   2. mtmsr (45)    # Enable Power Savings Mechanism (MSR(POW))
   3. isync         # Ensure that mtmsr has been executed.
   4. br loop       # After WAKEUP the processor will go back to sleep.
```

11.1.2 Leaving Power Saving Mode

The 620 will unconditionally leave power saving mode when the MSR(POW) is cleared by any of the following:

- INT
- Decrementer Interrupt
- CHECKSTOP
- HRESET
- SRESET
- MACHINECHECK_IN (Only if HID0(0) - MCI enable is asserted and MSR(ME)=1)
- SYSTEM_MANAGEMENT

The 620 will momentarily wake-up to service one or more snoop operations when the WAKEUP signal is asserted for one BUSCLK. The MSR(POW) is not cleared. Section 11.2, “The WAKEUP Signal.” The 620 will stay awake until the **mtmsr** succeeds, which occurs only when the snooper is idle.

11.2 The WAKEUP Signal

The system arbiter asserts WAKEUP to each 620 in Doze mode according to the following timing diagram. Each 620 that receives WAKEUP in cycle 1 will be awake by cycle 3 to snoop EATS and to sample the address bus operation by cycle 4.

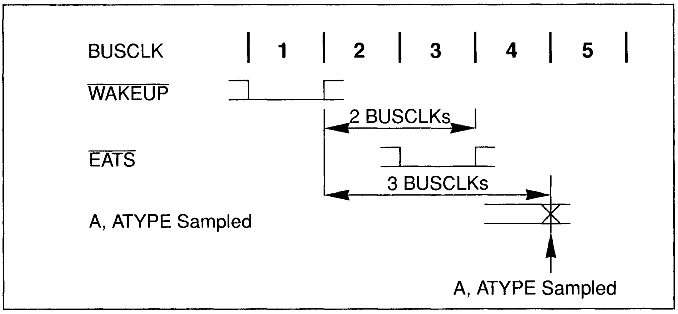


Figure 10-8. WAKEUP Timing Diagram: WAKEUP with Respect to EATS

The WAKEUP signal may be held asserted. This allows the arbiter to keep the doze mode 620(s) awake when the arbiter determines that the work load on the bus is too high to warrant putting processors back into Doze mode. This enables the arbiter to avoid the latency penalty from WAKEUP to EATS when WAKEUP is deasserted and an address bus request is detected by the arbiter.

The WAKEUP signal is listed in the section called Section 11.2, "The WAKEUP Signal."

11.3 External or Decrementer Interrupt Signals

If an external or decrementer interrupt (all doze mode signals except WAKEUP) occurs while the 620 is in process of enabling the Power Saving mechanism or after Power Savings has been asserted, the POW bit will be cleared and the processor will leave the Power Savings mode. The only exception to this rule is—MACHINECHECK_IN with HID0(0)=0. The processor will wake up, but since the Machine Check input is disabled, it will not service the interrupt and will go back to sleep.

The 620's Power Saving mechanism will be tested only under the code sequence in 11.1.

11.4 Preparing to Enter Nap Mode

To enter Nap mode, all the following has to happen before enabling the Power Saving MSR (POW) bit using the code sequence in page 279:

- Invalidate the SLB and MMU tables.
- Invalidate all instruction cache blocks.
- Flush all data cache and L2 blocks.

The WAKEUP signal is not needed, since there is no valid internal state to snoop in the 620. However, if WAKEUP is asserted, the 620 will wake up and snoop the operation in accordance to the timing described above, even though nothing will happen since it will be a miss in the caches. In order to eliminate all state that can be modified by bus snoop operations the following must be done:

- Invalidating the SLB and MMU tables
  - — Execute SLBIA to clear out the ERAT and SLB.
  - — Execute TLBIE to 64 contiguous pages.
- Invalidating the Instruction Cache (IL1)
  - — Reset SPR(HIDO) bit (ICE) to disable the I cache
  - — Set SPR(HIDO) bit (ICEFI) to invalidate the whole I cache.
- Flushing the Data Cache (DL1 and L2)

Since modified data may be resident in the Data cache, it may be necessary to flush out this modified data before entering NAP mode. In addition, all lines must be left in the INVALID state during nap mode to avoid incoherence upon wake up with another processor that may have modified a line.

- L2 Enabled
  - — Execute DCBT to consecutive blocks of data equal to the L2 cache capacity. (This sets up the DL1 and L2 into a known state.)
  - — Execute DCBF to the same cache blocks to force copybacks of any lines that were already resident in L2 when the DCBT's were executed and to leave all lines in the INVALID state.
- L2 Disabled—For systems without an L2, software has to rely on the Data cache LRU algorithm and use touch or load instructions in order to force all modified L1 data out to memory. Make sure that whatever flush routine is used, all lines end up in the INVALID state. If the flush routine itself does not do this, the following sequence will invalidate all lines in the data cache.
  - — Reset SPR(HIDO) bit (DCE) to disable D cache, and
  - — Set SPR(HIDO) bit (DCEFI) to invalidate the whole D cache

Appendix A
PowerPC Instruction Set Listings

This appendix lists the PowerPC 620 microprocessor instruction set as well as PowerPC instructions not implemented in the 620. Instructions are sorted by mnemonic, opcode, function, and form. Also included in this appendix is a quick reference table that contains general information, such as the architecture level, privilege level, and form, and indicates if the instruction is 64-bit and optional.

Note that split fields, that represent the concatenation of sequences from left to right, are shown in lowercase. For more information refer to Chapter 8, "Instruction Set," in *The Programming Environments Manual.*

A.1 Instructions Sorted by Mnemonic

Table A-1 lists the instructions implemented in the 620 in alphabetical order by mnemonic.

Key:

□ Reserved bits

▨ Instruction not implemented in the 620

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
add*x*	31	D	A	B	OE	266	Rc
addc*x*	31	D	A	B	OE	10	Rc
adde*x*	31	D	A	B	OE	138	Rc
addi	14	D	A	SIMM			
addic	12	D	A	SIMM			
addic.	13	D	A	SIMM			
addis	15	D	A	SIMM			
addme*x*	31	D	A	0 0 0 0 0	OE	234	Rc
addze*x*	31	D	A	0 0 0 0 0	OE	202	Rc
and*x*	31	S	A	B	28		Rc
andc*x*	31	S	A	B	60		Rc

Name	0	6 7 8	9	10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29	30	31
andi.	28	S			A	UIMM				
andis.	29	S			A	UIMM				
b_x_	18	LI							AA	LK
bc_x_	16	BO			BI	BD			AA	LK
bcctr_x_	19	BO			BI	0 0 0 0 0	528			LK
bclr_x_	19	BO			BI	0 0 0 0 0	16			LK
cmp	31	crfD	0	L	A	B	0			0
cmpi	11	crfD	0	L	A	SIMM				
cmpl	31	crfD	0	L	A	B	32			0
cmpli	10	crfD	0	L	A	UIMM				
cntlzd_x_[5]	31	S			A	0 0 0 0 0	58			Rc
cntlzw_x_	31	S			A	0 0 0 0 0	26			Rc
crand	19	crbD			crbA	crbB	257			0
crandc	19	crbD			crbA	crbB	129			0
creqv	19	crbD			crbA	crbB	289			0
crnand	19	crbD			crbA	crbB	225			0
crnor	19	crbD			crbA	crbB	33			0
cror	19	crbD			crbA	crbB	449			0
crorc	19	crbD			crbA	crbB	417			0
crxor	19	crbD			crbA	crbB	193			0
dcba[4]	31	0 0 0 0 0			A	B	758			0
dcbf	31	0 0 0 0 0			A	B	86			0
dcbi[1]	31	0 0 0 0 0			A	B	470			0
dcbst	31	0 0 0 0 0			A	B	54			0
dcbt	31	0 0 0 0 0			A	B	278			0
dcbtst	31	0 0 0 0 0			A	B	246			0
dcbz	31	0 0 0 0 0			A	B	1014			0
divd_x_[5]	31	D			A	B	OE	489		Rc
divdu_x_[5]	31	D			A	B	OE	457		Rc
divw_x_	31	D			A	B	OE	491		Rc
divwu_x_	31	D			A	B	OE	459		Rc
eciwx	31	D			A	B	310			0
ecowx	31	S			A	B	438			0

Name	0	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
eieio	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	854		0
eqv*x*	31	S	A	B	284		Rc
extsb*x*	31	S	A	0 0 0 0 0	954		Rc
extsh*x*	31	S	A	0 0 0 0 0	922		Rc
extsw*x*[5]	31	S	A	0 0 0 0 0	986		Rc
fabs*x*	63	D	0 0 0 0 0	B	264		Rc
fadd*x*	63	D	A	B	0 0 0 0 0	21	Rc
fadds*x*	59	D	A	B	0 0 0 0 0	21	Rc
fcfid*x*[5]	63	D	0 0 0 0 0	B	846		Rc
fcmpo	63	crfD 0 0	A	B	32		0
fcmpu	63	crfD 0 0	A	B	0		0
fctid*x*[5]	63	D	0 0 0 0 0	B	814		Rc
fctidz*x*[5]	63	D	0 0 0 0 0	B	815		Rc
fctiw*x*	63	D	0 0 0 0 0	B	14		Rc
fctiwz*x*	63	D	0 0 0 0 0	B	15		Rc
fdiv*x*	63	D	A	B	0 0 0 0 0	18	Rc
fdivs*x*	59	D	A	B	0 0 0 0 0	18	Rc
fmadd*x*	63	D	A	B	C	29	Rc
fmadds*x*	59	D	A	B	C	29	Rc
fmr*x*	63	D	0 0 0 0 0	B	72		Rc
fmsub*x*	63	D	A	B	C	28	Rc
fmsubs*x*	59	D	A	B	C	28	Rc
fmul*x*	63	D	A	0 0 0 0 0	C	25	Rc
fmuls*x*	59	D	A	0 0 0 0 0	C	25	Rc
fnabs*x*	63	D	0 0 0 0 0	B	136		Rc
fneg*x*	63	D	0 0 0 0 0	B	40		Rc
fnmadd*x*	63	D	A	B	C	31	Rc
fnmadds*x*	59	D	A	B	C	31	Rc
fnmsub*x*	63	D	A	B	C	30	Rc
fnmsubs*x*	59	D	A	B	C	30	Rc
fres*x*[4]	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsp*x*	63	D	0 0 0 0 0	B	12		Rc
frsqrte*x*[4]	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc

Name	0	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fsel*x* [4]	63	D	A	B	C	23	Rc
fsqrt*x* [4]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrts*x* [4]	59	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsub*x*	63	D	A	B	0 0 0 0 0	20	Rc
fsubs*x*	59	D	A	B	0 0 0 0 0	20	Rc
icbi	31	0 0 0 0 0	A	B	982		0
isync	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150		0
lbz	34	D	A	d			
lbzu	35	D	A	d			
lbzux	31	D	A	B	119		0
lbzx	31	D	A	B	87		0
ld [5]	58	D	A	ds			0
ldarx [5]	31	D	A	B	84		0
ldu [5]	58	D	A	ds			1
ldux [5]	31	D	A	B	53		0
ldx [5]	31	D	A	B	21		0
lfd	50	D	A	d			
lfdu	51	D	A	d			
lfdux	31	D	A	B	631		0
lfdx	31	D	A	B	599		0
lfs	48	D	A	d			
lfsu	49	D	A	d			
lfsux	31	D	A	B	567		0
lfsx	31	D	A	B	535		0
lha	42	D	A	d			
lhau	43	D	A	d			
lhaux	31	D	A	B	375		0
lhax	31	D	A	B	343		0
lhbrx	31	D	A	B	790		0
lhz	40	D	A	d			
lhzu	41	D	A	d			
lhzux	31	D	A	B	311		0
lhzx	31	D	A	B	279		0

<table>
<tr><th>Name</th><th>0</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>lmw[3]</td><td>46</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="16">d</td></tr>
<tr><td>lswi[3]</td><td>31</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="5">NB</td><td colspan="10">597</td><td>0</td></tr>
<tr><td>lswx[3]</td><td>31</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="5">B</td><td colspan="10">533</td><td>0</td></tr>
<tr><td>lwa[5]</td><td>58</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="14">ds</td><td colspan="2">2</td></tr>
<tr><td>lwarx</td><td>31</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="5">B</td><td colspan="10">20</td><td>0</td></tr>
<tr><td>lwaux[5]</td><td>31</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="5">B</td><td colspan="10">373</td><td>0</td></tr>
<tr><td>lwax[5]</td><td>31</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="5">B</td><td colspan="10">341</td><td>0</td></tr>
<tr><td>lwbrx</td><td>31</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="5">B</td><td colspan="10">534</td><td>0</td></tr>
<tr><td>lwz</td><td>32</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="16">d</td></tr>
<tr><td>lwzu</td><td>33</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="16">d</td></tr>
<tr><td>lwzux</td><td>31</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="5">B</td><td colspan="10">55</td><td>0</td></tr>
<tr><td>lwzx</td><td>31</td><td colspan="5">D</td><td colspan="5">A</td><td colspan="5">B</td><td colspan="10">23</td><td>0</td></tr>
<tr><td>mcrf</td><td>19</td><td colspan="3">crfD</td><td colspan="2">0 0</td><td colspan="3">crfS</td><td colspan="2">0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="10">0</td><td>0</td></tr>
<tr><td>mcrfs</td><td>63</td><td colspan="3">crfD</td><td colspan="2">0 0</td><td colspan="3">crfS</td><td colspan="2">0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="10">64</td><td>0</td></tr>
<tr><td>mcrxr</td><td>31</td><td colspan="3">crfD</td><td colspan="2">0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="10">512</td><td>0</td></tr>
<tr><td>mfcr</td><td>31</td><td colspan="5">D</td><td colspan="5">0 0 0 0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="10">19</td><td>0</td></tr>
<tr><td>mffs<i>x</i></td><td>63</td><td colspan="5">D</td><td colspan="5">0 0 0 0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="10">583</td><td>Rc</td></tr>
<tr><td>mfmsr[1]</td><td>31</td><td colspan="5">D</td><td colspan="5">0 0 0 0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="10">83</td><td>0</td></tr>
<tr><td>mfspr[2]</td><td>31</td><td colspan="5">D</td><td colspan="10">spr</td><td colspan="10">339</td><td>0</td></tr>
<tr><td>mfsr[1, 6]</td><td>31</td><td colspan="5">D</td><td>0</td><td colspan="4">SR</td><td colspan="5">0 0 0 0 0</td><td colspan="10">595</td><td>0</td></tr>
<tr><td>mfsrin[1, 6]</td><td>31</td><td colspan="5">D</td><td colspan="5">0 0 0 0 0</td><td colspan="5">B</td><td colspan="10">659</td><td>0</td></tr>
<tr><td>mftb</td><td>31</td><td colspan="5">D</td><td colspan="10">tbr</td><td colspan="10">371</td><td>0</td></tr>
<tr><td>mtcrf</td><td>31</td><td colspan="5">S</td><td>0</td><td colspan="8">CRM</td><td>0</td><td colspan="10">144</td><td>0</td></tr>
<tr><td>mtfsb0<i>x</i></td><td>63</td><td colspan="5">crbD</td><td colspan="5">0 0 0 0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="10">70</td><td>Rc</td></tr>
<tr><td>mtfsb1<i>x</i></td><td>63</td><td colspan="5">crbD</td><td colspan="5">0 0 0 0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="10">38</td><td>Rc</td></tr>
<tr><td>mtfsf<i>x</i></td><td>63</td><td>0</td><td colspan="8">FM</td><td>0</td><td colspan="5">B</td><td colspan="10">711</td><td>Rc</td></tr>
<tr><td>mtfsfi<i>x</i></td><td>63</td><td colspan="3">crfD</td><td colspan="2">0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="4">IMM</td><td>0</td><td colspan="10">134</td><td>Rc</td></tr>
<tr><td>mtmsr[1, 6]</td><td>31</td><td colspan="5">S</td><td colspan="5">0 0 0 0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="10">146</td><td>0</td></tr>
<tr><td>mtmsrd[1, 5]</td><td>31</td><td colspan="5">S</td><td colspan="5">0 0 0 0 0</td><td colspan="5">0 0 0 0 0</td><td colspan="10">178</td><td>0</td></tr>
<tr><td>mtspr[2]</td><td>31</td><td colspan="5">S</td><td colspan="10">spr</td><td colspan="10">467</td><td>0</td></tr>
<tr><td>mtsr[1, 6]</td><td>31</td><td colspan="5">S</td><td>0</td><td colspan="4">SR</td><td colspan="5">0 0 0 0 0</td><td colspan="10">210</td><td>0</td></tr>
<tr><td>mtsrd[1, 6]</td><td>31</td><td colspan="5">S</td><td>0</td><td colspan="4">SR</td><td colspan="5">0 0 0 0 0</td><td colspan="10">82</td><td>0</td></tr>
<tr><td>mtsrdin[1, 6]</td><td>31</td><td colspan="5">S</td><td colspan="5">0 0 0 0 0</td><td colspan="5">B</td><td colspan="10">114</td><td>0</td></tr>
</table>

Name	0	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25	26	27 28 29	30	31
mtsrin[1, 6]	31	S	00000	B	242					0
mulhd*x*[5]	31	D	A	B	0	73				Rc
mulhdu*x*[5]	31	D	A	B	0	9				Rc
mulhw*x*	31	D	A	B	0	75				Rc
mulhwu*x*	31	D	A	B	0	11				Rc
mulld*x*[5]	31	D	A	B	OE	233				Rc
mulli	7	D	A	SIMM						
mullw*x*	31	D	A	B	OE	235				Rc
nand*x*	31	S	A	B	476					Rc
neg*x*	31	D	A	00000	OE	104				Rc
nor*x*	31	S	A	B	124					Rc
or*x*	31	S	A	B	444					Rc
orc*x*	31	S	A	B	412					Rc
ori	24	S	A	UIMM						
oris	25	S	A	UIMM						
rfi[1, 6]	19	00000	00000	00000	50					0
rfid[1, 5]	19	00000	00000	00000	18					0
rldcl*x*[5]	30	S	A	B	mb			8		Rc
rldcr*x*[5]	30	S	A	B	me			9		Rc
rldic*x*[5]	30	S	A	sh	mb			2	sh	Rc
rldicl*x*[5]	30	S	A	sh	mb			0	sh	Rc
rldicr*x*[5]	30	S	A	sh	me			1	sh	Rc
rldimi*x*[5]	30	S	A	sh	mb			3	sh	Rc
rlwimi*x*	20	S	A	SH	MB		ME			Rc
rlwinm*x*	21	S	A	SH	MB		ME			Rc
rlwnm*x*	23	S	A	B	MB		ME			Rc
sc	17	00000	00000	00000000000000					1	0
slbia[1,4,5]	31	00000	00000	00000	498					0
slbie[1,4,5]	31	00000	00000	B	434					0
sld*x*[5]	31	S	A	B	27					Rc
slw*x*	31	S	A	B	24					Rc
srad*x*[5]	31	S	A	B	794					Rc
sradi*x*[5]	31	S	A	sh	413				sh	Rc

Name	0	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
sraw*x*	31	S	A	B	792	Rc
srawi*x*	31	S	A	SH	824	Rc
srd*x*[5]	31	S	A	B	539	Rc
srw*x*	31	S	A	B	536	Rc
stb	38	S	A	d		
stbu	39	S	A	d		
stbux	31	S	A	B	247	0
stbx	31	S	A	B	215	0
std[5]	62	S	A	ds		0
stdcx.[5]	31	S	A	B	214	1
stdu[5]	62	S	A	ds		1
stdux[5]	31	S	A	B	181	0
stdx[5]	31	S	A	B	149	0
stfd	54	S	A	d		
stfdu	55	S	A	d		
stfdux	31	S	A	B	759	0
stfdx	31	S	A	B	727	0
stfiwx	31	S	A	B	983	0
stfs	52	S	A	d		
stfsu	53	S	A	d		
stfsux	31	S	A	B	695	0
stfsx	31	S	A	B	663	0
sth	44	S	A	d		
sthbrx	31	S	A	B	918	0
sthu	45	S	A	d		
sthux	31	S	A	B	439	0
sthx	31	S	A	B	407	0
stmw[3]	47	S	A	d		
stswi[3]	31	S	A	NB	725	0
stswx[3]	31	S	A	B	661	0
stw	36	S	A	d		
stwbrx	31	S	A	B	662	0
stwcx.	31	S	A	B	150	1

Name	0	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
stwu	37	S	A	d			
stwux	31	S	A	B	183		0
stwx	31	S	A	B	151		0
subf*x*	31	D	A	B	OE	40	Rc
subfc*x*	31	D	A	B	OE	8	Rc
subfe*x*	31	D	A	B	OE	136	Rc
subfic	08	D	A	SIMM			
subfme*x*	31	D	A	0 0 0 0 0	OE	232	Rc
subfze*x*	31	D	A	0 0 0 0 0	OE	200	Rc
sync	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	598		0
td[5]	31	TO	A	B	68		0
tdi[5]	02	TO	A	SIMM			
tlbia[1,4]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	370		0
tlbie[1,4]	31	0 0 0 0 0	0 0 0 0 0	B	306		0
tlbsync[1,4]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	566		0
tw	31	TO	A	B	4		0
twi	03	TO	A	SIMM			
xor*x*	31	S	A	B	316		Rc
xori	26	S	A	UIMM			
xoris	27	S	A	UIMM			

[1] Supervisor-level instruction
[2] Supervisor- and user-level instruction
[3] Load and store string or multiple instruction
[4] Optional instruction
[5] 64-bit instruction

A.2 Instructions Sorted by Opcode

Table A-2 lists the 620 instruction set sorted in numeric order by opcode, including those PowerPC instructions not implemented by the 620.

Key:

□	Reserved bits
▨	Instruction not implemented in the 620

Table A-2. Complete Instruction List Sorted by Opcode

Name	0 … 5	6 7 8	9	10	11 12 13	14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29	30	31
tdi [5]	000010	TO			A		SIMM			
twi	000011	TO			A		SIMM			
mulli	000111	D			A		SIMM			
subfic	001000	D			A		SIMM			
cmpli	001010	crfD	0	L	A		UIMM			
cmpi	001011	crfD	0	L	A		SIMM			
addic	001100	D			A		SIMM			
addic.	001101	D			A		SIMM			
addi	001110	D			A		SIMM			
addis	001111	D			A		SIMM			
bc*x*	010000	BO			BI		BD		AA	LK
sc	010001	00000			00000		000000000000000		1	0
b*x*	010010	LI							AA	LK
mcrf	010011	crfD	00		crfS	00	00000	0000000000		0
bclr*x*	010011	BO			BI		00000	0000010000		LK
rfid [1, 5]	010011	00000			00000		00000	0000010010		0
crnor	010011	crbD			crbA		crbB	0000100001		0
rfi [1, 6]	010011	00000			00000		00000	0000110010		0
crandc	010011	crbD			crbA		crbB	0010000001		0
isync	010011	00000			00000		00000	0010010110		0
crxor	010011	crbD			crbA		crbB	0011000001		0
crnand	010011	crbD			crbA		crbB	0011100001		0
crand	010011	crbD			crbA		crbB	0100000001		0
creqv	010011	crbD			crbA		crbB	0100100001		0
crorc	010011	crbD			crbA		crbB	0110100001		0
cror	010011	crbD			crbA		crbB	0111000001		0

Name	0 … 5	6 7 8	9	10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25	26	27 28 29	30	31
bcctr*x*	010011	BO			BI	00000	1000010000					LK
rlwimi*x*	010100	S			A	SH	MB		ME			Rc
rlwinm*x*	010101	S			A	SH	MB		ME			Rc
rlwnm*x*	010111	S			A	B	MB		ME			Rc
ori	011000	S			A	UIMM						
oris	011001	S			A	UIMM						
xori	011010	S			A	UIMM						
xoris	011011	S			A	UIMM						
andi.	011100	S			A	UIMM						
andis.	011101	S			A	UIMM						
rldicl*x*[5]	011110	S			A	sh	mb			000	sh	Rc
rldicr*x*[5]	011110	S			A	sh	me			001	sh	Rc
rldic*x*[5]	011110	S			A	sh	mb			010	sh	Rc
rldimi*x*[5]	011110	S			A	sh	mb			011	sh	Rc
rldc!*x*[5]	011110	S			A	B	mb		01000			Rc
rldcr*x*[5]	011110	S			A	B	me			01001		Rc
cmp	011111	crfD	0	L	A	B	0000000000					0
tw	011111	TO			A	B	0000000100					0
subfc*x*	011111	D			A	B	OE	0000001000				Rc
mulhdu*x*[5]	011111	D			A	B	0	0000001001				Rc
addc*x*	011111	D			A	B	OE	0000001010				Rc
mulhwu*x*	011111	D			A	B	0	0000001011				Rc
mfcr	011111	D			00000	00000	0000010011					0
lwarx	011111	D			A	B	0000010100					0
ldx[5]	011111	D			A	B	0000010101					0
lwzx	011111	D			A	B	0000010111					0
slw*x*	011111	S			A	B	0000011000					Rc
cntlzw*x*	011111	S			A	00000	0000011010					Rc
sld*x*[5]	011111	S			A	B	0000011011					Rc
and*x*	011111	S			A	B	0000011100					Rc
cmpl	011111	crfD	0	L	A	B	0000100000					0
subf*x*	011111	D			A	B	OE	0000101000				Rc
ldux[5]	011111	D			A	B	0000110101					0

Name	0 5	6 7 8 9 10	11	12 13 14 15	16 17 18 19	20	21	22 23 24 25 26 27 28 29 30	31
dcbst	0 1 1 1 1 1	0 0 0 0 0	A		B		0 0 0 0 1 1 0 1 1 0		0
lwzux	0 1 1 1 1 1	D	A		B		0 0 0 0 1 1 0 1 1 1		0
cntlzd*x*[5]	0 1 1 1 1 1	S	A		0 0 0 0 0		0 0 0 0 1 1 1 0 1 0		Rc
andc*x*	0 1 1 1 1 1	S	A		B		0 0 0 0 1 1 1 1 0 0		Rc
td[5]	0 1 1 1 1 1	TO	A		B		0 0 0 1 0 0 0 1 0 0		0
mulhd*x*[5]	0 1 1 1 1 1	D	A		B		0	0 0 0 1 0 0 1 0 0 1	Rc
mulhw*x*	0 1 1 1 1 1	D	A		B		0	0 0 0 1 0 0 1 0 1 1	Rc
mtsrd[1, 6]	0 1 1 1 1 1	S	0	SR	0 0 0 0 0		0 0 0 1 0 1 0 0 1 0		0
mfmsr[1]	0 1 1 1 1 1	D	0 0 0 0 0		0 0 0 0 0		0 0 0 1 0 1 0 0 1 1		0
ldarx[5]	0 1 1 1 1 1	D	A		B		0 0 0 1 0 1 0 1 0 0		0
dcbf	0 1 1 1 1 1	0 0 0 0 0	A		B		0 0 0 1 0 1 0 1 1 0		0
lbzx	0 1 1 1 1 1	D	A		B		0 0 0 1 0 1 0 1 1 1		0
neg*x*	0 1 1 1 1 1	D	A		0 0 0 0 0		OE	0 0 0 1 1 0 1 0 0 0	Rc
mtsrdin[1, 6]	0 1 1 1 1 1	S	0 0 0 0 0		B		0 0 0 1 1 1 0 0 1 0		0
lbzux	0 1 1 1 1 1	D	A		B		0 0 0 1 1 1 0 1 1 1		0
nor*x*	0 1 1 1 1 1	S	A		B		0 0 0 1 1 1 1 1 0 0		Rc
subfe*x*	0 1 1 1 1 1	D	A		B		OE	0 0 1 0 0 0 1 0 0 0	Rc
adde*x*	0 1 1 1 1 1	D	A		B		OE	0 0 1 0 0 0 1 0 1 0	Rc
mtcrf	0 1 1 1 1 1	S	0	CRM		0	0 0 1 0 0 1 0 0 0 0		0
mtmsr[1, 6]	0 1 1 1 1 1	S	0 0 0 0 0		0 0 0 0 0		0 0 1 0 0 1 0 0 1 0		0
stdx[5]	0 1 1 1 1 1	S	A		B		0 0 1 0 0 1 0 1 0 1		0
stwcx.	0 1 1 1 1 1	S	A		B		0 0 1 0 0 1 0 1 1 0		1
stwx	0 1 1 1 1 1	S	A		B		0 0 1 0 0 1 0 1 1 1		0
mtmsrd[1, 5]	0 1 1 1 1 1	S	0 0 0 0 0		0 0 0 0 0		0 0 1 0 1 1 0 0 1 0		0
stdux[5]	0 1 1 1 1 1	S	A		B		0 0 1 0 1 1 0 1 0 1		0
stwux	0 1 1 1 1 1	S	A		B		0 0 1 0 1 1 0 1 1 1		0
subfze*x*	0 1 1 1 1 1	D	A		0 0 0 0 0		OE	0 0 1 1 0 0 1 0 0 0	Rc
addze*x*	0 1 1 1 1 1	D	A		0 0 0 0 0		OE	0 0 1 1 0 0 1 0 1 0	Rc
mtsr[1, 6]	0 1 1 1 1 1	S	0	SR	0 0 0 0 0		0 0 1 1 0 1 0 0 1 0		0
stdcx.[5]	0 1 1 1 1 1	S	A		B		0 0 1 1 0 1 0 1 1 0		1
stbx	0 1 1 1 1 1	S	A		B		0 0 1 1 0 1 0 1 1 1		0
subfme*x*	0 1 1 1 1 1	D	A		0 0 0 0 0		OE	0 0 1 1 1 0 1 0 0 0	Rc
mulld[5]	0 1 1 1 1 1	D	A		B		OE	0 0 1 1 1 0 1 0 0 1	Rc

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
addme*x*	0 1 1 1 1 1	D	A	0 0 0 0 0	OE	0 0 1 1 1 0 1 0 1 0	Rc
mullw*x*	0 1 1 1 1 1	D	A	B	OE	0 0 1 1 1 0 1 0 1 1	Rc
mtsrin[1, 6]	0 1 1 1 1 1	S	0 0 0 0 0	B	0 0 1 1 1 1 0 0 1 0		0
dcbtst	0 1 1 1 1 1	0 0 0 0 0	A	B	0 0 1 1 1 1 0 1 1 0		0
stbux	0 1 1 1 1 1	S	A	B	0 0 1 1 1 1 0 1 1 1		0
add*x*	0 1 1 1 1 1	D	A	B	OE	0 1 0 0 0 0 1 0 1 0	Rc
dcbt	0 1 1 1 1 1	0 0 0 0 0	A	B	0 1 0 0 0 1 0 1 1 0		0
lhzx	0 1 1 1 1 1	D	A	B	0 1 0 0 0 1 0 1 1 1		0
eqv*x*	0 1 1 1 1 1	S	A	B	0 1 0 0 0 1 1 1 0 0		Rc
tlbie[1,4]	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	B	0 1 0 0 1 1 0 0 1 0		0
eciwx	0 1 1 1 1 1	D	A	B	0 1 0 0 1 1 0 1 1 0		0
lhzux	0 1 1 1 1 1	D	A	B	0 1 0 0 1 1 0 1 1 1		0
xor*x*	0 1 1 1 1 1	S	A	B	0 1 0 0 1 1 1 1 0 0		Rc
mfspr[2]	0 1 1 1 1 1	D	spr		0 1 0 1 0 1 0 0 1 1		0
lwax[5]	0 1 1 1 1 1	D	A	B	0 1 0 1 0 1 0 1 0 1		0
lhax	0 1 1 1 1 1	D	A	B	0 1 0 1 0 1 0 1 1 1		0
tlbia[1,4]	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 1 0 1 1 1 0 0 1 0		0
mftb	0 1 1 1 1 1	D	tbr		0 1 0 1 1 1 0 0 1 1		0
lwaux[5]	0 1 1 1 1 1	D	A	B	0 1 0 1 1 1 0 1 0 1		0
lhaux	0 1 1 1 1 1	D	A	B	0 1 0 1 1 1 0 1 1 1		0
sthx	0 1 1 1 1 1	S	A	B	0 1 1 0 0 1 0 1 1 1		0
orc*x*	0 1 1 1 1 1	S	A	B	0 1 1 0 0 1 1 1 0 0		Rc
sradi*x*[5]	0 1 1 1 1 1	S	A	sh	1 1 0 0 1 1 1 0 1 1	sh	Rc
slbie[1,4,5]	0 1 1 1 1 1	0 0 0 0 0	0 0 0 0 0	B	0 1 1 0 1 1 0 0 1 0		0
ecowx	0 1 1 1 1 1	S	A	B	0 1 1 0 1 1 0 1 1 0		0
sthux	0 1 1 1 1 1	S	A	B	0 1 1 0 1 1 0 1 1 1		0
or*x*	0 1 1 1 1 1	S	A	B	0 1 1 0 1 1 1 1 0 0		Rc
divdu*x*[5]	0 1 1 1 1 1	D	A	B	OE	0 1 1 1 0 0 1 0 0 1	Rc
divwu*x*	0 1 1 1 1 1	D	A	B	OE	0 1 1 1 0 0 1 0 1 1	Rc
mtspr[2]	0 1 1 1 1 1	S	spr		0 1 1 1 0 1 0 0 1 1		0
dcbi[1]	0 1 1 1 1 1	0 0 0 0 0	A	B	0 1 1 1 0 1 0 1 1 0		0
nand*x*	0 1 1 1 1 1	S	A	B	0 1 1 1 0 1 1 1 0 0		Rc
divd*x*[5]	0 1 1 1 1 1	D	A	B	OE	0 1 1 1 1 0 1 0 0 1	Rc

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
divw*x*	011111	D	A	B	OE 0111101011	Rc
slbia [1,4,5]	011111	00000	00000	00000	0111110010	0
mcrxr	011111	crfD 00	00000	00000	1000000000	0
lswx [3]	011111	D	A	B	1000010101	0
lwbrx	011111	D	A	B	1000010110	0
lfsx	011111	D	A	B	1000010111	0
srw*x*	011111	S	A	B	1000011000	Rc
srd*x* [4]	011111	S	A	B	1000011011	Rc
tlbsync [1,4]	011111	00000	00000	00000	1000110110	0
lfsux	011111	D	A	B	1000110111	0
mfsr [1, 6]	011111	D	0 SR	00000	1001010011	0
lswi [3]	011111	D	A	NB	1001010101	0
sync	011111	00000	00000	00000	1001010110	0
lfdx	011111	D	A	B	1001010111	0
lfdux	011111	D	A	B	1001110111	0
mfsrin [1, 6]	011111	D	00000	B	1010010011	0
stswx [3]	011111	S	A	B	1010010101	0
stwbrx	011111	S	A	B	1010010110	0
stfsx	011111	S	A	B	1010010111	0
stfsux	011111	S	A	B	1010110111	0
stswi [3]	011111	S	A	NB	1011010101	0
stfdx	011111	S	A	B	1011010111	0
dcba [4]	31	00000	A	B	1011110110	0
stfdux	011111	S	A	B	1011110111	0
lhbrx	011111	D	A	B	1100010110	0
sraw*x*	011111	S	A	B	1100011000	Rc
srad*x* [5]	011111	S	A	B	1100011010	Rc
srawi*x*	011111	S	A	SH	1100111000	Rc
eieio	011111	00000	00000	00000	1101010110	0
sthbrx	011111	S	A	B	1110010110	0
extsh*x*	011111	S	A	00000	1110011010	Rc
extsb*x*	011111	S	A	00000	1110111010	Rc
icbi	011111	00000	A	B	1111010110	0

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29	30	31
stfiwx	0 1 1 1 1 1	S	A	B	1 1 1 1 0 1 0 1 1 1			0
extsw[5]	0 1 1 1 1 1	S	A	0 0 0 0 0	1 1 1 1 0 1 1 0 1 0			Rc
dcbz	0 1 1 1 1 1	0 0 0 0 0	A	B	1 1 1 1 1 1 0 1 1 0			0
lwz	1 0 0 0 0 0	D	A	d				
lwzu	1 0 0 0 0 1	D	A	d				
lbz	1 0 0 0 1 0	D	A	d				
lbzu	1 0 0 0 1 1	D	A	d				
stw	1 0 0 1 0 0	S	A	d				
stwu	1 0 0 1 0 1	S	A	d				
stb	1 0 0 1 1 0	S	A	d				
stbu	1 0 0 1 1 1	S	A	d				
lhz	1 0 1 0 0 0	D	A	d				
lhzu	1 0 1 0 0 1	D	A	d				
lha	1 0 1 0 1 0	D	A	d				
lhau	1 0 1 0 1 1	D	A	d				
sth	1 0 1 1 0 0	S	A	d				
sthu	1 0 1 1 0 1	S	A	d				
lmw[3]	1 0 1 1 1 0	D	A	d				
stmw[3]	1 0 1 1 1 1	S	A	d				
lfs	1 1 0 0 0 0	D	A	d				
lfsu	1 1 0 0 0 1	D	A	d				
lfd	1 1 0 0 1 0	D	A	d				
lfdu	1 1 0 0 1 1	D	A	d				
stfs	1 1 0 1 0 0	S	A	d				
stfsu	1 1 0 1 0 1	S	A	d				
stfd	1 1 0 1 1 0	S	A	d				
stfdu	1 1 0 1 1 1	S	A	d				
ld[5]	1 1 1 0 1 0	D	A	ds			0 0	
ldu[5]	1 1 1 0 1 0	D	A	ds			0 1	
lwa[5]	1 1 1 0 1 0	D	A	ds			1 0	
fdivs*x*	1 1 1 0 1 1	D	A	B	0 0 0 0 0	1 0 0 1 0		Rc
fsubs*x*	1 1 1 0 1 1	D	A	B	0 0 0 0 0	1 0 1 0 0		Rc
fadds*x*	1 1 1 0 1 1	D	A	B	0 0 0 0 0	1 0 1 0 1		Rc

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fsqrts*x*[4]	1 1 1 0 1 1	D	0 0 0 0 0	B	0 0 0 0 0	1 0 1 1 0	Rc
fres*x*[4]	1 1 1 0 1 1	D	0 0 0 0 0	B	0 0 0 0 0	1 1 0 0 0	Rc
fmuls*x*	1 1 1 0 1 1	D	A	0 0 0 0 0	C	1 1 0 0 1	Rc
fmsubs*x*	1 1 1 0 1 1	D	A	B	C	1 1 1 0 0	Rc
fmadds*x*	1 1 1 0 1 1	D	A	B	C	1 1 1 0 1	Rc
fnmsubs*x*	1 1 1 0 1 1	D	A	B	C	1 1 1 1 0	Rc
fnmadds*x*	1 1 1 0 1 1	D	A	B	C	1 1 1 1 1	Rc
std[5]	1 1 1 1 1 0	S	A	ds			0 0
stdu[5]	1 1 1 1 1 0	S	A	ds			0 1
fcmpu	1 1 1 1 1 1	crfD 0 0	A	B	0 0 0 0 0 0 0 0 0 0		0
frsp*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 0 0 1 1 0 0		Rc
fctiw*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 0 0 1 1 1 0		
fctiwz*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 0 0 1 1 1 1		Rc
fdiv*x*	1 1 1 1 1 1	D	A	B	0 0 0 0 0	1 0 0 1 0	Rc
fsub*x*	1 1 1 1 1 1	D	A	B	0 0 0 0 0	1 0 1 0 0	Rc
fadd*x*	1 1 1 1 1 1	D	A	B	0 0 0 0 0	1 0 1 0 1	Rc
fsqrt*x*[4]	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 0	1 0 1 1 0	Rc
fsel*x*[4]	1 1 1 1 1 1	D	A	B	C	1 0 1 1 1	Rc
fmul*x*	1 1 1 1 1 1	D	A	0 0 0 0 0	C	1 1 0 0 1	Rc
frsqrte*x*[5]	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 0	1 1 0 1 0	Rc
fmsub*x*	1 1 1 1 1 1	D	A	B	C	1 1 1 0 0	Rc
fmadd*x*	1 1 1 1 1 1	D	A	B	C	1 1 1 0 1	Rc
fnmsub*x*	1 1 1 1 1 1	D	A	B	C	1 1 1 1 0	Rc
fnmadd*x*	1 1 1 1 1 1	D	A	B	C	1 1 1 1 1	Rc
fcmpo	1 1 1 1 1 1	crfD 0 0	A	B	0 0 0 0 1 0 0 0 0 0		0
mtfsb1*x*	1 1 1 1 1 1	crbD	0 0 0 0 0	0 0 0 0 0	0 0 0 0 1 0 0 1 1 0		Rc
fneg*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 0 1 0 1 0 0 0		Rc
mcrfs	1 1 1 1 1 1	crfD 0 0	crfS 0 0	0 0 0 0 0	0 0 0 1 0 0 0 0 0 0		0
mtfsb0*x*	1 1 1 1 1 1	crbD	0 0 0 0 0	0 0 0 0 0	0 0 0 1 0 0 0 1 1 0		Rc
fmr*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 0 1 0 0 1 0 0 0		Rc
mtfsfi*x*	1 1 1 1 1 1	crfD 0 0	0 0 0 0 0	IMM 0	0 0 1 0 0 0 0 1 1 0		Rc
fnabs*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 0 1 0 0 0 1 0 0 0		Rc
fabs*x*	1 1 1 1 1 1	D	0 0 0 0 0	B	0 1 0 0 0 0 1 0 0 0		Rc

Name	0 5	6	7 8 9 10	11 12 13 14	15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mffs*x*	1 1 1 1 1 1	D		0 0 0 0 0		0 0 0 0 0	1 0 0 1 0 0 0 1 1 1	Rc
mtfsf*x*	1 1 1 1 1 1	0	FM		0	B	1 0 1 1 0 0 0 1 1 1	Rc
fctid*x* [5]	1 1 1 1 1 1	D		0 0 0 0 0		B	1 1 0 0 1 0 1 1 1 0	Rc
fctidz*x* [5]	1 1 1 1 1 1	D		0 0 0 0 0		B	1 1 0 0 1 0 1 1 1 1	Rc
fcfid*x* [5]	1 1 1 1 1 1	D		0 0 0 0 0		B	1 1 0 1 0 0 1 1 1 0	Rc

[1] Supervisor-level instruction
[2] Supervisor- and user-level instruction
[3] Load and store string or multiple instruction
[4] Optional instruction
[5] 64-bit instruction
[6] Optional 64-bit bridge instruction

A.3 Instructions Grouped by Functional Categories

Table A-3 through Table A-30 list the 620 instructions grouped by function, as well as the PowerPC instructions not implemented in the 620.

Key:

□ Reserved bits

▨ Instruction not implemented in the 620

Table A-3. Integer Arithmetic Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
add*x*	31	D	A	B	OE	266	Rc
addc*x*	31	D	A	B	OE	10	Rc
adde*x*	31	D	A	B	OE	138	Rc
addi	14	D	A	SIMM			
addic	12	D	A	SIMM			
addic.	13	D	A	SIMM			
addis	15	D	A	SIMM			
addme*x*	31	D	A	0 0 0 0 0	OE	234	Rc
addze*x*	31	D	A	0 0 0 0 0	OE	202	Rc
divd*x*[5]	31	D	A	B	OE	489	Rc
divdu*x*[5]	31	D	A	B	OE	457	Rc
divw*x*	31	D	A	B	OE	491	Rc
divwu*x*	31	D	A	B	OE	459	Rc
mulhd*x*[5]	31	D	A	B	0	73	Rc
mulhdu*x*[5]	31	D	A	B	0	9	Rc
mulhw*x*	31	D	A	B	0	75	Rc
mulhwu*x*	31	D	A	B	0	11	Rc
mulld[5]	31	D	A	B	OE	233	Rc
mulli	07	D	A	SIMM			
mullw*x*	31	D	A	B	OE	235	Rc
neg*x*	31	D	A	0 0 0 0 0	OE	104	Rc
subf*x*	31	D	A	B	OE	40	Rc
subfc*x*	31	D	A	B	OE	8	Rc

subfic*x*	08	D	A	SIMM			
subfe*x*	31	D	A	B	OE	136	Rc
subfme*x*	31	D	A	0 0 0 0 0	OE	232	Rc
subfze*x*	31	D	A	0 0 0 0 0	OE	200	Rc

Table A-4. Integer Compare Instructions

Name	0 5	6 7 8	9	10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
cmp	31	crfD	0	L	A	B	0 0 0 0 0 0 0 0 0 0	0
cmpi	11	crfD	0	L	A	SIMM		
cmpl	31	crfD	0	L	A	B	32	0
cmpli	10	crfD	0	L	A	UIMM		

Table A-5. Integer Logical Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
and*x*	31	S	A	B	28	Rc
andc*x*	31	S	A	B	60	Rc
andi.	28	S	A	UIMM		
andis.	29	S	A	UIMM		
cntlzd*x*[5]	31	S	A	0 0 0 0 0	58	Rc
cntlzw*x*	31	S	A	0 0 0 0 0	26	Rc
eqv*x*	31	S	A	B	284	Rc
extsb*x*	31	S	A	0 0 0 0 0	954	Rc
extsh*x*	31	S	A	0 0 0 0 0	922	Rc
extsw*x*[5]	31	S	A	0 0 0 0 0	986	Rc
nand*x*	31	S	A	B	476	Rc
nor*x*	31	S	A	B	124	Rc
or*x*	31	S	A	B	444	Rc
orc*x*	31	S	A	B	412	Rc
ori	24	S	A	UIMM		
oris	25	S	A	UIMM		
xor*x*	31	S	A	B	316	Rc
xori	26	S	A	UIMM		
xoris	27	S	A	UIMM		

Table A-6. Integer Rotate Instructions

Name	0 … 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 …	… 30	30	31
rldcl*x*[5]	30	S	A	B	mb	8		Rc
rldcr*x*[5]	30	S	A	B	me	9		Rc
rldic*x*[5]	30	S	A	sh	mb	2	sh	Rc
rldicl*x*[5]	30	S	A	sh	mb	0	sh	Rc
rldicr*x*[5]	30	S	A	sh	me	1	sh	Rc
rldimi*x*[5]	30	S	A	sh	mb	3	sh	Rc
rlwimi*x*	22	S	A	SH	MB	ME		Rc
rlwinm*x*	20	S	A	SH	MB	ME		Rc
rlwnm*x*	21	S	A	SH	MB	ME		Rc

Table A-7. Integer Shift Instructions

Name	0 … 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 … 30	30	31
sld*x*[5]	31	S	A	B	27		Rc
slw*x*	31	S	A	B	24		Rc
srad*x*[5]	31	S	A	B	794		Rc
sradi*x*[5]	31	S	A	sh	413	sh	Rc
sraw*x*	31	S	A	B	792		Rc
srawi*x*	31	S	A	SH	824		Rc
srd*x*[5]	31	S	A	B	539		Rc
srw*x*	31	S	A	B	536		Rc

Table A-8. Floating-Point Arithmetic Instructions

Name	0 … 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fadd*x*	63	D	A	B	00000	21	Rc
fadds*x*	59	D	A	B	00000	21	Rc
fdiv*x*	63	D	A	B	00000	18	Rc
fdivs*x*	59	D	A	B	00000	18	Rc
fmul*x*	63	D	A	00000	C	25	Rc
fmuls*x*	59	D	A	00000	C	25	Rc
fres*x*[4]	59	D	00000	B	00000	24	Rc
frsqrte*x*[4]	63	D	00000	B	00000	26	Rc
fsub*x*	63	D	A	B	00000	20	Rc

fsubs*x*	59	D	A	B	0 0 0 0 0	20	Rc
fsel*x*[4]	63	D	A	B	C	23	Rc
fsqrt*x*[4]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrts*x*[4]	59	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc

Table A-9. Floating-Point Multiply-Add Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fmadd*x*	63	D	A	B	C	29	Rc
fmadds*x*	59	D	A	B	C	29	Rc
fmsub*x*	63	D	A	B	C	28	Rc
fmsubs*x*	59	D	A	B	C	28	Rc
fnmadd*x*	63	D	A	B	C	31	Rc
fnmadds*x*	59	D	A	B	C	31	Rc
fnmsub*x*	63	D	A	B	C	30	Rc
fnmsubs*x*	59	D	A	B	C	30	Rc

Table A-10. Floating-Point Rounding and Conversion Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
fcfid*x*[5]	63	D	0 0 0 0 0	B	846	Rc
fctid*x*[5]	63	D	0 0 0 0 0	B	814	Rc
fctidz*x*[5]	63	D	0 0 0 0 0	B	815	Rc
fctiw*x*	63	D	0 0 0 0 0	B	14	Rc
fctiwz*x*	63	D	0 0 0 0 0	B	15	Rc
frsp*x*	63	D	0 0 0 0 0	B	12	Rc

Table A-11. Floating-Point Compare Instructions

Name	0 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
fcmpo	63	crfD	0 0	A	B	32	0
fcmpu	63	crfD	0 0	A	B	0	0

Table A-12. Floating-Point Status and Control Register Instructions

Name	0 5	6	7 8	9 10	11 12 13	14	15	16 17 18 19	20	21 22 23 24 25 26 27 28 29 30	31
mcrfs	63	crfD		0 0	crfS	0 0		0 0 0 0 0		64	0
mffs*x*	63	D			0 0 0 0 0			0 0 0 0 0		583	Rc
mtfsb0*x*	63	crbD			0 0 0 0 0			0 0 0 0 0		70	Rc
mtfsb1*x*	63	crbD			0 0 0 0 0			0 0 0 0 0		38	Rc
mtfsf*x*	31	0	FM				0	B		711	Rc
mtfsfi*x*	63	crfD		0 0	0 0 0 0 0			IMM	0	134	Rc

Table A-13. Integer Load Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29	30	31
lbz	34	D	A	d			
lbzu	35	D	A	d			
lbzux	31	D	A	B	119		0
lbzx	31	D	A	B	87		0
ld [5]	58	D	A	ds			0
ldu [5]	58	D	A	ds			1
ldux [5]	31	D	A	B	53		0
ldx [5]	31	D	A	B	21		0
lha	42	D	A	d			
lhau	43	D	A	d			
lhaux	31	D	A	B	375		0
lhax	31	D	A	B	343		0
lhz	40	D	A	d			
lhzu	41	D	A	d			
lhzux	31	D	A	B	311		0
lhzx	31	D	A	B	279		0
lwa [5]	58	D	A	ds			2
lwaux [5]	31	D	A	B	373		0
lwax [5]	31	D	A	B	341		0
lwz	32	D	A	d			
lwzu	33	D	A	d			
lwzux	31	D	A	B	55		0
lwzx	31	D	A	B	23		0

Table A-14. Integer Store Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
stb	38	S	A	d		
stbu	39	S	A	d		
stbux	31	S	A	B	247	0
stbx	31	S	A	B	215	0
std[5]	62	S	A	ds		0
stdu[5]	62	S	A	ds		1
stdux[5]	31	S	A	B	181	0
stdx[5]	31	S	A	B	149	0
sth	44	S	A	d		
sthu	45	S	A	d		
sthux	31	S	A	B	439	0
sthx	31	S	A	B	407	0
stw	36	S	A	d		
stwu	37	S	A	d		
stwux	31	S	A	B	183	0
stwx	31	S	A	B	151	0

Table A-15. Integer Load and Store with Byte Reverse Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lhbrx	31	D	A	B	790	0
lwbrx	31	D	A	B	534	0
sthbrx	31	S	A	B	918	0
stwbrx	31	S	A	B	662	0

Table A-16. Integer Load and Store Multiple Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lmw[3]	46	D	A	d		
stmw[3]	47	S	A	d		

Table A-17. Integer Load and Store String Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lswi[3]	31	D	A	NB	597	0
lswx[3]	31	D	A	B	533	0
stswi[3]	31	S	A	NB	725	0
stswx[3]	31	S	A	B	661	0

Table A-18. Memory Synchronization Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
eieio	31	00000	00000	00000	854	0
isync	19	00000	00000	00000	150	0
ldarx[5]	31	D	A	B	84	0
lwarx	31	D	A	B	20	0
stdcx.[5]	31	S	A	B	214	1
stwcx.	31	S	A	B	150	1
sync	31	00000	00000	00000	598	0

Table A-19. Floating-Point Load Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
lfd	50	D	A	d		
lfdu	51	D	A	d		
lfdux	31	D	A	B	631	0
lfdx	31	D	A	B	599	0
lfs	48	D	A	d		
lfsu	49	D	A	d		
lfsux	31	D	A	B	567	0
lfsx	31	D	A	B	535	0

Table A-20. Floating-Point Store Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
stfd	54	S	A	d		
stfdu	55	S	A	d		
stfdux	31	S	A	B	759	0
stfdx	31	S	A	B	727	0

stfiwx	31	S	A	B	983	0
stfs	52	S	A	d		
stfsu	53	S	A	d		
stfsux	31	S	A	B	695	0
stfsx	31	S	A	B	663	0

Table A-21. Floating-Point Move Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
fabs*x*	63	D	0 0 0 0 0	B	264	Rc
fmr*x*	63	D	0 0 0 0 0	B	72	Rc
fnabs*x*	63	D	0 0 0 0 0	B	136	Rc
fneg*x*	63	D	0 0 0 0 0	B	40	Rc

Table A-22. Branch Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29	30	31
b*x*	18	LI				AA	LK
bc*x*	16	BO	BI	BD		AA	LK
bcctr*x*	19	BO	BI	0 0 0 0 0	528		LK
bclr*x*	19	BO	BI	0 0 0 0 0	16		LK

Table A-23. Condition Register Logical Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
crand	19	crbD	crbA	crbB	257	0
crandc	19	crbD	crbA	crbB	129	0
creqv	19	crbD	crbA	crbB	289	0
crnand	19	crbD	crbA	crbB	225	0
crnor	19	crbD	crbA	crbB	33	0
cror	19	crbD	crbA	crbB	449	0
crorc	19	crbD	crbA	crbB	417	0
crxor	19	crbD	crbA	crbB	193	0
mcrf	19	crfD 0 0	crfS 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0	0

Table A-24. System Linkage Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29	30	31
rfi[1, 6]	19	00000	00000	00000	50		0
rfid[1, 5]	19	00000	00000	00000	18		0
sc	17	00000	00000	000000000000000		1	0

Table A-25. Trap Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
td[5]	31	TO	A	B	68	0
tdi[5]	03	TO	A	SIMM		
tw	31	TO	A	B	4	0
twi	03	TO	A	SIMM		

Table A-26. Processor Control Instructions

Name	0 5	6 7 8	9 10	11	12 13 14 15	16 17 18 19	20	21 22 23 24 25 26 27 28 29 30	31
mcrxr	31	crfS	00	00000		00000		512	0
mfcr	31	D		00000		00000		19	0
mfmsr[1]	31	D		00000		00000		83	0
mfspr[2]	31	D		spr				339	0
mftb	31	D		tpr				371	0
mtcrf	31	S		0	CRM		0	144	0
mtmsr[1, 6]	31	S		00000		00000		146	0
mtmsrd[1, 5]	31	S		00000		00000		178	0
mtspr[2]	31	D		spr				467	0

Table A-27. Cache Management Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
dcba[4]	31	00000	A	B	758	0
dcbf	31	00000	A	B	86	0
dcbi[1]	31	00000	A	B	470	0
dcbst	31	00000	A	B	54	0
dcbt	31	00000	A	B	278	0

dcbtst	31	00000	A	B	246	0
dcbz	31	00000	A	B	1014	0
icbi	31	00000	A	B	982	0

Table A-28. Segment Register Manipulation Instructions

Name	0 5	6 7 8 9 10	11	12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mfsr[1, 6]	31	D	0	SR	00000	595	0
mfsrin[1, 6]	31	D	00000		B	659	0
mtsr[1, 6]	31	S	0	SR	00000	210	0
mtsrd[1, 6]	31	S	0	SR	00000	82	0
mtsrdin[1, 6]	31	S	00000		B	114	0
mtsrin[1, 6]	31	S	00000		B	242	0

Table A-29. Lookaside Buffer Management Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
slbia[1,4,5]	31	00000	00000	00000	498	0
slbie[1,4,5]	31	00000	00000	B	434	0
tlbia[1,4]	31	00000	00000	00000	370	0
tlbie[1,4]	31	00000	00000	B	306	0
tlbsync[1,4]	31	00000	00000	00000	566	0

Table A-30. External Control Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
eciwx	31	D	A	B	310	0
ecowx	31	S	A	B	438	0

[1] Supervisor-level instruction
[2] Supervisor- and user-level instruction
[3] Load and store string or multiple instruction
[4] Optional instruction
[5] 64-bit instruction
[6] Optional 64-bit bridge instruction

A.4 Instructions Sorted by Form

Table A-31 through Table A-45 list the 620 instructions grouped by form, including those PowerPC instructions not implemented in the 620.

Key:

□ Reserved bits

▨ Instruction not implemented in the 620

Table A-31. I-Form

OPCD	LI	AA	LK

Specific Instruction

Name	0 … 5	6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	30	31
b*x*	18	LI	AA	LK

Table A-32. B-Form

OPCD	BO	BI	BD	AA	LK

Specific Instruction

Name	0 … 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29	30	31
bc*x*	16	BO	BI	BD	AA	LK

Table A-33. SC-Form

OPCD	00000	00000	000000000000000	1	0

Specific Instruction

Name	0 … 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29	30	31
sc	17	00000	00000	000000000000000	1	0

Table A-34. D-Form

OPCD	D			A	d
OPCD	D			A	SIMM
OPCD	S			A	d
OPCD	S			A	UIMM
OPCD	crfD	0	L	A	SIMM
OPCD	crfD	0	L	A	UIMM
OPCD	TO			A	SIMM

Specific Instructions

Name	0 5	6 7 8	9	10	11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
addi	14	D			A	SIMM
addic	12	D			A	SIMM
addic.	13	D			A	SIMM
addis	15	D			A	SIMM
andi.	28	S			A	UIMM
andis.	29	S			A	UIMM
cmpi	11	crfD	0	L	A	SIMM
cmpli	10	crfD	0	L	A	UIMM
lbz	34	D			A	d
lbzu	35	D			A	d
lfd	50	D			A	d
lfdu	51	D			A	d
lfs	48	D			A	d
lfsu	49	D			A	d
lha	42	D			A	d
lhau	43	D			A	d
lhz	40	D			A	d
lhzu	41	D			A	d
lmw[3]	46	D			A	d
lwz	32	D			A	d
lwzu	33	D			A	d
mulli	7	D			A	SIMM
ori	24	S			A	UIMM
oris	25	S			A	UIMM
stb	38	S			A	d
stbu	39	S			A	d
stfd	54	S			A	d
stfdu	55	S			A	d
stfs	52	S			A	d
stfsu	53	S			A	d
sth	44	S			A	d
sthu	45	S			A	d
stmw[3]	47	S			A	d

stw	36	S	A	d
stwu	37	S	A	d
subfic	08	D	A	SIMM
tdi [5]	02	TO	A	SIMM
twi	03	TO	A	SIMM
xori	26	S	A	UIMM
xoris	27	S	A	UIMM

Table A-35. DS-Form

OPCD	D	A	ds	XO
OPCD	S	A	ds	XO

Specific Instructions

Name 0 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

ld [5]	58	D	A	ds	0
ldu [5]	58	D	A	ds	1
lwa [5]	58	D	A	ds	2
std [5]	62	S	A	ds	0
stdu [5]	62	S	A	ds	1

Table A-36. X-Form

OPCD	D	A	B	XO	0
OPCD	D	A	NB	XO	0
OPCD	D	0 0 0 0 0	B	XO	0
OPCD	D	0 0 0 0 0	0 0 0 0 0	XO	0
OPCD	D	0 SR	0 0 0 0 0	XO	0
OPCD	S	A	B	XO	Rc
OPCD	S	A	B	XO	1
OPCD	S	A	B	XO	0
OPCD	S	A	NB	XO	0
OPCD	S	A	0 0 0 0 0	XO	Rc
OPCD	S	0 0 0 0 0	B	XO	0
OPCD	S	0 0 0 0 0	0 0 0 0 0	XO	0
OPCD	S	0 SR	0 0 0 0 0	XO	0
OPCD	S	A	SH	XO	Rc

OPCD	crfD	0	L	A		B		XO	0
OPCD	crfD	0 0		A		B		XO	0
OPCD	crfD	0 0		crfS	0 0	0 0 0 0 0		XO	0
OPCD	crfD	0 0		0 0 0 0 0		0 0 0 0 0		XO	0
OPCD	crfD	0 0		0 0 0 0 0		IMM	0	XO	Rc
OPCD	TO			A		B		XO	0
OPCD	D			0 0 0 0 0		B		XO	Rc
OPCD	D			0 0 0 0 0		0 0 0 0 0		XO	Rc
OPCD	crbD			0 0 0 0 0		0 0 0 0 0		XO	Rc
OPCD	0 0 0 0 0			A		B		XO	0
OPCD	0 0 0 0 0			0 0 0 0 0		B		XO	0
OPCD	0 0 0 0 0			0 0 0 0 0		0 0 0 0 0		XO	0

Specific Instructions

and*x*	31	S			A	B	28	Rc
andc*x*	31	S			A	B	60	Rc
cmp	31	crfD	0	L	A	B	0	0
cmpl	31	crfD	0	L	A	B	32	0
cntlzd*x* [5]	31	S			A	0 0 0 0 0	58	Rc
cntlzw*x*	31	S			A	0 0 0 0 0	26	Rc
dcba [4]	31	0 0 0 0 0			A	B	758	0
dcbf	31	0 0 0 0 0			A	B	86	0
dcbi [1]	31	0 0 0 0 0			A	B	470	0
dcbst	31	0 0 0 0 0			A	B	54	0
dcbt	31	0 0 0 0 0			A	B	278	0
dcbtst	31	0 0 0 0 0			A	B	246	0
dcbz	31	0 0 0 0 0			A	B	1014	0
eciwx	31	D			A	B	310	0
ecowx	31	S			A	B	438	0
eieio	31	0 0 0 0 0			0 0 0 0 0	0 0 0 0 0	854	0
eqv*x*	31	S			A	B	284	Rc
extsb*x*	31	S			A	0 0 0 0 0	954	Rc
extsh*x*	31	S			A	0 0 0 0 0	922	Rc
extsw*x* [5]	31	S			A	0 0 0 0 0	986	Rc
fabs*x*	63	D			0 0 0 0 0	B	264	Rc

fcfid_x_[5]	63	D		0 0 0 0 0		B	846	Rc
fcmpo	63	crfD	0 0	A		B	32	0
fcmpu	63	crfD	0 0	A		B	0	0
fctid_x_[5]	63	D		0 0 0 0 0		B	814	Rc
fctidz_x_[5]	63	D		0 0 0 0 0		B	815	Rc
fctiw_x_	63	D		0 0 0 0 0		B	14	Rc
fctiwz_x_	63	D		0 0 0 0 0		B	15	Rc
fmr_x_	63	D		0 0 0 0 0		B	72	Rc
fnabs_x_	63	D		0 0 0 0 0		B	136	Rc
fneg_x_	63	D		0 0 0 0 0		B	40	Rc
frsp_x_	63	D		0 0 0 0 0		B	12	Rc
icbi	31	0 0 0 0 0		A		B	982	0
lbzux	31	D		A		B	119	0
lbzx	31	D		A		B	87	0
ldarx[5]	31	D		A		B	84	0
ldux[5]	31	D		A		B	53	0
ldx[5]	31	D		A		B	21	0
lfdux	31	D		A		B	631	0
lfdx	31	D		A		B	599	0
lfsux	31	D		A		B	567	0
lfsx	31	D		A		B	535	0
lhaux	31	D		A		B	375	0
lhax	31	D		A		B	343	0
lhbrx	31	D		A		B	790	0
lhzux	31	D		A		B	311	0
lhzx	31	D		A		B	279	0
lswi[3]	31	D		A		NB	597	0
lswx[3]	31	D		A		B	533	0
lwarx	31	D		A		B	20	0
lwaux[5]	31	D		A		B	373	0
lwax[5]	31	D		A		B	341	0
lwbrx	31	D		A		B	534	0
lwzux	31	D		A		B	55	0
lwzx	31	D		A		B	23	0
mcrfs	63	crfD	0 0	crfS	0 0	0 0 0 0 0	64	0

mcrxr	31	crfD	0 0	0 0 0 0 0		0 0 0 0 0		512	0
mfcr	31	D		0 0 0 0 0		0 0 0 0 0		19	0
mffs*x*	63	D		0 0 0 0 0		0 0 0 0 0		583	Rc
mfmsr [1]	31	D		0 0 0 0 0		0 0 0 0 0		83	0
mfsr [1, 6]	31	D		0	SR	0 0 0 0 0		595	0
mfsrin [1, 6]	31	D		0 0 0 0 0		B		659	0
mtfsb0*x*	63	crbD		0 0 0 0 0		0 0 0 0 0		70	Rc
mtfsb1*x*	63	crfD		0 0 0 0 0		0 0 0 0 0		38	Rc
mtfsfi*x*	63	crbD	0 0	0 0 0 0 0		IMM	0	134	Rc
mtmsr [1, 6]	31	S		0 0 0 0 0		0 0 0 0 0		146	0
mtmsrd [1, 5]	31	S		0 0 0 0 0		0 0 0 0 0		178	0
mtsr [1, 6]	31	S		0	SR	0 0 0 0 0		210	0
mtsrd [1, 6]	31	S		0	SR	0 0 0 0 0		82	0
mtsrin [1, 6]	31	S		0 0 0 0 0		B		242	0
mtsrdin [1, 6]	31	S		0 0 0 0 0		B		114	0
nand*x*	31	S		A		B		476	Rc
nor*x*	31	S		A		B		124	Rc
or*x*	31	S		A		B		444	Rc
orc*x*	31	S		A		B		412	Rc
slbia [1,4,5]	31	0 0 0 0 0		0 0 0 0 0		0 0 0 0 0		498	0
slbie [1,4,5]	31	0 0 0 0 0		0 0 0 0 0		B		434	0
sld*x* [5]	31	S		A		B		27	Rc
slw*x*	31	S		A		B		24	Rc
srad*x* [5]	31	S		A		B		794	Rc
sraw*x*	31	S		A		B		792	Rc
srawi*x*	31	S		A		SH		824	Rc
srd*x* [5]	31	S		A		B		539	Rc
srw*x*	31	S		A		B		536	Rc
stbux	31	S		A		B		247	0
stbx	31	S		A		B		215	0
stdcx. [5]	31	S		A		B		214	1
stdux [5]	31	S		A		B		181	0
stdx [5]	31	S		A		B		149	0
stfdux	31	S		A		B		759	0
stfdx	31	S		A		B		727	0

stfiwx	31	S	A	B	983	0
stfsux	31	S	A	B	695	0
stfsx	31	S	A	B	663	0
sthbrx	31	S	A	B	918	0
sthux	31	S	A	B	439	0
sthx	31	S	A	B	407	0
stswi[3]	31	S	A	NB	725	0
stswx[3]	31	S	A	B	661	0
stwbrx	31	S	A	B	662	0
stwcx.	31	S	A	B	150	1
stwux	31	S	A	B	183	0
stwx	31	S	A	B	151	0
sync	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	598	0
td[5]	31	TO	A	B	68	0
tlbia[1,4]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	370	0
tlbie[1,4]	31	0 0 0 0 0	0 0 0 0 0	B	306	0
tlbsync[1,4]	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	566	0
tw	31	TO	A	B	4	0
xor*x*	31	S	A	B	316	Rc

Table A-37. XL-Form

OPCD	BO		BI		0 0 0 0 0	XO	LK
OPCD	crbD		crbA		crbB	XO	0
OPCD	crfD	0 0	crfS	0 0	0 0 0 0 0	XO	0
OPCD	0 0 0 0 0		0 0 0 0 0		0 0 0 0 0	XO	0

Specific Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
bcctr*x*	19	BO	BI	0 0 0 0 0	528	LK
bclr*x*	19	BO	BI	0 0 0 0 0	16	LK
crand	19	crbD	crbA	crbB	257	0
crandc	19	crbD	crbA	crbB	129	0
creqv	19	crbD	crbA	crbB	289	0
crnand	19	crbD	crbA	crbB	225	0
crnor	19	crbD	crbA	crbB	33	0

cror	19	crbD	crbA	crbB	449	0
crorc	19	crbD	crbA	crbB	417	0
crxor	19	crbD	crbA	crbB	193	0
isync	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150	0
mcrf	19	crfD 0 0	crfS 0 0	0 0 0 0 0	0	0
rfi [1, 6]	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	50	0
rfid [1, 5]	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	18	0

Table A-38. XFX-Form

OPCD	D	spr	XO	0
OPCD	D	0 CRM 0	XO	0
OPCD	S	spr	XO	0
OPCD	D	tbr	XO	0

Specific Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15 16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mfspr [2]	31	D	spr	339	0
mftb	31	D	tbr	371	0
mtcrf	31	S	0 CRM 0	144	0
mtspr [2]	31	D	spr	467	0

Table A-39. XFL-Form

OPCD	0	FM	0	B	XO	Rc

Specific Instructions

Name	0 5	6	7 8 9 10 11 12 13 14	15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31
mtfsf*x*	63	0	FM	0	B	711	Rc

Table A-40. XS-Form

OPCD	S	A	sh	XO	sh	Rc

Specific Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29	30	31
sradi*x* [5]	31	S	A	sh	413	sh	Rc

Table A-41. XO-Form

OPCD	D	A	B	OE	XO	Rc
OPCD	D	A	B	0	XO	Rc
OPCD	D	A	0 0 0 0 0	OE	XO	Rc

Specific Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31
add_x_	31	D	A	B	OE	266	Rc
addc_x_	31	D	A	B	OE	10	Rc
adde_x_	31	D	A	B	OE	138	Rc
addme_x_	31	D	A	0 0 0 0 0	OE	234	Rc
addze_x_	31	D	A	0 0 0 0 0	OE	202	Rc
divd_x_[5]	31	D	A	B	OE	489	Rc
divdu_x_[5]	31	D	A	B	OE	457	Rc
divw_x_	31	D	A	B	OE	491	Rc
divwu_x_	31	D	A	B	OE	459	Rc
mulhd_x_[5]	31	D	A	B	0	73	Rc
mulhdu_x_[5]	31	D	A	B	0	9	Rc
mulhw_x_	31	D	A	B	0	75	Rc
mulhwu_x_	31	D	A	B	0	11	Rc
mulld_x_[5]	31	D	A	B	OE	233	Rc
mullw_x_	31	D	A	B	OE	235	Rc
neg_x_	31	D	A	0 0 0 0 0	OE	104	Rc
subf_x_	31	D	A	B	OE	40	Rc
subfc_x_	31	D	A	B	OE	8	Rc
subfe_x_	31	D	A	B	OE	136	Rc
subfme_x_	31	D	A	0 0 0 0 0	OE	232	Rc
subfze_x_	31	D	A	0 0 0 0 0	OE	200	Rc

Table A-42. A-Form

OPCD	D	A	B	0 0 0 0 0	XO	Rc
OPCD	D	A	B	C	XO	Rc
OPCD	D	A	0 0 0 0 0	C	XO	Rc
OPCD	D	0 0 0 0 0	B	0 0 0 0 0	XO	Rc

Specific Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
fadd*x*	63	D	A	B	0 0 0 0 0	21	Rc
fadds*x*	59	D	A	B	0 0 0 0 0	21	Rc
fdiv*x*	63	D	A	B	0 0 0 0 0	18	Rc
fdivs*x*	59	D	A	B	0 0 0 0 0	18	Rc
fmadd*x*	63	D	A	B	C	29	Rc
fmadds*x*	59	D	A	B	C	29	Rc
fmsub*x*	63	D	A	B	C	28	Rc
fmsubs*x*	59	D	A	B	C	28	Rc
fmul*x*	63	D	A	0 0 0 0 0	C	25	Rc
fmuls*x*	59	D	A	0 0 0 0 0	C	25	Rc
fnmadd*x*	63	D	A	B	C	31	Rc
fnmadds*x*	59	D	A	B	C	31	Rc
fnmsub*x*	63	D	A	B	C	30	Rc
fnmsubs*x*	59	D	A	B	C	30	Rc
fres*x* [4]	59	D	0 0 0 0 0	B	0 0 0 0 0	24	Rc
frsqrte*x* [4]	63	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
fsel*x* [4]	63	D	A	B	C	23	Rc
fsqrt*x* [4]	63	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsqrts*x* [4]	59	D	0 0 0 0 0	B	0 0 0 0 0	22	Rc
fsub*x*	63	D	A	B	0 0 0 0 0	20	Rc
fsubs*x*	59	D	A	B	0 0 0 0 0	20	Rc

Table A-43. M-Form

OPCD	S	A	SH	MB	ME	Rc
OPCD	S	A	B	MB	ME	Rc

Specific Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31
rlwimi*x*	20	S	A	SH	MB	ME	Rc
rlwinm*x*	21	S	A	SH	MB	ME	Rc
rlwnm*x*	23	S	A	B	MB	ME	Rc

Table A-44. MD-Form

OPCD	S	A	sh	mb	XO	sh	Rc
OPCD	S	A	sh	me	XO	sh	Rc

Specific Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26	27 28 29	30	31
rldic*x*[5]	30	S	A	sh	mb	2	sh	Rc
rldicl*x*[5]	30	S	A	sh	mb	0	sh	Rc
rldicr*x*[5]	30	S	A	sh	me	1	sh	Rc
rldimi*x*[5]	30	S	A	sh	mb	3	sh	Rc

Table A-45. MDS-Form

OPCD	S	A	B	mb	XO	Rc
OPCD	S	A	B	me	XO	Rc

Specific Instructions

Name	0 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26	27 28 29 30	31
rldcl*x*[5]	30	S	A	B	mb	8	Rc
rldcr*x*[5]	30	S	A	B	me	9	Rc

[1] Supervisor-level instruction
[2] Supervisor- and user-level instruction
[3] Load/store string/multiple instruction
[4] Optional instruction
[5] 64-bit instruction
[6] Optional 64-bit bridge instruction

Appendix B
Invalid Instruction Forms

This appendix describes how invalid instructions are treated by the PowerPC 620 microprocessor.

B.1 Invalid Forms Excluding Reserved Fields

Table B-1 illustrates the invalid instruction forms of the PowerPC architecture that are not a result of a nonzero reserved field in the instruction encoding.

Table B-1. Invalid Forms (Excluding Reserved Fields)

Mnemonic	$BO_2 = 0$	rA = 0 or rA = rD	rA = 0	rA = r T = 0	rA in Range	rA or rB in Range	L = 1	SPR Not Implemented
bcctr	X							
bcctrl	X							
lbzu		X						
lbzux		X						
lhzu		X						
lhzux		X						
lhau		X						
lhaux		X						
lwzu		X						
lwzux		X						
stbu			X					
stbux			X					
sthu			X					
sthux			X					
stwu			X					
stwux			X					
lmw				X	X			

Table B-1. Invalid Forms (Excluding Reserved Fields) (Continued)

Mnemonic	BO$_2$ = 0	rA = 0 or rA = rD	rA = 0	rA = r T = 0	rA in Range	rA or rB in Range	L = 1	SPR Not Implemented
lswi				X	X			
lswx				X		X		
cmpi							X	
cmp							X	
cmpli							X	
cmpl							X	
mtspr								X
mfspr								X
LFSU			X					
lfsux			X					
lfdu			X					
lfdux			X					
stfsu			X					
stfsux			X					
stfdu			X					
stfdux			X					

B.2 Invalid Forms with Reserved Fields (Bit 31 Exclusive)

Table B-2 lists the invalid instruction forms of the PowerPC architecture that result from a nonzero reserved field in the instruction encoding. This table takes into consideration all reserved fields in an instruction that must be zero, excluding only those instructions that would become invalid if only bit 31 were set. Note that any combination of a one being detected in the instructions field(s) marked X results in an invalid form.

The **tlbsync** instruction has the same opcode and format as the **sync** instruction. Setting bit 31 in the instruction indicates a **tlbsync**.

Table B-2. Invalid Forms with Reserved Fields (Bit 31 Exclusive)

Mnemonic	6	6 to 10	6 to 15	6 to 20	6 to 29	9	9 to 10	9 to 15	11	11 to 15	11 to 20	14 to 20	15	16 to 20	20	21	21 to 25	31
bclr														X				
bclrl														X				
bcctr														X				
bcctrl														X				
sc					X													X
mcrf							X					X						X
sync				X														*
addme[o][.]														X				
subfme[o][.]														X				
addze[o][.]														X				
subfze[o][.]														X				
neg[o][.]														X				
mulhw[u][.]																X		
cmpi						X												X
cmp						X												
cmpli						X												X
cmpl						X												
extsb[.]														X				
extsh[.]														X				
cntlzw[.]														X				
mtcrf									X						X			X
mcrxr						X					X							X
mtpmr											X							X
mfpmr											X							X
fmr[.]										X								
fneg[.]										X								
fabs[.]										X								
fnabs[.]										X								
fadd[.]																	X	
fadds[.]																	X	

Table B-2. Invalid Forms with Reserved Fields (Bit 31 Exclusive) (Continued)

Mnemonic	6	6 to 10	6 to 15	6 to 20	6 to 29	9	9 to 10	9 to 15	11	11 to 15	11 to 20	14 to 20	15	16 to 20	20	21	21 to 25	31
fsub[.]																	X	
fsubs[.]																	X	
fmul[.]														X				
fmuls[.]														X				
fdiv[.]																	X	
fdivs[.]																	X	
frsp[.]										X								
fctiw[.]										X								
fctiwz[.]										X								
fcmpu							X											X
fcmpuo							X											X
mffs[.]											X							
mcrfs							X					X						X
mtfsfi[.]								X							X			
mtfsf[.]	X												X					
mtfsb0[.]											X							
mtfsb1[.]											X							
icbi		X																X
isync				X														X
dcbt		X																X
dcbtst		X																X
dcbz		X																X
dcbst		X																X
dcbf		X																X
eieio				X														X
mftb											X							X
mftbu											X							X
rfi				X														X
mtmsr											X							X
mfmsr											X							X
dcbi		X																X

Table B-2. Invalid Forms with Reserved Fields (Bit 31 Exclusive) (Continued)

Mnemonic	6	6 to 10	6 to 15	6 to 20	6 to 29	9	9 to 10	9 to 15	11	11 to 15	11 to 20	14 to 20	15	16 to 20	20	21	21 to 25	31
mtsr									X					X				X
mfsr									X					X				X
mtsrin										X								X
mfsrin										X								X
tlbie			X															X
mttb			X															X
mttbu			X															X
tlbsync				X														*

B.3 Invalid Form with Only Bit 31 Set

The following instructions generate invalid instruction forms if only bit 31 is set in the instruction:

- **cror**
- **crxor**
- **crnand**
- **crnor**
- **crandc**
- **creqv**
- **crorc**
- **lbzx**
- **lbzux**
- **lhzx**
- **lhzux**
- **lhax**
- **lhaux**
- **lwzx**
- **lwzux**
- **stbx**
- **stbux**
- **sthx**
- **sthux**
- **stwx**
- **stwux**
- **lhbrx**

- **lwbrx**
- **sthbrx**
- **stwbrx**
- **lswi**
- **lswx**
- **stswi**
- **stswx**
- **lwarx**
- **tw**
- **mtspr**
- **mfspr**
- **lfsx**
- **lfsux**
- **lfdx**
- **lfdux**
- **stfsx**
- **stfsux**
- **stfdx**
- **stfdux**

B.4 Invalid Forms from Invalid BO Field Encodings

The following list illustrates the invalid BO fields for the conditional branch instructions (**bc**, **bca**, **bcl**, **bcla**, **bclr**, **bclrl**, **bcctr**, and **bcctrl**). Specifying a conditional branch instruction with one of these fields results in a invalid instruction form. Note that entries with the *y* bit represent two possible instruction encodings. Invalid BO field encodings are as follows:

- 0011y
- 0111y
- 1100y
- 1101y
- 10101
- 10110
- 10111
- 11100
- 11101
- 11110
- 11111

The 620 treats the bits listed above as causing an invalid form as "don't cares."

Appendix C Hardware Configuration

This appendix is devoted to a consolidated collection of narrower requirements to bring up a PowerPC 620 in a defined state. Included is a summary of hardware configuration signals, their use and the prescribed sequence to use on a 620 processor during and after reset. This involves bus configuration, the L2 interface and other tidbits.

C.1 Hardware Configuration Signals

Table C-1 provides a summary of signals used for hardware configuration. Additional information can be found in Chapter 7, "Signal Descriptions." All of these inputs must be set at power-on reset. Changing these signals with hard reset deasserted will have no effect.

Table C-1. Hardware Configuration Signals Summary

Signal Name	Purpose	Typical Values to Use
SHIFTGATE	This signal is used during LSSD testing to enable shift clocks. In system applications this signal defines the MSB for BUSRATIO.	1
BUSRATIO1	This is a dedicated signal to define the LSB for BUSRATIO on the 620. Acceptable ratios are—10:2:1 mode, 11:3:1 mode, 00:4:1 mode. **Note:** A setting of 01 is invalid and will result in unpredictable behavior.	0
BUSCLKGTL	This is a dedicated signal to set the I/O interface for the input clock to the 620. For CMOS single-ended clock input this net should be tied low. Then drive the input clock into the BUSCLK input signal and ground the BUSCLK_B signal. If using a differential input pair of clocks with a GTL source, tie the BUSCLKGTL input high and source the clock in on the BUSCLK, BUSCLK_B pair of inputs.	0
BUSLAEN	This is another dedicated input that is used to enable or disable the late address enable mode. When active high, this input indicates that the address associated with an $\overline{\text{ABG}}$ should be enabled 1 PCLK after the beginning of the next BUSCLK cycle. When deasserted, the address should be asserted at the beginning of the next BUSCLK cycle. For further information refer to Section 2.1.2.4, "Bus Status and Control Register (BUSCSR)."	0
BUSRESPTEN[0–1]	These two inputs are dedicated signals to set the latency between valid ASTATOUT and ARESPOUT being driven and ASTATIN/ARESPIN being sampled. Valid decodes are 01 – 1 BUSCLK, 10 – 2 BUSCLKS, 11 – 3 BUSCLKS. **Note:** 00 is invalid.	10

Table C-1. Hardware Configuration Signals Summary (Continued)

Signal Name	Purpose	Typical Values to Use
BUSDX	This input when asserted high forces the 620 to run in DX mode. In this mode, bus data transfers occur over the DataHigh bus only in double word format. The DataLow bus is therefore dibbled. The normal mode power on default is to enable quadword operations with DX mode off.	0
BUSPID[0–4]	BUSPID[0–4] comprises a five bit processor ID number on the bus. This value is sampled during hard reset and used to form the bus tag for all subsequent bus operations. The first three bits of this bus are formed from dedicated signals.	000
DRVRINHIBIT1	Shared signal between LSSD function to disable drivers and hard reset function of BUSPID[3]	0
DRVRINHIBIT2	Shared signal between LSSD function to disable drivers and hard reset function of BUSPID[4]	0
RCVRINHIBIT	These final three pins are shared between LSSD function and PLL configuration during power up. During power up, that is, when hard reset is asserted, this signal is used to decode the function of PllVcoBand.	0
L1_TESTCLK	During hard reset this signal is decoded as PllPumpLow	0
L2_TESTCLK	During hard reset this signal is decoded as PllVcoDiv.	0

C.2 Processor Start Up

To illustrate what happens when the 620 is powered up, assume the default values are asserted during hard reset as illustrated in Table C-1. Given this back drop and input BUSCLK of 50 MHz, CMOS single ended, the 620 will be running at an internal frequency of 100 MHz.

The following sequence is important to follow for safe operation:

1. Assert hard reset prior to applying power or as the power suply is ramping up. To accomplish this a simple method is via a pull down resistor. This is important to avoid internal bus contention and put the internal caches in a safe state.
2. After power stabilizes, hold reset for 2ms to ensure the internal PLL has time to lock. During this time, the 620 is scanning all internal latches to initial values and getting set to branch to the system reset vector.
3. Upon deassertion of reset, 620 will start fetching at address FFF00100, expecting to find the system interrupt handler. In so doing, the 620 will assert that address on the lower 32 bits of address along with $\overline{\text{ABR}}$ and $\overline{\text{DBR}}$. In response 620 will expect $\overline{\text{ABG}}$ and $\overline{\text{DBG}}$.

C.3 Setting Bus and Cache Configuration Registers

Not all 620 configuration registers are set during the hardware reset process. Several, including the BUSCSR, L2CR and HID0 registers require a prescribed software sequence to be set. This sequence is illustrated in PowerPC pseudo-assembly code. It assumes that the HID0, BUSCSR, and L2CR registers are at reset state, and that address translation is disabled. It also assumes that the run state has both L1 caches and snooping enabled.

Note that, in order to program the BUSCSR and L2CR correctly, the bus must be running incoherently. In particular, since snooping is disabled, there can be no master or slave running on the bus that could rerun the SYNC operations. The code to form an incoherent barrier before enabling snooping for processors in an multiprocessor system is not included here.

```
# this number is guaranteed to run a countdown loop for at least
# 100 us on a 200 MHz machine (20,000 cycles)
const int32 one_hundred_us = 0x4e20;


#
# initialize HID0, BUSCSR and L2CR registers
# assume the values to be set are in some memory location
#           called hwavprr_<register> offset from the address in register 2
# Note: each sync, mtspr, isync sequence for setting buscsr, hid0 or l2cr
#           must be aligned on a quadword boundary
#
# step 1: set all buscsr parameters except snpen
# step 2: set all hid0 parameters
# step 3: set all L2CR except l2clc, l2pllen, l2initb
# step 4: set l2pllen and wait 100 us.
# step 5: set all L2CR except l2initb
# step 6: touch all L2 lines
# step 7: set l2initb
# step 8: flush L2
# step 9: enable snooping
#
# step 1 - set buscsr except snpen
#
ld r3, hwavprr_BUSCSR(r2);
andi. r3, r3, 0xfffffeff;  # don't enable snooping
.align 16;                                          # align to QW
sync;
mtbuscsr r3;
isync;
```

```
#
# step 2 - set all hid0 parameters
#
lwz r3, hwavprr_HID0(r2);
ori r3, r3, 0x00000c00; # add flash invalidates to config
.align 16;
sync;
mthid0 r3;
isync;
#
# steps 3-8 - set and initialize L2
#
ld r3, hwavprr_L2CR(r2);
# skip init if l2clc<0> is 0
andis. r4, r3, 0x0001;
cmpwi cr0, r4, 0;
beq cr0, l2_done;
#
# Step 3 - set l2cr except write 0's to l2clc, l2pllen, l2init_b
#
addi r5, r0, 0;
ori r5, r5, 0x1fff;
oris r5, r5, 0xfffe;
and r4, r3, r5;
.align 16;
sync;
mtl2cr r4;
isync;
#
# step 4 - assert l2pllen and wait for pll to lock
#
addi r5, r0, 0;
ori r5, r0, 0x5fff;
oris r5, r5, 0xfffe;
and r4, r3, r5;
.align 16;
sync;
```

```
        mtl2cr r4;
        isync;
        # 100 us loop
        addi r4, r0, one_hundred_us;
        mtctr r4;
loop:
        bdnz.y loop;
        #
        # step 5: enable L2 in init mode
        #
        addi r5, r0, 0;
        ori r5, r5, 0xdfff;
        oris r5, r5, 0xffff;
        and r4, r3, r5;
        .align 16;
        sync;
        mtl2cr r4;
        isync;
        #
        # Step 6
        #
        # touch all of L2 (16384*(l2size+1) cache blocks)
        # Note: these loops assume that the memory starting at location
        # zero and of size == l2 exists.
        rlwinm r4, r3, 12, 20, 31;# srwi r4, r4, 20
        andi. r4, r4, 0xf;
        addi r4, r4, 1;
        rlwinm r4, r4, 14, 0, 17;# slwi r4, r4, 14 <=> n*16384
        mtctr r4;
        addi r5, r0, 0;
initloop:
        lwz r6, 0(r5);
        addi r5, r5, 64;
        bdnz.y initloop;
        # clear l2sr
        addi r6, r0, 0;
        sync;                    # no align needed for l2sr set
```

```
            mtl2sr r6;
            isync;
            #
            # Step 7: fully enable L2
            #
            .align 16;
            sync;
            mtl2cr r3;
            isync;
            #
            # Step 8: flush L2
            #
            mtctr r4;
            addi r5, r0, 0;
flushloop:
            dcbf r0, r5;
            addi r5, r5, 64;
            bdnz.y flushloop;
l2_done:
            #
            # step 9 - enable snooping
            #
            mfbuscsr r3;
            ori r3, r3, 0x100;      # enable snooping
            .align 16;
            sync;
            mtbuscsr r3;
            isync;
            #
            # end special bringup code
            #
```

C.4 Enabling Address Translation

Setting the MSR also requires a special code sequence. Address translation should be enabled after the initialization of the SPRs (HID0, BUSCSR, L2CR), the L2 interface, and after enabling snooping. The following code is recommended to make sure that setting the IR and DR bits in the MSR register has completed execution before any new instructions are prefetched (the reason for the 16 Nops).

```
mfmsr r12
mask and set IR and DR
mtmsr r12
sync
isync
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
```

C.5 Flushing Data from the L1 Data Cache

Flushing data from the data cache can be accomplished in two ways. First, the **dcbst** or **dcbf** instructions can be used to force a particular address from the data cache back to main memory. If you know the addresses of the data to be flushed, this is the preferred method.

For example, if a page of instructions has been loaded into the data cache and must be flushed to main memory in order to be executed, a simple routine could be written to do a **dcbf** to each line in that page.

Flushing the entire data cache regardless of its contents is a more difficult proposition. This may be required, for example, if the processor is being placed in nap mode. The **dcbf** and

dcbst instructions cannot be used because they will only force copy-backs for specific addresses, and you cannot practically do a **dcbf** for the entire physical address space. The better solution is to perform enough loads to guarantee that all previous data has been replaced in the data cache with new data. The difficulty with this method is that accidental hits, aliased hits, or ECAM matches can confuse the PLRU algorithm so that more loads are required than might be expected.

If loads are done from an address space that is known not to be in the data cache, then the cache can be completely flushed by doing 13 loads to each camlet, each with a unique ECAM value (that is, EA[44–51] must be unique). If, however, the data loaded to force copy-backs might already be in the cache, making hits and aliased hits possible, then it is required to do 20 loads with unique ECAM values. In addition, since any of the loads may have hit rather than forced a copy-back, it is also necessary to perform a **dcbf** on each address loaded.

The code segment below should perform the necessary operations for a complete flush of the data cache. Notice that this code segment assumes that data being loaded may in fact be in the cache, so 20 loads are performed for each camlet, and **dcbf** instructions are performed after the loads. The comments indicate where the code can be modified if the data being loaded is known not to be in the cache.

Note that if the **dcbf** loop is performed, the final state of all lines in the data cache will be invalid. If the **dcbf** instruction is replaced with **dcbst** or if loads are done from unused memory and the **dcbf** loop is skipped altogether, then the final state of all lines in the data cache will be shared or exclusive.

```
full_cache_flush:

            .set        FLUSH_START_HI,0x0000 # Change to whatever starting

            .set        FLUSH_START_LO,0x0000 # address you like....

            .set        HID0,0x3F0# SPR value for HID0

            .set        BLOCK_SIZE,0x40# 64 bytes per line

            .set        LOOP_COUNT,20# This number should be 13 if loads

                                    # are done from unused memory,

                                    # or 20 if not.


            mfmsr       G0          # Save MSR value to G0

            mfspr       G1,HID0     # Save HID0 value to G1


            rlwinm      G2,G0,0,17,15# Disable EE

            rlwinm      G3,G1,0,16,17# Leave I and D caches on

                                    # Turn superscalar mode off

                                    # No speculative ifetch
```

```
                                # No dynamic branch prediction or
                                #    branch history table updates

          .align     4          # Align to quad-word boundary
          sync                  # Update SPR's
          mtmsr      G2         #
          mtspr      G3,HID0    # G2 and G3 are free to be reused
          isync                 #

          addi       G2,G0,0x1000# G2 = ECAM incrementer
          addi       G3,G0,0    # Clear G3
          oris       G4,G3,FLUSH_START_HI# G4 = start address of data
          ori        G4,G4,FLUSH_START_LO# used to flush cache
          subf       G4,G2,G4   # minus initial offset.
          addi       G3,G0,0x0FC0# G3 = camlet offset
                                # Start at 63 and work back to 0
          addi       G5,G0,LOOP_COUNT# G5 = number of loads/flushes
          add        G6,G0,BLOCK_SIZE# G6 = line size

camlet_flush_loop:

          mtctr      G5         # ctr = # of load's to do
          add        G7,G4,G3   # add camlet offset to address
load_loop:
          lbzux      G8,G7,G2   # do 20 loads to the same camlet
          bc         16,0,load_loop# each with a unique ECAM value
                                # until CTR=0

          # This loop can be removed if data is loaded from unused memory
          mtctr      G5         # ctr = # of dcbf's to do
          add        G7,G4,G3   # reset G7 to start address
dcbf_loop:
          dcbf       G7,G2      # Flush the line from the cache
          add        G7,G7,G2   # in case the line actually hit
          bc         16,0,dcbf_loop# Loop until CTR=0

          subf.      G3,G6,G3   # Subtract line size from G3 to
```

```
                        # point to next camlet.
    bne     camlet_flush_loop# Repeat camlet flush loop
                        # until G3<0
    .align  4
    sync                # restore SPR's
    mtmsr   G0          #
    mtspr   G1,HID0     #
    sync                #
```

If the **dcbf** instruction is changed to a **dcbst** or if the **dcbf** loop is not executed, and it is desired to leave all lines in the data cache in the invalid state, the following sequence can be appended to the previous procedure.

- Reset SPR (HID0[DCE]) to disable data cache
- Set SPR(HID0[DCEFI]) to invalidate the whole data cache

Appendix D
Bus Protocol Livelock Scenarios

In any multiprocessor system, livelock scenarios are possible. Measures to avoid these scenarios include, but are not limited to, the following:

- Special-purpose hardware in the PowerPC 620 such as pacing counters.
- Special-purpose hardware in the system such as a smart arbiter or the use of high-priority requests.

The remainder of this appendix documents livelocks encountered during either simulation or hardware debug.

D.1 2-way Multiprocessing—Imprecise Collision Hit Between Processors

This livelock was encountered during 2-way multiprocessing (MP) simulation where the second processor was retried all the time by the first processor. The situation is as follows:

- The codes sequence for both processors are illustrated in Table D-1:

Table D-1. Processor Code Sequence

Processor 1	Processor 2
Loop:	
Load A	Load A
....	....
Load B	Store A
....	
Branch back to “Loop:” if A is not updated by processor 2	

- Other conditions required for this scenario to occur include:
  - — Both A and B have the same 20 least significant bits for their effective addresses. Based on the current L1 cache placement policy, this will cause A and B to map to the same L1 entry. In other words, fetching B will force the invalidation of A in the L1 cache and vice versa.
  - — A and B have the same 20 least significant bits for the physical addresses. This will cause imprecise collision detection between B, from processor 1, and A, from processor 2.
  - — The size of L2 cache is greater than 1 Mbyte. This will allow both A and B to reside in the L2 cache at the same time.
  - — Both A and B are located in the L2 cache of processor 1, but A is not in the L2 cache of processor 2. In this case, Processor 2 will go out to the 620 bus to fetch the cache line for A when "load A" is executed.
- After the first iteration of the loop in processor 1, B (not A) is in the L1 cache. In order to load A, DCMMU has to send a request to BIU to fetch A from the L2 cache. After that, A (not B) is in the L1 cache, which will cause DCMMU to request B from the L2 cache when "Load B" is executed next. Therefore, DCMMU will keep sending requests to L2 for either A or B alternatively as long as processor 1 stays within that loop. During the iteration of each loop, if processor 2's fetching request to A which is sent to 620 bus is always snooped by processor 1 within a particular timing window, then processor 1's collision detection logic will win and retry processor 2 every time. Since processor 2 is retried all the times, it cannot load A and update it. On the other hand, if A is not updated, processor 2 will stay in the loop forever and also keeps processor 1 from making any progress. This causes a livelock problem.

This livelock may be broken by using the 620 pacing counters or by having a smart arbiter detect the sequence and change the grant pattern.

D.2 4-way Multiprocessing—All Processors Running lwarx and stwcx.

This livelock was encountered during hardware debug. Processors 0–3 (P0–P3) were all attempting to access the same cache line. The situation is as follows:

1. P0 is waiting for a lock variable to update and is using a **lwarx** to check the value of the memory.
   P1 is executing a similar code sequence.
   P2 has a reservation and is attempting to execute a **stwcx** to the shared cache line.
   P3 is executing a similar code sequence.
2. When P0 accesses L2, it temporarily takes ownership of the cache line so it moves its CD state to IN-BUSY.

3. While P0 is IN-BUSY, it snoops the **stwcx** from P3. This **stwcx** fails CD since the P0 **lwarx** is IN-BUSY and the **stwcx** is subsequently ARESPIN Retried. P0 runs the **lwarx** on the bus, finds that the value is unchanged and loops back to the **lwarx**.
4. The same sequence exists for the P1–P2 pairs.
5. When P0 and P1 run the **lwarx** operations on the bus, they line up in previous adjacent windows and each takes turns ASTATIN Retrying its operation.

This livelock may be broken by using the 620 pacing counters or by having a smart arbiter detect the sequence and change the grant pattern.

D.3 4-way Multiprocessing—3 Processors Reading Line with Pending W = 1 Stores to L2

A variation of this livelock was encountered during 4-way MP simulation. Processor 0 (P0) is processing two write-through stores and Processors 1–3 (P1–P3) are all attempting to read the same lines. The situation is as follows:

1. The L2 is 2:1, single-synchronous. BUSTLAR=3; BUSRESPTEN=2.
2. P0 moves a W=1 store to Address A to the Store Buffer. The line is valid in L2. The L2 write is pending.
3. P0 moves a W=1 store to Address B to the Store Buffer. The line is valid in L2. The L2 write is pending.
4. P1 generates a bus read for Address A. P0 snoops this operation and generates an L2 read.
5. P2 generates a bus read for Address B. P0 snoops this operation and generates an L2 read. The P1 and P2 addresses are adjacent.
6. P3 generates a bus read for Address A. P0 snoops this operation and generates an L2 read. The P2 and P3 addresses are adjacent.
7. P1 is AResp Retried since Address A is held in the Store Buffer.
8. P2 is AResp Retried since Address B is held in the Store Buffer.
9. P3 is AResp Retried since Address A is held in the Store Buffer.
10. Since L2 reads are higher priority than L2 writes, the pending P0 stores cannot access the L2.
11. P1, P2 and P3 continually get AResp Retried and repeat this access pattern.

Note that the livelock could also occur for other types of P0 stores. Examples include consecutive L1 deallocates to L2 or two snooped "Clean" operations which cause pushes to the bus and the L2.

This livelock may be broken by using the 620 pacing counters or by having a smart arbiter detect the sequence and change the grant pattern.

INDEX

INDEX

INDEX

INDEX

INDEX

6/1/97

MOTOROLA AUTHORIZED DISTRIBUTOR & WORLDWIDE SALES OFFICES

NORTH AMERICAN DISTRIBUTORS

UNITED STATES

ALABAMA

Huntsville	
Arrow/Schweber Electronics	(205)837–6955
FAI	(205)837–9209
Future Electronics	(205)830–2322
Hamilton/Hallmark	(205)837–8700
Newark	(205)837–9091
Wyle Electronics	(205)830–1119

ARIZONA

Phoenix	
FAI	(602)731–4661
Future Electronics	(602)968–7140
Hamilton/Hallmark	(602)736–7000
Wyle Electronics	(602)804–7000
Tempe	
Arrow/Schweber Electronics	(602)431–0030
Newark	(602)966–6340
PENSTOCK	(602)967–1620

CALIFORNIA

Agoura Hills	
Future Electronics	(818)865–0040
Calabassas	
Arrow/Schweber Electronics	(818)880–9686
Wyle Electronics	(818)880–9000
Culver City	
Hamilton/Hallmark	(310)558–2000
Garden Grove	
Newark	(714)893–4909
Irvine	
Arrow/Schweber Electronics	(714)587–0404
FAI	(714)753–4778
Future Electronics	(714)453–1515
Hamilton/Hallmark	(714)789–4100
Wyle Laboratories Corporate	(714)753–9953
Wyle Electronics	(714)789–9953
Los Angeles	
FAI	(818)879–1234
Manhattan Beach	
PENSTOCK	(310)546–8953
Newberry Park	
PENSTOCK	(805)375–6680
Palo Alto	
Newark	(415)812–6300
Rancho Cordova	
Wyle Electronics	(916)638–5282
Riverside	
Newark	(909)980–2105
Rocklin	
Hamilton/Hallmark	(916)632–4500
Sacramento	
FAI	(916)782–7882
Newark	(916)565–1760
San Diego	
Arrow/Schweber Electronics	(619)565–4800
FAI	(619)623–2888
Future Electronics	(619)625–2800
Hamilton/Hallmark	(619)571–7540
Newark	(619)453–8211
PENSTOCK	(619)623–9100
Wyle Electronics	(619)558–6600
San Jose	
Arrow/Schweber Electronics	(408)441–9700
Arrow/Schweber Electronics	(408)428–6400
FAI	(408)434–0369
Future Electronics	(408)434–1122
Santa Clara	
Wyle Electronics	(408)727–2500
Santa Fe Springs	
Newark	(310)929–9722
Sierra Madre	
PENSTOCK	(818)355–6775
Sunnyvale	
Hamilton/Hallmark	(408)435–3600
PENSTOCK	(408)730–0300
Thousand Oaks	
Newark	(805)449–1480
Woodland Hills	
Hamilton/Hallmark	(818)594–0404

COLORADO

Lakewood	
FAI	(303)237–1400
Future Electronics	(303)232–2008
Denver	
Newark	(303)373–4540
Englewood	
Arrow/Schweber Electronics	(303)799–0258
Hamilton/Hallmark	(303)790–1662
PENSTOCK	(303)799–7845
Thornton	
Wyle Electronics	(303)457–9953

CONNECTICUT

Bloomfield	
Newark	(203)243–1731
Cheshire	
FAI	(203)250–1319
Future Electronics	(203)250–0083
Hamilton/Hallmark	(203)271–5700
Wallingford	
Arrow/Schweber Electronics	(203)265–7741
Wyle Electronics	(203)269–8077

FLORIDA

Altamonte Springs	
Future Electronics	(407)865–7900
Clearwater	
FAI	(813)530–1665
Future Electronics	(813)530–1222
Deerfield Beach	
Arrow/Schweber Electronics	(305)429–8200
Wyle Electronics	(305)420–0500
Ft. Lauderdale	
FAI	(305)428–9494
Future Electronics	(305)436–4043
Hamilton/Hallmark	(954)677–3500
Newark	(305)486–1151
Lake Mary	
Arrow/Schweber Electronics	(407)333–9300
Largo/Tampa/St. Petersburg	
Hamilton/Hallmark	(813)507–5000
Newark	(813)287–1578
Wyle Electronics	(813)376–3004
Maitland	
Wyle Electronics	(407)740–7450
Orlando	
FAI	(407)865–9555
Newark	(407)896–8350
Tallahassee	
FAI	(904)668–7772
Tampa	
Newark	(813)287–1578
PENSTOCK	(813)247–7556
Winter Park	
Hamilton/Hallmark	(407)657–3300
PENSTOCK	(407)672–1114

GEORGIA

Atlanta	
FAI	(404)447–4767
Duluth	
Arrow/Schweber Electronics	(404)497–1300
Hamilton/Hallmark	(770)623–4400
Norcross	
Future Electronics	(770)441–7676
Newark	(770)448–1300
PENSTOCK	(770)734–9990
Wyle Electronics	(770)441–9045

IDAHO

Boise	
FAI	(208)376–8080
Newark	(208)342–4311

ILLINOIS

Addison	
Wyle Laboratories	(708)620–0969
Arlington Heights	
Hamilton/Hallmark	(847)797–7300
Chicago	
FAI	(708)843–0034
Newark Electronics Corp.	1–800–4NEWARK
Hoffman Estates	
Future Electronics	(708)882–1255
Itasca	
Arrow/Schweber Electronics	(708)250–0500
Palatine	
PENSTOCK	(708)934–3700
Schaumburg	
Newark	(708)310–8980

INDIANA

Indianapolis	
Arrow/Schweber Electronics	(317)299–2071
Hamilton/Hallmark	(317)575–3500
FAI	(317)469–0441
Future Electronics	(317)469–0447
Newark	(317)259–0085
Ft. Wayne	
Newark	(219)484–0766
PENSTOCK	(219)432–1277

IOWA

Cedar Rapids	
Newark	(319)393–3800

KANSAS

Kansas City	
FAI	(913)381–6800
Lenexa	
Arrow/Schweber Electronics	(913)541–9542
Olathe	
PENSTOCK	(913)829–9330
Overland Park	
Future Electronics	(913)649–1531
Hamilton/Hallmark	(913)663–7900
Newark	(913)677–0727

MARYLAND

Baltimore	
FAI	(410)312–0833
Columbia	
Arrow/Schweber Electronics	(301)596–7800
Future Electronics	(410)290–0600
Hamilton/Hallmark	(410)720–3400
PENSTOCK	(410)290–3746
Wyle Electronics	(410)312–4844
Hanover	
Newark	(410)712–6922

MASSACHUSETTS

Bedford	
Wyle Electronics	(617)271–9953
Boston	
Arrow/Schweber Electronics	(508)658–0900
FAI	(508)779–3111
Newark	1–800–4NEWARK
Bolton	
Future Corporate	(508)779–3000
Burlington	
PENSTOCK	(617)229–9100
Peabody	
Hamilton/Hallmark	(508)532–3701
Woburn	
Newark	(617)935–8350

continued on next page

For changes to this information contact Technical Publications at FAX (602) 244-6560

AUTHORIZED DISTRIBUTORS – continued

UNITED STATES – continued

MICHIGAN

Detroit	FAI	(313)513–0015
	Future Electronics	(616)698–6800
Grand Rapids	Newark	(616)954–6700
Livonia	Arrow/Schweber Electronics	(810)455–0850
	Future Electronics	(313)261–5270
	Hamilton/Hallmark	(313)416–5800
Troy	Newark	(810)583–2899

MINNESOTA

Bloomington	Wyle Electronics	(612)853–2280
Burnsville	PENSTOCK	(612)882–7630
Eden Prairie	Arrow/Schweber Electronics	(612)941–5280
	FAI	(612)947–0909
	Future Electronics	(612)944–2200
	Hamilton/Hallmark	(612)881–2600
Minneapolis	Newark	(612)331–6350

MISSOURI

Earth City	Hamilton/Hallmark	(314)291–5350
St. Louis	Arrow/Schweber Electronics	(314)567–6888
	Future Electronics	(314)469–6805
	FAI	(314)542–9922
	Newark	(314)453–9400

NEW JERSEY

Bridgewater	PENSTOCK	(908)575–9490
East Brunswick	Newark	(908)937–6600
Fairfield	FAI	(201)331–1133
Marlton	Arrow/Schweber Electronics	(609)596–8000
	FAI	(609)988–1500
	Future Electronics	(609)596–4080
Mt. Laurel	Hamilton/Hallmark	(609)222–6400
	Wyle Electronics	(609)439–9110
Oradell	Wyle Electronics	(201)261–3200
Pinebrook	Arrow/Schweber Electronics	(201)227–7880
	Wyle Electronics	(201)882–8358
Parsippany	Future Electronics	(201)299–0400
	Hamilton/Hallmark	(201)515–1641

NEW MEXICO

Albuquerque	Hamilton/Hallmark	(505)293–5119
	Newark	(505)828–1878

NEW YORK

Bohemia	Newark	(516)567–4200
Hauppauge	Arrow/Schweber Electronics	(516)231–1000
	FAI	(516)348–3700
	Future Electronics	(516)234–4000
	Hamilton/Hallmark	(516)434–7400
	Newark	1–800–4NEWARK
	PENSTOCK	(516)724–9580
	Wyle Electronics	(516)231–7850
Henrietta	Wyle Electronics	(716)334–5970
Konkoma	Hamilton/Hallmark	(516)737–0600
Pittsford	Newark	(716)381–4244
Rochester	Arrow/Schweber Electronics	(716)427–0300
	Future Electronics	(716)387–9550
	FAI	(716)387–9600
	Hamilton/Hallmark	(716)272–2740
Syracuse	FAI	(315)451–4405
	Future Electronics	(315)451–2371
	Newark	(315)457–4873

NORTH CAROLINA

Charlotte	FAI	(704)548–9503
	Future Electronics	(704)547–1107
	Newark	(704)535–5650
Morrisville	Wyle Electronics	(919)469–1502
Raleigh	Arrow/Schweber Electronics	(919)876–3132
	FAI	(919)876–0088
	Future Electronics	(919)790–7111
	Hamilton/Hallmark	(919)872–0712
	Newark	1–800–4NEWARK

OHIO

Centerville	Arrow/Schweber Electronics	(513)435–5563
Cleveland	FAI	(216)446–0061
	Newark	(216)391–9330
Columbus	Newark	(614)326–0352
Dayton	FAI	(513)427–6090
	Future Electronics	(513)426–0090
	Hamilton/Hallmark	(513)439–6735
	Newark	(513)294–8980
Mayfield Heights	Future Electronics	(216)449–6996
Miamisburg	Wyle Electronics	(937)436–9953
Solon	Arrow/Schweber Electronics	(216)248–3990
	Hamilton/Hallmark	(216)498–1100
	Wyle Electronics	(216)248–9996
Worthington	Hamilton/Hallmark	(614)888–3313

OKLAHOMA

Tulsa	FAI	(918)492–1500
	Hamilton/Hallmark	(918)459–6000
	Newark	(918)252–5070

OREGON

Beaverton	Arrow/Almac Electronics Corp.	(503)629–8090
	Future Electronics	(503)645–9454
	Hamilton/Hallmark	(503)526–6200
Portland	FAI	(503)297–5020
	Newark	(503)297–1984
	PENSTOCK	(503)646–1670
	Wyle Electronics	(503)598–9953

PENNSYLVANIA

Coatesville	PENSTOCK	(610)383–9536
Ft. Washington	Newark	(215)654–1434
Pittsburgh	Arrow/Schweber Electronics	(412)963–6807
	Newark	(412)788–4790

TENNESSEE

Knoxville	Newark	(615)588–6493

TEXAS

Austin	Arrow/Schweber Electronics	(512)835–4180
	Future Electronics	(512)502–0991
	FAI	(512)346–6426
	Hamilton/Hallmark	(512)219–3700
	Newark	(972)458–2528
	PENSTOCK	(512)346–9762
	Wyle Electronics	(512)833–9953
Benbrook	PENSTOCK	(817)249–0442
Carollton	Arrow/Schweber Electronics	(214)380–6464
Dallas	FAI	(214)231–7195
	Future Electronics	(214)437–2437
	Hamilton/Hallmark	(214)553–4300
	Newark	(214)458–2528
El Paso	FAI	(915)577–9531
	Newark	(915)772–6367
Ft. Worth	Allied Electronics	(817)336–5401
Houston	Arrow/Schweber Electronics	(713)647–6868
	FAI	(713)952–7088
	Future Electronics	(713)785–1155
	Hamilton/Hallmark	(713)781–6100
	Newark	(713)894–9334
	Wyle Electronics	(713)784–9953
Richardson	PENSTOCK	(214)479–9215
	Wyle Electronics	(214)235–9953
San Antonio	FAI	(210)738–3330
	Newark	(210)734–7960

UTAH

Draper	Wyle Electronics	(801)523–2335
Salt Lake City	Arrow/Schweber Electronics	(801)973–6913
	FAI	(801)467–9696
	Future Electronics	(801)467–4448
	Hamilton/Hallmark	(801)266–2022
	Newark	(801)261–5660
West Valley City	Wyle Electronics	(801)974–9953

WASHINGTON

Bellevue	Almac Electronics Corp.	(206)643–9992
	PENSTOCK	(206)454–2371
Bothell	Future Electronics	(206)489–3400
Kirkland	Newark	(206)814–6230
Redmond	Hamilton/Hallmark	(206)882–7000
	Wyle Electronics	(206)881–1150
Seattle	FAI	(206)485–6616

WISCONSIN

Brookfield	Arrow/Schweber Electronics	(414)792–0150
	Future Electronics	(414)879–0244
	Wyle Electronics	(414)879–0434
Madison	Newark	(608)278–0177
Milwaukee	FAI	(414)792–9778
New Berlin	Hamilton/Hallmark	(414)780–7200
Wauwatosa	Newark	(414)453–9100

For changes to this information contact Technical Publications at FAX (602) 244-6560

AUTHORIZED DISTRIBUTORS – continued

CANADA

ALBERTA

Calgary

FAI	(403)291–5333
Future Electronics	(403)250–5550
Hamilton/Hallmark	(800)663–5500

Edmonton

FAI	(403)438–5888
Future Electronics	(403)438–2858
Hamilton/Hallmark	(800)663–5500

Saskatchewan

Hamilton/Hallmark	(800)663–5500

BRITISH COLUMBIA

Vancouver

Arrow Electronics	(604)421–2333
FAI	(604)654–1050
Future Electronics	(604)294–1166
Hamilton/Hallmark	(604)420–4101

MANITOBA

Winnipeg

FAI	(204)786–3075
Future Electronics	(204)944–1446
Hamilton/Hallmark	(800)663–5500

ONTARIO

Kanata

PENSTOCK	(613)592–6088

London

Newark	(519)685–4280

Mississauga

PENSTOCK	(905)403–0724
Newark	(905)670–2888

Ottawa

Arrow Electronics	(613)226–6903
FAI	(613)820–8244
Future Electronics	(613)727–1800
Hamilton/Hallmark	(613)226–1700

Toronto

Arrow Electronics	(905)670–7769
FAI	(905)612–9888
Future Electronics	(905)612–9200
Hamilton/Hallmark	(905)564–6060
Newark	(905)670–2888

QUEBEC

Montreal

Arrow Electronics	(514)421–7411
FAI	(514)694–8157
Future Electronics	(514)694–7710
Hamilton/Hallmark	(514)335–1000

Mt. Royal

Newark	(514)738–4488

Quebec City

Arrow Electronics	(418)687–4231
FAI	(418)682–5775
Future Electronics	(418)877–6666

INTERNATIONAL DISTRIBUTORS

AUSTRALIA

AVNET VSI Electronics (Aust.)	(61)2 9878–1299
Veltek Australia Pty Ltd	(61)3 9574–9300

AUSTRIA

EBV Elektronik	(43) 1 8941774
SEI/Elbatex GmbH	(43) 1 866420
Spoerle Electronic	(43) 1 31872700

BELGIUM

Spoerle Electronic	(32) 2 725 4660
EBV Elektronik	(32) 2 716 0010
SEI/Rodelco B.V.	(32) 2 460 0560

BULGARIA

Macro Group	(359) 2708140

CZECH REPUBLIC

Spoerle Electronic	(420) 2 731355
SEI/Elbatex	(420) 2 4763707
Macro Group	(420) 2 3412182

CHINA

Advanced Electronics Ltd.	(852)2 305–3633
AVNET WKK Components Ltd.	(852)2 357–8888
China El. App. Corp. XiaMan Co.	(86)10 6818–9750
Nanco Electronics Supply Ltd.	(852) 2 765–3025
	or (852) 2 333–5121
Qing Cheng Enterprises Ltd.	(852) 2 493–4202

DENMARK

Arrow Exatec	(45) 44 927000
Avnet Nortec A/S	(45) 44 880800
EBV Elektronik	(45) 39690511

ESTONIA

Arrow Field Eesti	(372) 6503288
Avnet Baltronic	(372) 6397000

FINLAND

Arrow Field OY	(358)97 775 71
Avnet Nortec OY	(358)96 13181
EBV Elektronik	(358)98557730

FRANCE

Arrow Electronique	(33) 1 49 78 49 78
Avnet Components	(33) 1 49 65 25 00
EBV Elektronik	(33) 1 64 68 86 00
Future Electronics	(33)1 69821111
Newark	(33)1–30954060
SEI/Scaib	(33) 1 69 19 89 00

GERMANY

Avnet E2000	(49) 89 4511001
EBV Elektronik GmbH	(49) 89 99114–0
Future Electronics GmbH	(49) 89–957 270
SEI/Jermyn GmbH	(49) 6431–5080
Newark	(49)2154–70011
Sasco Semiconductor	(49) 89–46110
Spoerle Electronic	(49) 6103–304–0

GREECE

EBV Elektronik	(30) 13414300

HONG KONG

AVNET WKK Components Ltd.	(852)2 357–8888
Nanshing Clr. & Chem. Co. Ltd	(852)2 333–5121

INDIA

Canyon Products Ltd	(91) 80 558–7758

INDONESIA

P.T. Ometraco	(62) 21 619–6166

IRELAND

Arrow	(353) 14595540
Future Electronics	(353) 6541330
Macro Group	(353) 16766904

ITALY

AVNET EMG SRL	(39) 2 381901
EBV Elektronik	(39) 2 660961
Future Electronics	(39) 2 660941
Silverstar Ltd. SpA	(39) 2 66 12 51

JAPAN

AMSC Co., Ltd.	81–422–54–6800
Fuji Electronics Co., Ltd.	81–3–3814–1411
Marubun Corporation	81–3–3639–8951
Nippon Motorola Micro Elec.	81–3–3280–7300
OMRON Corporation	81–3–3779–9053
Tokyo Electron Ltd.	81–3–5561–7254

KOREA

Jung Kwang Sa	(82)2278–5333
Lite–On Korea Ltd.	(82)2858–3853
Nasco Co. Ltd.	(82)23772–6800

LATVIA

Avnet	(371) 8821118

LITHUANIA

Macro Group	(370) 7751487

NETHERLANDS

HOLLAND

EBV Elektronik	(31) 3465 623 53
Spoerle Electronic	(31) 4054 5430
SEI/Rodelco B.V.	(31) 7657 227 00

NEW ZEALAND

AVNET VSI (NZ) Ltd	(64)9 636–7801

NORWAY

Arrow Tahonic A/S	(47)2237 8440
Avnet Nortec A/S Norway	(47)6684 6210
EBV Elektronik	(47)2267 1780

PHILIPPINES

Alexan Commercial	(63) 2241–9493

POLAND

EBV Elektronik	(48) 713 422944
Macro Group	(48) 22 224337
SEI/Elbatex	(48) 22 6254877
Spoerle Electronic	(48) 22 6060447

PORTUGAL

Amitron Arrow	(35) 114714806

ROMANIA

Macro Group	(401) 6343129

RUSSIA

Macro Group	(781) 25311476

SCOTLAND

EBV Elektronik	(44) 161 4993434

SINGAPORE

Future Electronics	(65) 479–1300
Strong Pte. Ltd	(65) 276–3996
Uraco Technologies Pte Ltd.	(65) 545–7811

SLOVAKIA

Macro Group	(42) 89634181

SLOVENIA

EBV Elektronik	(386) 611 330216
SEI/Elbatex	(386) 611 597113

SPAIN

Amitron Arrow	(34) 1 304 30 40
EBV Elektronik	(34) 1 804 32 56
SEI/Selco S.A.	(34) 1 637 10 11

SWEDEN

Arrow–Th:s	(46) 8 362970
Avnet Nortec AB	(46) 8 629 14 00
EBV Elektronik	(46) 405 92100

SWITZERLAND

EBV Elektronik	(41) 1 7456161
SEI/Elbatex AG	(41) 56 4375111
Spoerle Electronic	(41) 1 8746262

S. AFRICA

Advanced	(27) 11 4442333
Reuthec Components	(27) 11 8233357

THAILAND

Shapiphat Ltd.	(66)2221–0432 or 2221–5384

TAIWAN

Avnet–Mercuries Co., Ltd	(886)2 516–7303
Solomon Technology Corp.	(886)2 788–8989
Strong Electronics Co. Ltd.	(886)2 917–9917

TURKEY

EBV Elektronik	(90) 2164 631352

UNITED KINGDOM

Arrow Electronics (UK) Ltd	(44) 1 234 270027
Avnet/Access	(44) 1 462 488500
EBV Elektronik	(44) 1 628 783688
Future Electronics Ltd.	(44) 1 753 763000
Macro Group	(44) 1 628 60600
Newark	(44) 1 420 543333

For changes to this information contact Technical Publications at FAX (602) 244-6560

6/1/97

MOTOROLA WORLDWIDE SALES OFFICES

UNITED STATES

ALABAMA	
Huntsville	(205)464–6800
ALASKA	(800)635–8291
ARIZONA	
Phoenix	(602)302–8056
CALIFORNIA	
Calabasas	(818)878–6800
Irvine	(714)753–7360
Los Angeles	(818)878–6800
San Diego	(619)541–2163
Sunnyvale	(408)749–0510
COLORADO	
Denver	(303)337–3434
CONNECTICUT	
Wallingford	(203)949–4100
FLORIDA	
Clearwater	(813)524–4177
Maitland	(407)628–2636
Pompano Beach/Ft. Lauderdale	(954)351–6040
GEORGIA	
Atlanta	(770)729–7100
IDAHO	
Boise	(208)323–9413
ILLINOIS	
Chicago/Schaumburg	(847)413–2500
INDIANA	
Indianapolis	(317)571–0400
Kokomo	(317)455–5100
IOWA	
Cedar Rapids	(319)378–0383
KANSAS	
Kansas City/Mission	(913)451–8555
MARYLAND	
Columbia	(410)381–1570
MASSACHUSETTS	
Marlborough	(508)357–8207
Woburn	(617)932–9700
MICHIGAN	
Detroit	(810)347–6800
MINNESOTA	
Minnetonka	(612)932–1500
MISSOURI	
St. Louis	(314)275–7380
NEW JERSEY	
Fairfield	(201)808–2400
NEW YORK	
Fairport	(716)425–4000
Fishkill	(914)896–0511
Hauppauge	(516)361–7000
NORTH CAROLINA	
Raleigh	(919)870–4355
OHIO	
Cleveland	(216)349–3100
Columbus/Worthington	(614)431–8492
Dayton	(937)438–6800
OKLAHOMA	
Tulsa	(918)251–3414
or	(918)258–0933
OREGON	
Portland	(503)641–3681
PENNSYLVANIA	
Colmar	(215)997–1020
Philadelphia/Horsham	(215)957–4100
TENNESSEE	
Knoxville	(423)584–4841
TEXAS	
Austin	(512)502–2100
Houston	(713)251–0006
Plano	(972)516–5100
VIRGINIA	
Richmond	(804)285–2100
WASHINGTON	
Bellevue	(206)454–4160
Seattle (toll free)	(206)622–9960
WISCONSIN	
Milwaukee/Brookfield	(414)792–0122

Field Applications Engineering Available
Through All Sales Offices

CANADA

BRITISH COLUMBIA	
Vancouver	(604)606–8502
ONTARIO	
Ottawa	(613)226–3491
Toronto	(416)497–8181
QUEBEC	
Montreal	(514)333–3300

INTERNATIONAL

AUSTRALIA	
Melbourne	(61–3)98870711
Sydney	(61–2)99661071
BRAZIL	
Sao Paulo	55(11)815–4200
CHINA	
Beijing	86–10–68437222
Guangzhou	86–20–87537888
Shanghai	86–21–63747668
Tianjin	86–22–25325072
DENMARK	
Copenhagen	(45) 43488393
FINLAND	
Helsinki	358 9 6824 400
Direct Sales Lines	358 9 6824 4044
	358 9 6824 4045
FRANCE	
Paris	33134 635900
GERMANY	
Langenhagen/Hanover	49(511)786880
Munich	49 89 92103–0
Nuremberg	49 911 96–3190
Sindelfingen	49 7031 79 710
Wiesbaden	49 611 973050
HONG KONG	
Kwai Fong	852–2–610–6888
Tai Po	852–2–666–8333
INDIA	
Bangalore	91–80–5598615
ISRAEL	
Herzlia	972–9–9522333
ITALY	
Milan	39(2)82201
JAPAN	
Kyusyu	81–92–725–7583
Gotanda	81–3–5487–8311
Nagoya	81–52–232–3500
Osaka	81–6–305–1801
Sendai	81–22–268–4333
Takamatsu	81–878–37–9972
Tokyo	81–3–3440–3311
KOREA	
Pusan	82(51)4635–035
Seoul	82(2)554–5118
MALAYSIA	
Penang	60(4)228–2514
MEXICO	
Mexico City	52(5)282–0230
Guadalajara	52(36)21–8977
Zapopan Jalisco	52(36)78–0750
Marketing	52(36)21–2023
Customer Service	52(36)669–9160
NETHERLANDS	
Best	(31)4993 612 11
PHILIPPINES	
Manila	(63)2 822–0625
PUERTO RICO	
Rio Piedras	(787)282–2300
SCOTLAND	
East Kilbride	(44)1355 565447
SINGAPORE	(65)4818188
SPAIN	
Madrid	34(1)457–8204
or	34(1)457–8254
SWEDEN	
Solna	46(8)734–8800
SWITZERLAND	
Geneva	41(22)799 11 11
Zurich	41(1)730–4074
TAIWAN	
Taipei	886(2)717–7089
THAILAND	
Bangkok	66(2)254–4910
UNITED KINGDOM	
Aylesbury	44 1 (296)395252

NORTH AMERICA
FULL LINE REPRESENTATIVES

CALIFORNIA, Loomis	
Galena Technology Group	(916)652–0268
INDIANA, Indianapolis	
Bailey's Electronics	(317)848–9958
NEVADA, Reno	
Galena Tech. Group	(702)746–0642
NEW MEXICO, Albuquerque	
S&S Technologies, Inc.	(505)414–1100
UTAH, Salt Lake City	
Utah Comp. Sales, Inc.	(801)572–4010
WASHINGTON, Spokane	
Doug Kenley	(509)924–2322

NORTH AMERICA
HYBRID/MCM COMPONENT SUPPLIERS

Chip Supply	(407)298–7100
Elmo Semiconductor	(818)768–7400
Minco Technology Labs Inc.	(512)834–2022
Semi Dice Inc.	(310)594–4631

For changes to this information contact Technical Publications at FAX (602) 244-6560

1ATX35554-1 Printed in USA 7/97 BANTA CO. MOTO# 124 5,000 LITRISC

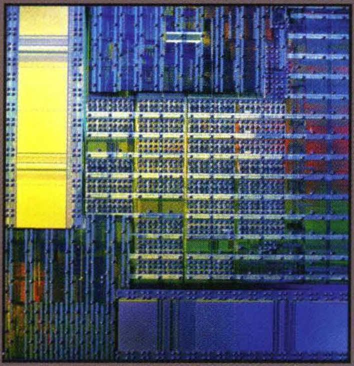
Attention!

This book is a companion to the *PowerPC Microprocessor Family: The Programming Environments*, referred to as *The Programming Environments Manual.* Note that the companion *Programming Environments Manual* exists in two versions. See the Preface for a description of the following two versions:

- *PowerPC Microprocessor Family: The Programming Environments,* Rev 1
  Order #: MPCFPE/AD
- *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, Rev 1*
  Order #: MPCFPE32B/AD

Call the Motorola LDC at 1-800-441-2447 (website: http://ldc.nmd.com) or contact your local sales office to obtain copies.

PowerPC™



MOTOROLA

Motorola Literature Distribution Centers:
USA/EUROPE: Motorola Literature Distribution,
P.O. Box 5405, Denver, Colorado 80217;
Tel.: 1-800-441-2447 or (303) 675-2140
JAPAN: Nippon Motorola Ltd. SPD; Strategic
Planning Office 4-32-1, Nishi-Gotanda Shinagawa-ku, Tokyo 141, Japan; Tel.: 81-3-5487-8488
ASIA-PACIFIC: Motorola Semiconductors H.K. Ltd.,
8B Tai Ping Industrial Park, 51 Ting Kok Road,
Tai Po, N.T., Hong Kong;
Tel.: 852-26629298

Mfax™: RMFAX0@email.sps.mot.com;
TOUCHTONE (602) 244-6609
US & Canada ONLY (800) 774-1848
INTERNET: http://www.motorola.com/sps/

Technical Information:
Motorola Inc. SPS Customer Support Center;
Tel. (800) 521-6274
Document Comments:
FAX (512) 891-2638,
Attn: RISC Applications Engineering
World Wide Web Address:
http://www.motorola.com/PowerPC/

MPC620UM/AD